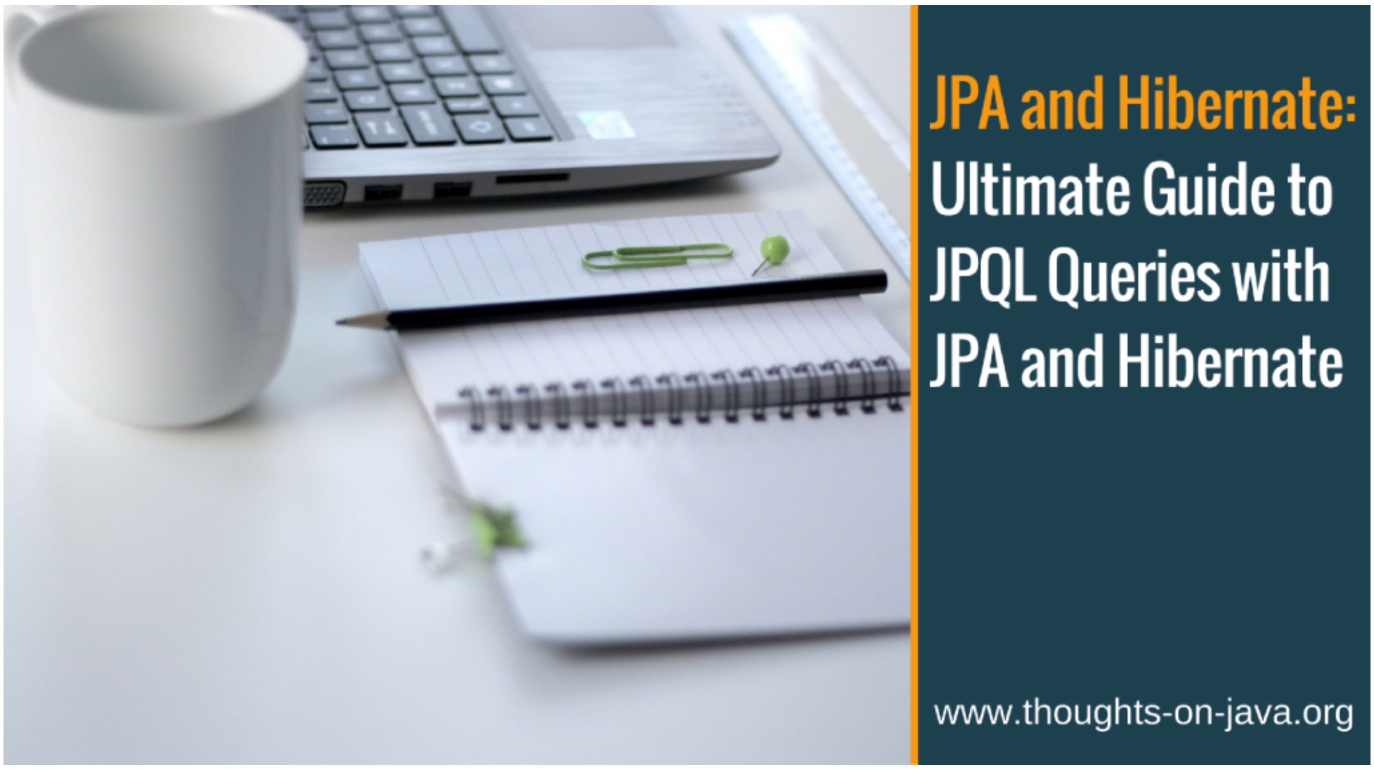
Ultimate Guide to JPQL Queries with JPA and Hibernate

