Thoughts on Java Library

Native Queries with Hibernate

www.thoughts-on-java.org

Native queries with Hibernate

Thoughts on Java Library

Thorben Janssen

Contents

Foreword

Hi,

I'm Thorben, the author and founder of thoughts-on-java.org[1]. Thank you for downloading this ebook.

The Java Persistence Query Language (JPQL) is the most common way to query data from a database with JPA. But it supports only a small subset of the SQL standard, and it also provides no support for database specific features.

So what shall you do, if you need to use a database-specific query feature, or your DBA gives you a highly optimized query that can not be transformed into JPQL?

Just ignore it and do all the work in the Java code?

Of course not!

JPA has its own query language, but it also supports native SQL. You can create these queries in a very similar way as JPQL queries, and they can even return managed entities if you want. I will explain all the details in this ebook.

Take care,

Thorben

[1] http://www.thoughts-on-java.org

Create Native Queries

Similar to JPQL, you can create dynamic and named native queries. Dynamic native queries are instantiated at runtime, and you can dynamically adapt them. Named native queries are created at deploy time, and you can not change them afterward. That allows Hibernate to prepare and optimize the execution of named native queries.

Create dynamic native queries

It is quite easy to create a dynamic native query. The *EntityManager* interface provides a method called *createNativeQuery* for it. This method returns an implementation of the *Query* interface which is the same as if you call the *createQuery* method to create a JPQL query.

The following code snippet shows a simple example in which I used a native query to select the first and last names from the *author* table. I know, there is no need to do this with a native SQL query. I could use a standard JPQL query for this, but I want to focus on the JPA part and not bother you with some crazy SQL stuff.

The persistence provider does not parse the SQL statement, so you can use any SQL statement that is supported by your database. In one of my recent projects, for example, I used it to query PostgreSQL specific jsonb columns with Hibernate and mapped the query results to POJOs and entities.

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a");
List<Object[]> authors = q.getResultList();

for (Object[] a : authors) {
    System.out.println("Author " + a[0] + " " + a[1]);
}
```

As you can see, you can use the created Query in the same way as any JPQL query. I didn't provide any mapping information for the result and so the * EntityManager* returns a *List* of *Object[]* which need to be handled afterward. Instead of mapping the result yourself, you can also provide additional mapping information and let the *EntityManager* do the mapping for you. I get into more details about that in the result mapping chapter at the end of this ebook.

Parameter binding

Similar to JPQL queries, you can and should use parameter bindings for your query parameters instead of putting the values directly into the query *String*. This provides several advantages:

- you do not need to worry about SQL injection,
- Hibernate maps your query parameters to the correct types and
- Hibernate provider can do internal optimizations to provide better performance.

JPQL and native SQL queries use the same *Query* interface which provides a *setParameter* method for positional and named parameter bindings. But the use of named parameter bindings for native queries is not defined by the JPA specification, and you should use positional parameters to keep your implementation vendor independent.

Positional parameters are referenced as "?" in your native Query, and their numbering starts at 1.

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a WHE\
RE a.id = ?");
q.setParameter(1, 1);
Object[] author = (Object[]) q.getSingleResult();

System.out.println("Author "
        + author[0]
        + " "
        + author[1]);
```

Hibernate also supports named parameter bindings for native queries. When you use named parameter bindings, you define a name for each parameter and provide it to the *setParameter* method to bind a value to it. The name is case-sensitive and needs to be prefixed with a ":" symbol.

```
Query q = em.createNativeQuery("SELECT a.firstname, a.lastname FROM Author a WHE\
RE a.id = :id");
q.setParameter("id", 1);
Object[] author = (Object[]) q.getSingleResult();

System.out.println("Author "
        + author[0]
        + " "
        + author[1]);
```

As you have seen in the previous code snippets, your native Query returns an *Object[]* or a *List* of *Object[]*. I will show you how to change that in the *SQL Result Set Mapping* chapter of this ebook.

Create named native queries

You will not be surprised if I tell you that the definition and usage of a named native query is again very similar to a named JPQL query.

