

Java / Technical Details /  
Technical Article

# Querying JPA Entities with JPQL and Native SQL

Developer: Java Persistence

*By Yuli Vasiliev* Published September 2008

**Learn how to take advantage of the Java Persistence query language and native SQL when querying over JPA entities.**

In Java EE, the Java Persistence API (JPA) is the standard API for accessing relational databases, providing a simple and efficient way for managing the object/relational mapping (ORM) of regular Java objects (POJO) to relational data. Such Java objects are called JPA entities or just entities.

An entity is typically (but not always) associated with a single relational table in the underlying database, so that each entity's instance represents a certain row of that table in Java. Like relational tables, entities are typically related to each other with relationships such as one-to-one, one-to-many, many-to-one, or many-to-many. It's fairly obvious that Java applications dealing with entities require a standard mechanism to access and navigate entity instances. The Java

## DOWNLOAD

[Oracle Database](#)

[Oracle WebLogic Server](#)

[Sample Code](#)

Ask "Where do I download Java Runtime Environment?"

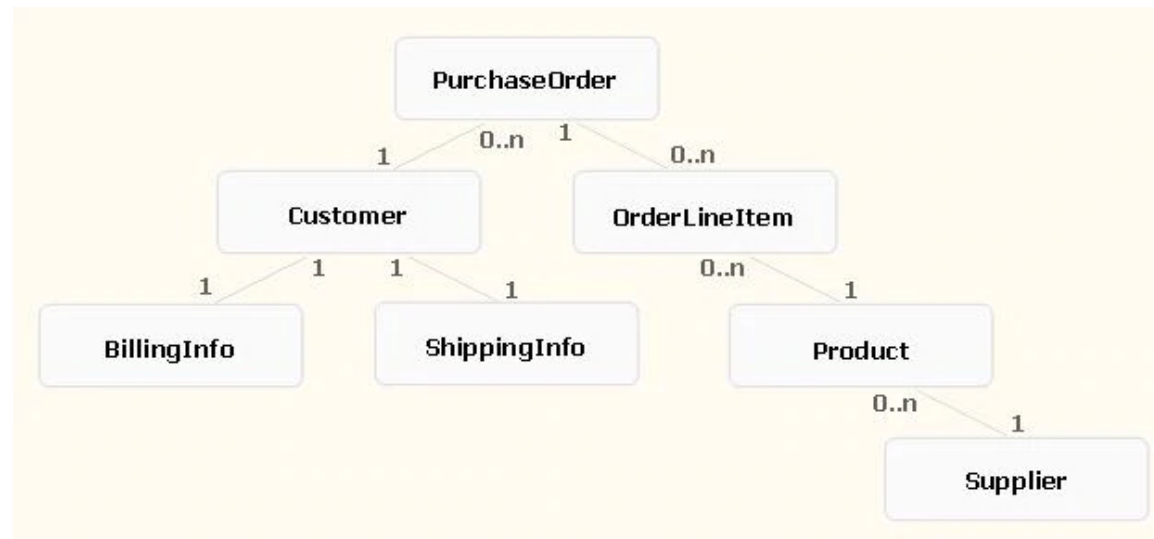


entities utilized within a Java application.

## Sample Application

The code snippets discussed in the article are taken from the Java source files used in the **sample application** accompanying the article. Looking through the sample archive, you may notice that this is a simple Web application based on the Java Servlet and Java Persistence API technologies. For simplicity, it doesn't use enterprise beans, issuing JPQL queries directly from within servlets. It doesn't mean, however, that you won't be able to utilize the JPQL queries discussed here in enterprise beans—you can define JPQL queries in any Java EE components.

Figure 1 illustrates the sample entities structure. As you can see, it contains a set of entities related to each other with relationships of different types. Such a branchy structure is needed in order to illustrate the use of JPQL join queries discussed in the Defining JPQL Joins section later in the article.



**Figure 1** Relationships among the entities utilized within the sample application

For a detailed instruction on how to set up and then launch the sample app, you can refer to the `readme.txt` file in the root directory of the sample archive.