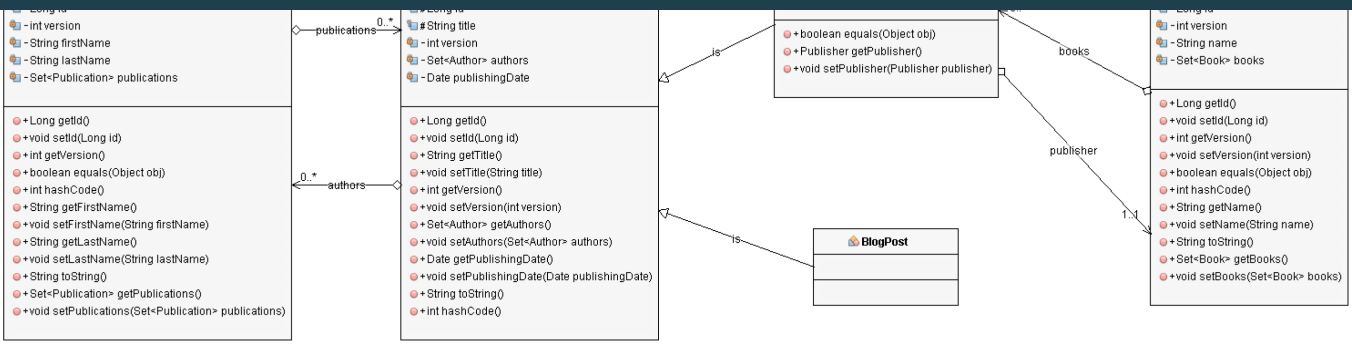
by Thorben Janssen — Leave a Comment



JPQL is a powerful query language that allows you to define database queries based on your entity model. It's structure and syntax is very similar to SQL. But there is an important difference that I want to point out before I walk you through the different parts of a JPQL query.

JPQL uses the entity object model instead of database tables to define a query. That makes it very comfortable for us Java developers, but you have to keep in mind that the database still uses SQL. Hibernate, or any other JPA implementation, has to transform the JPQL query into SQL. It is, therefore, a good practice to activate the logging of the SQL statements during development to check the generated SQL statements.

Entity Model


-int version
-String firstName
-String lastName
-Set<Publication> publications
+Long getId()
+void setId(Long id)
+int getVersion()
+boolean equals(Object obj)
+int hashCode()
+String getFirstName()
+void setFirstName(String firstName)
+String getLastName()
+void setLastName(String lastName)
+String toString()
+Set<Publication> getPublications()
+void setPublications(Set<Publication> publications)
publications
0..*
authors
0..*
#String title
-int version
-Set<Author> authors
-Date publishingDate
+Long getId()
+void setId(Long id)
+String getTitle()
+void setTitle(String title)
+int getVersion()
+void setVersion(int version)
+Set<Author> getAuthors()
+void setAuthors(Set<Author> authors)
+Date getPublishingDate()
+void setPublishingDate(Date publishingDate)
+String toString()
+int hashCode()
is
is
+boolean equals(Object obj)
+Publisher getPublisher()
+void setPublisher(Publisher publisher)
BlogPost
books
publisher
1..1
-int version
-String name
-Set<Book> books
+Long getId()
+void setId(Long id)
+int getVersion()
+void setVersion(int version)
+boolean equals(Object obj)
+int hashCode()
+String getName()
+void setName(String name)
+String toString()
+Set<Book> getBooks()
+void setBooks(Set<Book> books)

It consists of an *Author* who has written one or more *Publication*s. A *Publication* can be a *Book* or a *BlogPost*. A *Book* might have been published by one *Publisher*.

Get this post as a free cheat sheet!

I prepared a free cheat sheet with the most important information and code snippets of this post. As always, you can download it for free from the Thoughts on Java Library.

JOIN NOW!

Already a member? Login here.

Selection – The *FROM* clause

The *FROM* clause defines from which entities the data gets selected. Hibernate, or any other JPA implementation, maps the entities to the according database tables.

The syntax of a JPQL *FROM* clause is similar to SQL but uses the entity model instead of table or column names. The following code snippet shows a simple JPQL query in which I select all *Author* entities.

```
SELECT a FROM Author a
```

From.java hosted with ❤ by GitHub view raw

As you can see, I reference the *Author* entity instead of the author table and assign the identification variable *a* to it. The identification variable is often called alias and is similar to a variable in your Java code. It is used in all other parts of the query to reference this entity.