In the previous code snippets, I created a dynamic native query to select the names of all authors. I use the same query in the following code snippet and create a named query for it.

```
@NamedNativeQueries({
    @NamedNativeQuery(name = "selectAuthorNames", query = "SELECT a.firstname, a\
.lastname FROM Author a")
})
```

As you can see, the definition looks very similar to the definition of a named JPQL query. The named native query is used in the same way as a named JPQL query. You only need to provide the name of the named native query as a parameter to the *createNamedQuery* method of the *EntityManager*.

```
Query q = em.createNamedQuery("selectAuthorValue");
List<AuthorValue> authors = q.getResultList();

for (AuthorValue a : authors) {
    System.out.println("Author "
            + a.getFirstName()
            + " "
            + a.getLastName()
            + " wrote "
            + a.getNumBooks()
            + " books.");
}
```

Summary

JPQL provides an easy way to query data from the database but it supports only a small subset of the SQL standard, and it also does not support database specific features. But this is not a real issue. You can use all these features by creating native SQL queries via *EntityManager.createNativeQuery(String sql)*. Hibernate sends native queries directly to the database and you can use all SQL and proprietary features that are supported by your database.

Use Native Queries to Perform Bulk Updates

If you just want to change 1 or 2 entities, you can simply fetch them from the database and perform the update operation on them. But what about updating hundreds of entities?

You can, of course, use the same approach and load and update each of these entities. But that is often too slow because Hibernate performs one or more queries to load the entity and an additional one to update it. This quickly results in a few hundred SQL statements which are obviously slower than just 1 statement which lets the database do the work.

As I explain in great detail in the Hibernate Performance Tuning Online Training[2], the number of performed SQL statements is crucial for the performance of your application. So you better have an eye on your statistics and keep the number of statements as low as possible. You can do that with JPQL or native SQL queries which define the update in one statement.

Using a native UPDATE statement is quite easy as I will show you in the next section. But it also creates issues with the always activated 1st level cache and doesn't trigger any entity lifecycle events. I'll show you how to handle these problems at the end of this chapter.

Native UPDATE statements

As I explained in the previous chapter, creating a native query is pretty simple. You just have to call the *createNativeQuery* method on the *EntityManager* and provide a native SQL statement to it.

```
em.createNativeQuery("UPDATE person p SET firstname = firstname || '-changed'").\
executeUpdate();
```

In this example, I update the *firstName* of all 200 persons in my test database with one query. That takes about 30ms on my local test setup.

The typical JPA approach would require 200 *SELECT* statements to fetch each Person entity from the database and additional 200 *UPDATE* statements to update each of them. The execution of these 400 statements and all the Hibernate-internal processing takes about 370ms on my local test setup.

I just used *System.currentTimeMillis()* to measure the execution time on my laptop which is also running a lot of other applications. The setup is far from optimal and not suited for a real performance test. So don't rely on the measured milliseconds. But it becomes evident which approach is the faster one, and that's what it's all about.

But it also creates some problems.

[2] http://www.thoughts-on-java.org/course-hibernate-performance-tuning

Problem 1: Outdated 1st level cache

Hibernate puts all entities you use within a session into the first level cache. This is pretty useful for write-behind optimizations and to avoid duplicate selects of the same entity. But it also creates an issue, if you use a native query to update a bunch of database records.

Hibernate doesn't know which records the native query updates and can't update or remove the corresponding entities from the first level cache. That means that Hibernate uses an outdated version of the entity if you fetched it from the database before you executed the native SQL UPDATE statement. You can see an example of it in the following code snippet. Both log statements print out the old *firstName*.

```
PersonEntity p = em.find(PersonEntity.class, 1L);

em.createNativeQuery("UPDATE person p SET firstname = firstname || '-changed'").\
executeUpdate();
log.info("FirstName: "+p.getFirstName());

p = em.find(PersonEntity.class, 1L);
log.info("FirstName: "+p.getFirstName());
```

There are two options to avoid this issue:

The most obvious one is to not fetch any entities from the database which will be affected by the UPDATE statement. But we both know that this is not that easy in a complex, modular application.

If you can't avoid fetching some of the affected entities, you need to update the 1st level cache yourself. The only way to do that is to detach them from the active persistence context and let Hibernate fetch them again as soon as you need them. But be careful, Hibernate doesn't perform any dirty check before detaching the entity. So you also have to make sure that all updates are written to the database before you detach any entity.

```
PersonEntity p = em.find(PersonEntity.class, 1L);

log.info("Detach PersonEntity");
em.flush();
em.detach(p);

em.createNativeQuery("UPDATE person p SET firstname = firstname || '-changed'").\
executeUpdate();

p = em.find(PersonEntity.class, 1L);
```

As you can see, I call the *flush()* and *detach()* method on the *EntityManager* before I perform the native query. The call of the *flush()* method tells Hibernate to write the changed entities from the 1st level cache to the database. That makes sure that you don't lose any update. You can then detach the entity from the current persistence context and due to this remove it from the 1st level cache. Afterward, you can modify the corresponding database record with a native query.