Ask "Where do I download Java Runtime Environment?"



If you have some practical experience with databases, you've most likely already got your feet wet with SQL, the standard tool offering a set of statements for accessing and manipulating information in relational databases. In fact, there are many similarities between JPQL and SQL. Both are used to access and manipulate database data, in the long run. And both use nonprocedural statements—commands recognized by a special interpreter. Furthermore, JPQL is similar to SQL in its syntax.

The main difference between JPQL and SQL lies in that the former deals with JPA entities, while the latter deals directly with relational data. As a Java developer, you also maybe interested to learn that using JPQL, unlike SQL/JDBC, eliminates the need for you to use JDBC API from your Java code—the container does all this work for you behind the scenes.

JSQL lets you define queries using one of the following three statements: `SELECT`, `UPDATE`, or `DELETE`. It's interesting to note that the `EntityManager` API interface offers methods that can also be used to perform retrieve, update and delete operations over entities. In particular, these are `find`, `merge`, and `remove` methods. The use of those methods, however, is typically limited to a single entity instance, unless cascading takes effect, of course. In contrast, JSQL statements do not have such a limitation—you can define bulk update and delete operations over sets of entities, and define queries returning sets of entity instances.

To issue a JPQL query from within your Java code, you have to utilize appropriate methods of the EntityManager API and Query API, performing the following general steps:

1. Obtain an instance of `EntityManager`, using injection or explicitly through an `EntityManagerFactory` instance.
2. Create an instance of `Query` by invoking an appropriate `EntityManager`'s method, such as `createQuery`.
3. Set a query parameter or parameters, if any, using an appropriate `Query`'s `setParameter` method.
4. If needed, set the maximum number of instances to retrieve and/or specify the position of the first instance to retrieve, using the `setMaxResults` and/or `setFirstResult` `Query`'s methods.
5. If needed, set a vendor-specific hint, using the `setHint` `Query`'s method.
6. If needed, set the flush mode for the query execution with the `setFlushMode` `Query`'s method, overriding the entity manager's flush mode.
7. Execute the query using an appropriate `Query`'s method: `getSingleResult` or `getResultList`. In the case of an update or delete operation, though, you must use the `executeUpdate` method, which returns the number of entity instances updated or deleted.

The full list of the EntityManager interface methods, as well as the Query API interface methods, can be found in the Enterprise JavaBeans 3.0 Specification: Java Persistence API document, which is part of [JSR-220](#).



COPY

```
...
@PersistenceUnit
private EntityManagerFactory emf;
public void doGet()
...
EntityManager em = emf.createEntityManager();
PrintWriter out = response.getWriter();
List<Customer> arr_cust = (List<Customer>)em.createQuery("SELECT c FROM Customer c")
.getResultList();
out.println("List of all customers: "+"");
Iterator i = arr_cust.iterator();
Customer cust;
while (i.hasNext()) {
    cust = (Customer) i.next();
    out.println(cust.getCust_id()+"");
    out.println(cust.getCust_name()+"");
    out.println(cust.getEmail()+"");
    out.println(cust.getPhone()+"");
    out.println("-----" + "");
}
...

```

Of special interest here are the `createQuery` method of the `EntityManager` instance and the `getResultList` method of the `Query` instance. The `EntityManager`'s `createQuery` is used to create the `Query` instance whose `getResultList` method is then used to execute the JPQL query passed to `createQuery` as the parameter. As you might guess, the `Query`'s `getResultList` method returns the result of a query as a `List` whose elements, in this particular example, are cast to type `Customer`.

If you need to retrieve a single result, the `Query` API interface offers the `getSingleResult` method, as shown in the following example. Note, however, that using `getSingleResult` will cause an exception if you get multiple results back.

Ask "Where do I download Java Runtime Environment?"

↑ Parameter, you can bind both named and positional parameters. Here, though, you bind a named parameter.