If you want to select data from multiple entities, like all authors and the books they've written, you have to join the entities in the *FROM* clause. The easiest way to do that is to use the defined associations of an entity like in the following code snippet.

```
SELECT a, b FROM Author a JOIN a.books b
```

InnerJoin.java hosted with ❤ by GitHub view raw

The definition of the Author entity provides all information Hibernate needs to join it to the Book entity, and you don't have to provide an additional *ON* statement. In this example, Hibernate uses the primary keys of the Author and Book entity to join them via the association table of the many-to-many association.

*JOIN*s of unrelated entities are not supported by the JPA specification, but you can use a theta join which creates a cartesian product and restricts it in the *WHERE* clause to the records with matching foreign and primary keys. I use this approach in the following example to join the *Book* with the *Publisher* entities.

```
SELECT b, p FROM Book b, Publisher p WHERE b.fk_publisher = p.id
```

ThetaJoin.java hosted with ❤ by GitHub view raw

You can read more about this workaround and Hibernates proprietary support for JOINs of unrelated entities in How to join unrelated entities with JPA and Hibernate.

Left Outer Joins

*INNER JOIN*s, like the one in the previous example, require that the selected entities fulfill the join condition. The query returned only the *Author* entities with associated Book entities but not the ones for which the database doesn't contain a *Book* entity. If you want to include the authors without published books, you have to use a *LEFT JOIN*, like in the following code snippet.

```
SELECT a, b FROM Author a LEFT JOIN a.books b
```

LeftOuterJoins.java hosted with ❤ by GitHub view raw

Additional Join Conditions

JoinCondition.java hosted with ❤ by GitHub view raw

Path expressions or implicit joins

Path expressions create implicit joins and are one of the benefits provided by the entity model. You can use the '.' operator to navigate to related entities like I do in the following code snippet.

```
SELECT b FROM Book b WHERE b.publisher.name LIKE '%es%
```

PathExpressions.java hosted with ❤ by GitHub view raw

As you can see, I use the '.' operator to navigate via the publisher attribute of the *Book* entity *b* to the related *Publisher* entities. That creates an implicit join between the *Book* and *Publisher* entity which will be translated into an additional join statement in the SQL query.

Polymorphism and Downcasting

Polymorphism

When you choose an inheritance strategy that supports polymorphic queries, your query selects all instances of the specified class and its subclasses. With the model in the example for this blog post, you can, for example, select all *Publication* entities, which are either *Book* or *BlogPost* entities.

```
SELECT p FROM Publication p
```

Polymorphism.java hosted with ❤ by GitHub view raw

Or you can select a specific subtype of a *Publication*, like a *BlogPost*.

```
SELECT b FROM BlogPost b
```

PolymorphismSubtype.java hosted with ❤ by GitHub view raw

Downcasting

Since JPA 2.1, you can also use the *TREAT* operator for downcasting in *FROM* and *WHERE* clauses. I use that in the following code snippet to select all *Author* entities with their related

Downcasting.java hosted with ❤ by GitHub view raw

ATTENTION: There are several issues with the implementation of *TREAT* in Hibernate 5.1. Based on my experiments, Hibernate 5.1 handles *TREAT* only, if it is written in lower case and used in the *WHERE* clause. The treat operator in this example is ignored by Hibernate 5.1.0.Final.

Restriction – The *WHERE* clause

The next important part of a JPQL query is the *WHERE* clause which you can use to restrict the selected entities to the ones you need for your use case. The syntax is very similar to SQL, but JPQL supports only a small subset of the SQL features. If you need more sophisticated features for your query, you can use a native SQL query.

JPQL supports a set of basic operators to define comparison expressions. Most of them are identical to the comparison operators supported by SQL and you can combine them with the logical operators *AND*, *OR* and *NOT* into more complex expressions.

Operators for single-valued expressions:

- Equal: author.id = 10
- Not equal: author.id <> 10
- Greater than: author.id > 10
- Greater or equal then: author.id => 10
- Smaller than: author.id < 10
- Smaller or equal then: author.id <= 10
- Between: author.id *BETWEEN* 5 *and* 10

with *NOT* to exclude all Authors with a matching firstName.
- Is null: author.firstName *IS NULL*
  You can negate the operator with *NOT* to restrict the query result to all Authors who's firstName *IS NOT NULL*.
- In: author.firstName IN ('John', 'Jane')
  Restricts the query result to all Authors with the first name John or Jane.