Problem 2: Not part of the entity life cycle

In most applications, this is not a huge problem. But I want to mention it anyways.

The native UPDATE statement is executed in the database and doesn't use any entities. That provides performance benefits, but it also avoids the execution of any entity lifecycle methods or entity listeners.

If you use a framework like Hibernate Envers or implement any code yourself that relies on lifecycle events, you have to either avoid native UPDATE statements or implement the operations of your listeners within your use case.

Summary

With the standard JPA approach, you fetch an entity from the database and call some setter methods to update it. This feels very natural to Java developers, but the number of required SQL statements can create performance issues for huge sets of entities. It's often a lot faster to update all entities with one native or JPQL UPDATE statement.

But you then have to take care of your 1st level cache. Hibernate doesn't know which records were updated in the database and didn't refresh the corresponding entities. You either have to make sure that you haven't fetched any entities which are affected by the update, or you have to detach them from the Hibernate session before you execute the update.

You also have to check if you use any entity lifecycle methods or entity listeners. The native UPDATE statement doesn't use any entities and therefore doesn't trigger any lifecycle event. If you rely on lifecycle events, you either have to avoid native UPDATE statements, or you have to handle the missing lifecycle events within your use case.

SQL Result Set Mappings

One downside of native queries is that they return a *List* of *Object[]* instead of the mapped entities and value objects you normally use. Each of the *Object[]* represents one record returned by the database.

You then need to iterate through the array, cast each Object to its particular type and map them to our domain model. This creates lots of repetitive code and type casts as you can see in the following example.

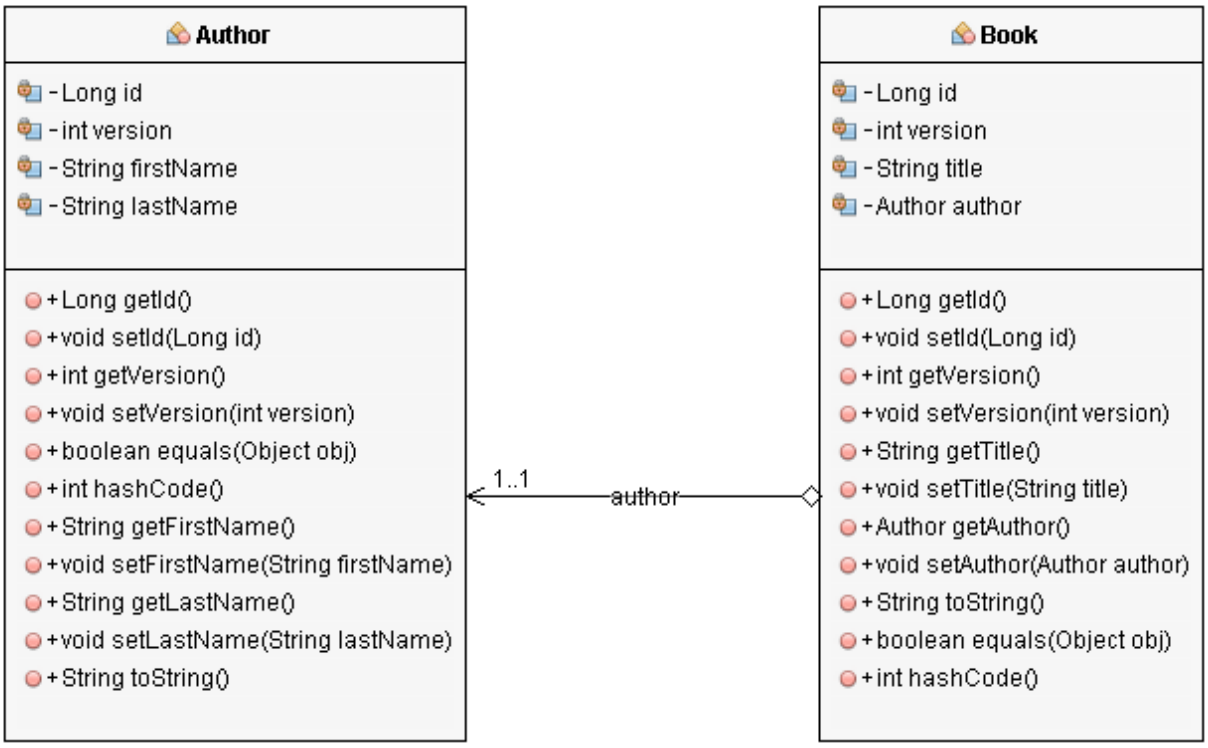
```
List<Object[]> results = this.em.createNativeQuery("SELECT a.id, a.firstName, a.\
lastName, a.version FROM Author a").getResultList();

results.stream().forEach((record) -> {
        Long id = ((BigInteger) record[0]).longValue();
        String firstName = (String) record[1];
        String lastName = (String) record[2];
        Integer version = (Integer) record[3];
});
```