COPY

```
...
Integer cust_id =2;
Customer cust = (Customer)em.createQuery("SELECT c FROM Customer c WHERE c.cust_id=:cust_id")
.setParameter("cust_id", cust_id)
.getSingleResult();
out.println("Customer with id "+cust.getCust_id()+" is: "+ cust.getCust_name()+"");
...
```


It is interesting to note that using a SELECT JPQL statement is not the only way to go when it comes to retrieving a single entity instance. Alternatively, you might utilize the EntityManager's find method, which lets you retrieve a single entity instance based on the entity's id passed in as the parameter.

In some situations, you may need to retrieve only some information from the target entity instance or instances, defining a JPQL query against a certain entity field or fields. This is what the above snippet would look like, if you need to retrieve only the value of the cust\_name field of the Customer entity instance queried here:

COPY

```
...
Integer cust_id =2;
String cust_name = (String)em.createQuery("SELECT c.cust_name FROM Customer c WHERE c.cust_id=:cust_id")
.setParameter("cust_id", cust_id)
.getSingleResult();
out.println("Customer with id "+cust_id+" is: "+cust_name+"");
...
```

Ask "Where do I download Java Runtime Environment?"



```

...
List<String> arr_cust_name = (List<String>)em.createQuery("SELECT c.cust_name FROM Customer c")
.getResultList();
out.println("List of all customers: "+"<br/>");
Iterator i = arr_cust_name.iterator();
String cust_name;
while (i.hasNext()) {
    cust_name = (String) i.next();
    out.println(cust_name+"<br/>");
}
...

```

Turning back to SQL, you might recall that the select list of a SQL query can be comprised of several fields from the table or tables specified in the FROM clause. In JPQL, you also can use a comprised select list, selecting the data only from the entity fields of interest. In that case, however, you need to create the class to which you will cast the query result. In the following section, you will see an example of a JPQL join query whose select list is comprised of the fields derived from more than one entity.

## Defining JPQL Joins

Like SQL, JPQL lets you define join queries. In SQL, however, you normally define a join that combines records from two or more tables and/or views, including only required fields from those tables and views in the select list of the join query. In contrast, the select list of a JPQL join query typically includes a single entity or even a single entity field. The reason for this lies in the nature of the JPA technology. Once you obtain an entity instance, you can then navigate to its related instances using corresponding getter methods. This approach makes it unnecessary for you to define a query that will return all related entity instances of interest at once.

For example, to obtain information about orders along with their line items in SQL, you would need to define a join query on both the purchaseOrders and orderLineItems tables, specifying the fields from the both tables in the select list of the query. When using JPQL, however, you might define a query only over the PurchaseOrder entity, and then navigate to corresponding OrderLineItem instances using the PurchaseOrder's getOrderLineItems method as required. In this example, you might want to define a JPQL query over the PurchaseOrder and OrderLineItem entities only if you need to filter retrieved PurchaseOrder instances based on a condition or conditions applied to OrderLineItem

Ask "Where do I download Java Runtime Environment?"



turn back to Figure 1 shown in the Sample Application section earlier in the article.

[COPY](#)

```
...
Double max = (Double) em.createQuery("SELECT MAX(p.price) FROM PurchaseOrder
o JOIN o.orderLineItems l JOIN l.product p JOIN p.supplier s WHERE s.sup_name = 'Tortuga Trading'")
.getSingleResult();
out.println("The highest price for an ordered product supplied by Tortuga Trading: "+ max + "<br/>");
...
```

In the above example, you use the MAX aggregate function in the SELECT clause of the join query in order to determine the highest price product, of those that have been supplied by Tortuga Trading and have been ordered at least once.