Operators for collection expressions:

- Is empty: author.books *IS EMPTY*
  Restricts the query result to all *Author*s without associated *Book* entities. You can negate the operator (*IS NOT EMPTY*) to restrict the query result to all *Author*s with associated *Book* entities.
- Size: *size(*author.books*)* > 2
  Restricts the query result to all *Author*s who are associated with more than 2 *Book* entities.
- Member of: :myBook *member of* author.books
  Restricts the query result to all *Author*s who are associated with a specific *Book* entity.

You can use one or more of the operators to restrict your query result. The following query returns all Author entities with a *firstName* attribute that contains the String "and" and an id attribute greater or equal 20 and who have written at least 5 books.

```
SELECT a FROM Author a WHERE a.firstName like '%and%' and a.id >= 20 and size(author.books) >= 5
```

ComparisonExpression.java hosted with ❤ by GitHub view raw

Projection – The *SELECT* clause

The projection of your query defines which information you want to retrieve from the database. This part of the query is very different to SQL. In SQL, you specify a set of database columns and functions as your projection. You can do the same in JPQL by selecting a set of entity attributes or functions as scalar values, but you can also define entities or constructor calls as your projection. Hibernate, or any other JPA implementation, maps this information to a set of database columns and function calls to define the projection of the generated SQL statement.

Entities are the most common projection in JPQL queries. Hibernate uses the mapping information of the selected entities to determine the database columns it has to retrieve from the database. It then maps each row of the result set to the selected entities.

```
SELECT a FROM Author a
```

EntityProjection.java hosted with ❤ by GitHub view raw

It's comfortable to use entities as your projection. But you should always keep in mind that all entities are managed by the persistence context which creates an overhead for read-only use cases. In these situations, it's better to use scalar values or a constructor reference as a projection.

Scalar values

Scalar value projections are very similar to the projections you know from SQL. Instead of database columns, you select one or more entity attributes or the return value of a function call with your query.

```
SELECT a.firstName, a.lastName FROM Author a
```

ScalarValueProjection.java hosted with ❤ by GitHub view raw

Constructor references

Constructor references are a good projection for read-only use cases. They're more comfortable to use than scalar value projections and avoid the overhead of managed entities.

JPQL allows you to define a constructor call in the *SELECT* clause. You can see an example of it in the following code snippet. You just need to provide the fully qualified class name and specify the constructor parameters of an existing constructor. Similar to the entity projection, Hibernate generates an SQL query which returns the required database columns and uses the constructor reference to instantiate a new object for each record in the result set.

```
SELECT new org.thoughts.on.java.model.AuthorValue(a.id, a.firstName, a.lastName) FROM Author a
```

ConstructorProjection.java hosted with ❤ by GitHub view raw supports this operator as well.

```
SELECT DISTINCT a.lastName FROM Author a
```

Functions

Functions are another powerful feature of JPQL that you probably know from SQL. It allows you to perform basic operations in the *WHERE* and *SELECT* clause. You can use the following functions in your query:

- *upper(String s)*: transforms *String s* to upper case
- *lower(String s)*: transforms *String s* to lower case
- *current_date()*: returns the current date of the database
- *current_time()*: returns the current time of the database
- *current_timestamp()*: returns a timestamp of the current date and time of the database
- *substring(String s, int offset, int length)*: returns a substring of the given *String s*
- *trim(String s)*: removes leading and trailing whitespaces from the given *String s*
- *length(String s)*: returns the length of the given *String s*
- *locate(String search, String s, int offset)*: returns the position of the *String search* in *s*. The search starts at the position *offset*
- *abs(Numeric n)*: returns the absolute value of the given number
- *sqrt(Numeric n)*: returns the square root of the given number
- *mod(Numeric dividend, Numeric divisor)*: returns the remainder of a division
- *treat(x as Type)*: downcasts *x* to the given *Type*
- *size(c)*: returns the size of a given *Collection c*
- *index(orderdCollection)*: returns the index of the given value in an ordered Collection

entity attributes that are not part of the function in the *GROUP BY* clause.