It would be more comfortable if we could tell the *EntityManager* to map the result of the query into entities or value objects as it is the case for JPQL statements. The good news is, JPA provides this functionality. It is called SQL result set mapping, and I will explain it in this chapter.

The example

Before we dive into the details of SQL result set mappings, let's have a look at the entity model that we will use during this chapter.

It consists of an *Author* entity with an id, a version, a first name and a last name. For the more complex mapping examples, I also need the *Book* entity which has an id, a version, a title and a reference to the *Author*. To keep it simple, each book is written by only one author.



Class diagram entites

Basic mappings

Let's begin with some basic mappings that tell Hibernate to map each record of the result set to an entity. The easiest way to do that is to use the default mapping, but you can also provide your own mapping definition.

How to use the default mapping

You just need to provide the entity class as a parameter to the *createNativeQuery(String sqlString, Class resultClass)* method of the *EntityManager* to use the default mapping. The following snippet shows how this is done with a very simple query. In a real project, you would use this with a stored procedure or a very complex SQL query.

```
List<Author> results = this.em.createNativeQuery("SELECT a.id, a.firstName, a.la\
stName, a.version FROM Author a", Author.class).getResultList();
```

The query needs to return all attributes of the entity, and the JPA implementation (e.g. Hibernate) will try to map the returned columns to the entity attributes based on their name and type. The

EntityManager will then return a list of fully initialized *Author* entities that are managed by the current persistence context. So the result is the same as if you had used a JPQL query, but you are not limited to the small feature set of JPQL.

How to define a custom mapping

While this automatic mapping is useful and easy to use, it is often not sufficient. If you perform a more complex query or call a stored procedure, the names of the returned columns might not match the entity definition. In these cases, you need to define a custom result mapping which specifies the mapping for all entity attributes, even if the default mapping could be applied to some of them.

Let's have a look at the example and rename the *id* column returned by the query to *authorId*:

```
SELECT a.id as authorId, a.firstName, a.lastName, a.version FROM Author a
```

The default mapping to the *Author* entity will not work with this query result because the names of the selected columns and the entity attributes do not match. You need to define a custom mapping for it. You can do this with annotations or in a mapping file (e.g. *orm.xml*).

The following code snippet shows how to define the result mapping with the *@SqlResultSetMapping* annotation. The mapping consists of a name and an *@EntityResult* definition. The name of the mapping, *AuthorMapping* in this example, will later be used to tell the *EntityManager* which mapping definition it shall apply. The *@EntityResult* annotation defines the entity class to which the result has to be mapped and an array of *@FieldResult* annotations which define the mapping between the column names and the entity attributes. Each *@FieldResult* gets the name of the attribute and the column name as a parameter.

```
@SqlResultSetMapping(
        name = "AuthorMapping",
        entities = @EntityResult(
                entityClass = Author.class,
                fields = {
                    @FieldResult(name = "id", column = "authorId"),
                    @FieldResult(name = "firstName", column = "firstName"),
                    @FieldResult(name = "lastName", column = "lastName"),
                    @FieldResult(name = "version", column = "version")
                }
            )
)
```

As JPA 2.1 is based on Java 7, there is no support for repeatable annotations. Therefore you need to place your *@SqlResultSetMapping* annotations within a *@SqlResultMappings* annotation if you want to define more than one mapping at an entity.