A more common situation, however, is when you need to calculate, say, the total price of the ordered products, which have been supplied by a certain supplier. This is where the SUM aggregate function may come in handy. In SQL, such a join query might look like this:

[COPY](#)

```
SELECT SUM(p.price*l.quantity) FROM purchaseorders o JOIN orderlineitems l ON
o.pono=l.pono JOIN products p ON l.prod_id=p.prod_id JOIN suppliers s ON
p.sup_id=s.sup_id WHERE sup_name ='Tortuga Trading';
```

Unfortunately, the SUM function used in JPQL does not allow you to pass an arithmetic expression as the argument. What this means in practice is that you won't be able to pass `p.price*l.quantity` as the argument to the JPQL's SUM. However, there are ways to work around this issue. In the following example, you define class `LineItemSum` whose constructor is then used in the select list of the query, taking `p.price` and `l.quantity` as the parameters. What the `LineItemSum` constructor does is multiply `p.price` by `l.quantity`, saving the result to its `rslt` class variable. Next, you can iterate through the `LineItemSum` list retrieved by the query, summing the values of the `LineItemSum`'s `rslt` variable. The following snippet shows how all this can be implemented in code:

Ask "Where do I download Java Runtime Environment?"



```

package jpqlexample.servlets;
...

class LineItemSum {
private Double price;
private Integer quantity;
private Double rslt;
public LineItemSum (Double price, Integer quantity){
this.rslt = quantity*price;
}
public Double getRslt () {
return this.rslt;
}
public void setRslt (Double rslt) {
this.rslt = rslt;
}
}

public class JpqlJoinsServlet extends HttpServlet {
...

public void doGet(
...
List<LineItemSum> arr = (List<LineItemSum>)em.createQuery
("SELECT NEW jpqlexample.servlets.LineItemSum(p.price, l.quantity) FROM PurchaseOrder o
JOIN o.orderLineItems l JOIN l.product p JOIN p.supplier s WHERE s.sup_name = 'Tortuga Trading'")
.getResultList();
Iterator i = arr.iterator();
LineItemSum lineItemSum;
Double sum = 0.0;
while (i.hasNext()) {
lineItemSum = (LineItemSum) i.next();
sum = sum + lineItemSum.getRslt();
}
}

```

Ask "Where do I download Java Runtime Environment?"



↑ ng other things, the above example illustrates how you might use a custom Java class, not an entity class, in the JPQL query's select list that includes the fields derived from more than one entity, casting the result of the query to that class. In most cases, however, you will have to deal with queries that receive an instance or a list of instances of a certain entity.

## Retrieved Entity Instances and the Current Persistence Context

The query results in the article examples so far have been simply printed out. In real-world applications, though, you may need to perform some further operations on the query results. For example, you may need to update the retrieved instances and then persist them back to the database. This raises the question: are the instances being retrieved by a JPQL query ready to be further processed by the application, or some additional steps are required to make them ready for that? In particular, it would be interesting to learn in what state, concerning the current persistence context, retrieved entity instances are.

If you have some experience with Java Persistence, you should know what a persistence context is. To recap, a persistence context is a set of entity instances managed by the EntityManager instance associated with that context. In the preceding examples, you used the EntityManager's createQuery method to create an instance of Query for executing a JPQL query. Actually, the EntityManager API includes more than twenty methods to manage the lifecycle of entity instances, control transactions, and create instances of Query whose methods are then used to execute the query specified and retrieve the query result.

With respect to a persistence context, an entity instance can be in one of the following four states: new, managed, detached, or removed. Using an appropriate EntityManager's method, you can change the state of a certain entity instance as needed. It's interesting to note, however, that only instances in managed state are synchronized to the database, when flushing to the database occurs. To be precise, the entity instances in the removed state are also synchronized, meaning the database records corresponding to those instances are removed from the database.

By contrast, instances in the new or detached state won't be synchronized to the database. For example, if you create a new PurchaseOrder instance and then invoke the EntityManager's flush method, another record will not appear in the purchaseOrders table to which the PurchaseOrder entity is mapped. This is because that new PurchaseOrder instance has not been attached to the persistence context. Here is what the code might look like:

COPY

Ask "Where do I download Java Runtime Environment?"



```
PurchaseOrder ord = new PurchaseOrder();
ord.setOrder_date(new Date());
ord.setCustomer(cust);
em.getTransaction().commit();
...
```

To fix the problem, you need to invoke the EntityManager's persist method for the new PurchaseOrder instance before invoking the flush, as shown in the following example:

COPY

```
...
em.getTransaction().begin();
Customer cust = (Customer) em.find(Customer.class, 1);
PurchaseOrder ord = new PurchaseOrder();
ord.setOrder_date(new Date());
ord.setCustomer(cust);
em.persist(ord);
em.getTransaction().commit();
...
```

Alternatively, provided you've set the cascade option to PERSIST or ALL when defining the relationship with PurchaseOrder in the Customer entity, you might add the newly created PurchaseOrder instance to the list of the orders associated with the customer instance, replacing the persist operation with the following:

COPY

```
cust.getPurchaseOrders().add(ord);
```

The above discussion regarding entity instance states leads us to the interesting question of whether the entity instances retrieved by a JPQL

Ask "Where do I download Java Runtime Environment?"



text. This means entity instances retrieved by a JPQL query become automatically managed. You can, for example, change the value of a retrieved instance's field and then synchronize that change to the database by invoking the EntityManager's flush method or committing the current transaction. You don't need to worry about the state of the instances associated with the retrieved instances either. The fact is that the first time you access an associated instance it becomes managed automatically. Here is a simple example showing how all this works in practice:

COPY

```
...
em.getTransaction().begin();
PurchaseOrder ord = (PurchaseOrder)em.createQuery("SELECT o FROM PurchaseOrder o WHERE o.pono = 1")
    .getSingleResult();
List<OrderLineItem> items = ord.getOrderLineItems();
Integer qnt = items.get(0).getQuantity();
out.println("Quantity of the first item : "+ qnt + "<br/>");
items.get(0).setQuantity(qnt+1);
qnt = items.get(0).getQuantity();
em.getTransaction().commit();
out.println("Quantity of the first item : "+ qnt + "<br/>");
...
```

Note that you don't invoke the persist method for the retrieved PurchaseOrder instance, nor for its related OrderLineItem instance being modified here. In spite of this fact, the changes made to the first line item in the order will be persisted to the database upon committing the transaction. This happens because both the retrieved entity instances and their associations are automatically attached to the current persistence context. As mentioned earlier, the former become managed when they are retrieved, and the latter are attached to the context as you access them.

In some situations, you may want associations to be attached to the context upon the query execution, rather than upon the first access. This is where a FETCH JOIN comes in handy. Say, you want to obtain all the orders belonging to a certain customer, upon retrieving that customer instance. This approach guarantees you're dealing with the customer orders available at the time of query execution. If, for example, a new order is added to another context and then synchronized to the database before you first time access the orders list associated with the customer instance retrieved, you won't see this change until you refresh the state of the customer instance from the database. In the following snippet, you use the join query that

Ask "Where do I download Java Runtime Environment?"



```
...
Customer cust = (Customer)em.createQuery("SELECT DISTINCT c FROM Customer c
LEFT JOIN FETCH c.purchaseOrders WHERE c.cust_id=1")
    .getSingleResult();
...
List<PurchaseOrder> orders = cust.getPurchaseOrders();
...
```

Being not part of the explicit query result, the PurchaseOrder entity instances associated with the Customer instance retrieved here are also retrieved and attached to the current persistence context upon the query execution.

## Utilizing Native SQL Queries

It's interesting to note that you're not limited to JPQL when defining queries to be then executed with Query API. You may be surprised to learn that the EntityManager API offers methods for creating instances of Query for executing native SQL statements. The most important thing to understand about native SQL queries created with EntityManager methods is that they, like JPQL queries, return entity instances, rather than database table records. Here is a simple example of a dynamic native SQL query:

[COPY](#)

```
...
List<Customer> customers = (List<Customer>)em.createNativeQuery

("SELECT * FROM customers", jpqlexample.entities.Customer.class)
    .getResultList();
Iterator i = customers.iterator();
Customer cust;
out.println("Customers: " + "<br/>");
while (i.hasNext()) {
```

Ask "Where do I download Java Runtime Environment?"



...