The following code snippet shows an example that uses the aggregate function *count()* to count how often each last name occurs in the Author table.

```
SELECT a.lastName, COUNT(a) FROM Author a GROUP BY a.lastName
```

GroupBy.java hosted with ♥ by GitHub view raw

The *HAVING* clause is similar to the *WHERE* clause and allows you to define additional restrictions for your query. The main difference is that the restrictions defined in a *HAVING* clause are applied to a group and not to a row.

I use it in the following example to select all last names that start with a 'B' and count how often each of them occurs in the *Author* table.

```
SELECT a.lastName, COUNT(a) AS cnt FROM Author a GROUP BY a.lastName HAVING a.lastName LIKE 'B%'
```

Having.java hosted with ♥ by GitHub view raw

Ordering – The ORDER BY clause

You can define the order in which the database shall return your query results with an *ORDER BY* clause. Its definition in JPQL is similar to SQL. You can provide one or more entity attributes to the *ORDER BY* clause and specify an ascending (*ASC*) or a descending (*DESC*) order.

The following query selects all *Author* entities from the database in an ascending order of their *lastName* attributes. All *Author*s with the same *lastName* are returned in descending order of their *firstName*.

```
SELECT a FROM Author a ORDER BY a.lastName ASC, a.firstName DESC
```

OrderBy.java hosted with ♥ by GitHub view raw

Subselects

A subselect is a query embedded into another query. It's a powerful feature you probably know

SELECT or

```
SELECT a FROM Author a WHERE (SELECT count(b) FROM Book b WHERE a MEMBER OF b.authors ) > 1
```

Subselect.java hosted with ❤ by GitHub view raw

Summary

As you've seen, the syntax and structure of JPQL are pretty similar to SQL. This makes JPQL easy to learn when you're already familiar with SQL. But you have to keep in mind that SQL supports a lot of advanced features that you can't use with JPQL. If you need one or more of them for a specific use case, you should use a native SQL query.

Filed Under: Featured, Hibernate, JPA, JPA2.1

Hibernate Tips
Implement your persistence layer with ease.
Hibernate Tips
More than 70 solutions to common Hibernate problems
Thorben Janssen

Want to learn more about Hibernate?

Join one of my Hibernate courses.

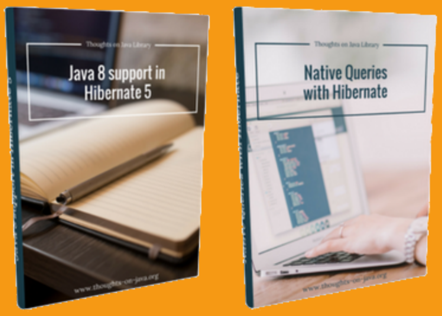
Name *

Email *

Website

Comment

POST COMMENT



LET'S CONNECT

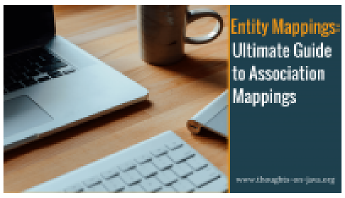
Thorben Janssen

SPEAKING AT

16th-17th May 2017 in Munich
Training:"Hibernate für Fortgeschrittene"

18th-19th May 2017 in Munich
Training:"Hibernate Performance Tuning"

FEATURED POSTS



Ultimate Guide – Association Mappings with JPA and Hibernate

Ultimate Guide to JPQL Queries with JPA and Hibernate

Hibernate Best Practices

Advanced Queries:
11 JPA and Hibernate query hints every developer should know
www.thoughts-on-java.org

11 JPA and Hibernate query hints every developer should know

Native Queries – How to call native SQL queries with JPA

RECENT POSTS

Ordering vs Sorting with Hibernate – What should you use?

Hibernate Tips: How to exclude unchanged columns from generated update statements

How to map an association as a java.util.Map

Thoughts on Java Report May 2017

Ultimate Guide – Association Mappings with JPA and Hibernate

Hibernate Tips: How to filter entities from a mapped association?

Mapping Definitions in JPA and Hibernate – Annotations, XML or your database with Hibernate

Hibernate Tips: How to map a bidirectional many-to-one association