JPQL is still evolving, and doesn't have many of those important features available in SQL. In the Defining JPQL Joins earlier section, you saw an example of JPQL's incompleteness: you had to do a lot of work on your own because the JPQL's SUM aggregate function cannot take an arithmetic expression as the parameter. In contrast, the SQL's SUM function doesn't have such a limitation. So, this is a good example of where replacing JPQL with native SQL could be efficient. The following code illustrates how you might simplify things in this particular example by choosing native SQL over JPQL:

COPY

```
...
String sup_name = "Tortuga Trading";
BigDecimal sum = (List)em.createNativeQuery("SELECT SUM(p.price*l.quantity)
FROM orders o JOIN orderlineitems l ON o.pono=l.pono
JOIN products p ON l.prod_id=p.prod_id
JOIN suppliers s ON p.sup_id=s.sup_id WHERE sup_name =?1")
.setParameter(1, sup_name)
.getSingleResult();
out.println("The total cost of the ordered products supplied by Tortuga Trading: " + sum + "<br/>");
...
```

Among other things, the above example illustrates that you can bind arguments to native query parameters. In particular, you can bind arguments to positional parameters in the same way as if you were dealing with a JPQL query.

The main disadvantage of native queries is complexity of result binding. In the example, the query produces a single result of a simple type, thus avoiding this problem. In practice, however, you often have to deal with a result set of a complex type. In this case, you will have to declare an entity to which you can map your native query, or define a complex result set mapped to multiple entities or to a blend of entities and scalar results.

## Using Stored Procedures

Ask "Where do I download Java Runtime Environment?"



underlying structure, you will have to adjust the native queries concerned in your servlets and/or other application components, having to compile and redeploy those components after that. To work around this problem, while still using native queries, you might take advantage of stored procedures, moving complex SQL queries into programs stored and executed inside the database, and then calling those stored programs instead of making direct calls to the underlying tables. What this means in practice is that stored procedures may save you the trouble of dealing with the underlying tables directly from the queries that are hard-coded in your Java code. The benefit to this approach is that in most cases you won't need to modify your Java code to follow the changes in the underlying database structure. Instead, only the stored procedures will need fixing.

Turning back to the example discussed in the preceding section, you might move the complex join query used there into a stored function, created as follows:

COPY

```
CREATE OR REPLACE FUNCTION sum_total(supplier VARCHAR2)
RETURN NUMBER AS
sup_sum NUMBER;
BEGIN
SELECT SUM(p.price*l.quantity) INTO sup_sum FROM orders o
JOIN orderlineitems l ON o.pono=l.pono
JOIN products p ON l.prod_id=p.prod_id
JOIN suppliers s ON p.sup_id=s.sup_id
WHERE sup_name = supplier;
RETURN sup_sum;
END;
/
```

This simplifies the native query used in your Java code and removes the dependency from the underlying tables:

COPY

```
...
String sup_name ="Tortuga Trading";
```

Ask "Where do I download Java Runtime Environment?"



```
out.println("The total cost of the ordered products supplied by Tortuga Trading: " + sum + "<br/>");  
...
```

## Conclusion

As you learned in this article, JPQL is a powerful tool when it comes to accessing relational data from within Java applications utilizing Java Persistence. With JPQL, unlike with SQL/JDBC, you define queries over JPA entities mapped to underlying database tables rather than querying those tables directly, thus dealing with a layer of abstraction that hides database details from the business logic layer. You also learned that JPQL is not the only option when it comes to creating queries over JPA entities—in some situations using native SQL queries is more convenient.

**Yuli Vasiliev** is a software developer, freelance author, and consultant currently specializing in open source development, Java technologies, databases, and SOA. He is the author of *Beginning Database-Driven Application Development in Java EE: Using GlassFish* (Apress, 2008).

### Resources for

Developers  
Startups  
Students and Educators

### Partners

Oracle PartnerNetwork  
Find a Partner  
Log in to OPN

### Emerging Technologies

Artificial Intelligence  
Internet of Things (IoT)  
More Solutions

### How We Operate

Corporate Security  
Practices  
Doing Business with  
Oracle  
Oracle@Oracle

### Contact Us

US Sales: +1.800.633.0738  
Global Contacts  
Subscribe to emails