If you don't like to add huge blocks of annotations to your entities, you can define the mapping in an XML mapping file. Hibernate's default mapping file is called *orm.xml* and will be used automatically, if it is added to the *META-INF* directory of the jar file. As you can see below, the mapping is very similar to the annotation based mapping I showed you before. I use the name *AuthorMappingXml* to avoid name clashes with the annotation based mapping. In a real project, you don't need to worry about this, because you would use only one of the two described mappings.

```
<sql-result-set-mapping name="AuthorMappingXml">
    <entity-result entity-class="org.thoughts.on.java.jpa.model.Author">
        <field-result name="id" column="authorId"/>
        <field-result name="firstName" column="firstName"/>
        <field-result name="lastName" column="lastName"/>
        <field-result name="version" column="version"/>
    </entity-result>
</sql-result-set-mapping>
```

OK, so now you have defined your own mapping between the query result and the *Author* entity. You can now provide the name of the mapping instead of the entity class as a parameter to the *createNativeQuery(String sqlString, String resultSetMapping)* method. In the code snippet below, I used the annotation defined mapping.

```
List<Author> results = this.em.createNativeQuery("SELECT a.id as authorId, a.fir\
stName, a.lastName, a.version FROM Author a", "AuthorMapping").getResultList();
```

Complex Mappings

The mappings described so far were quite simple. In real applications, you often need more complex mappings that can handle multiple entities and additional columns or that can map to value objects instead of entities.

How to map multiple entities

Let's begin with a mapping that maps a query result to an *Author* entity and initializes the relationship to a list of *Book* entities. You can define a query that fetches all columns of the *Author* entity and all columns of the all related *Book* entities.

```
SELECT b.id, b.title, b.author_id, b.version, a.id as authorId, a.firstName, a.l\
astName, a.version as authorVersion FROM Book b JOIN Author a ON b.author_id = a\
.id
```

As the *Author* and the *Book* table both have an *id* and a *version* column, we need to rename them in the SQL statement. I decided to rename the *id* and *version* column of the *Author* to *authorId* and *authorVersion*. The columns of the *Book* stay unchanged.

OK, so how do you define an SQL result set mapping that transforms the returned *List* of *Object[]* to a *List* of fully initialized *Book* and *Author* entities?

The mapping definition looks similar to the custom mapping that we discussed earlier. The *@SqlResultMapping* annotation defines the name of the mapping that you will need to reference it later on. The main difference here is, that it uses two *@EntityResult* annotations, one for the *Book* and one for the *Author* entity. Each *@EntityResult* looks again similar to the previous mapping and defines the entity class and a list of *@FieldResult* mappings.

```
@SqlResultSetMapping(
        name = "BookAuthorMapping",
        entities = {
            @EntityResult(
                    entityClass = Book.class,
                    fields = {
                        @FieldResult(name = "id", column = "id"),
                        @FieldResult(name = "title", column = "title"),
                        @FieldResult(name = "author", column = "author_id"),
                        @FieldResult(name = "version", column = "version")}),
            @EntityResult(
                    entityClass = Author.class,
                    fields = {
                        @FieldResult(name = "id", column = "authorId"),
                        @FieldResult(name = "firstName", column = "firstName"),
                        @FieldResult(name = "lastName", column = "lastName"),
                        @FieldResult(name = "version", column = "authorVersion")
                    }
            )
        }
)
```

You can, of course, also define the mapping in an XML file. As described before, the default mapping file is called *orm.xml* and will be automatically used, if it is added to the *META-INF* directory of the jar file. The mapping definition itself looks similar to the already described annotation based mapping definition.

```
<sql-result-set-mapping name="BookAuthorMappingXml">
    <entity-result entity-class="org.thoughts.on.java.jpa.model.Author">
        <field-result name="id" column="authorId"/>
        <field-result name="firstName" column="firstName"/>
        <field-result name="lastName" column="lastName"/>
        <field-result name="version" column="authorVersion"/>
    </entity-result>
    <entity-result entity-class="org.thoughts.on.java.jpa.model.Book">
        <field-result name="id" column="id"/>
        <field-result name="title" column="title"/>
        <field-result name="author" column="author_id"/>
        <field-result name="version" column="version"/>
    </entity-result>
</sql-result-set-mapping>
```

Now you have a custom result set mapping definition, which defines the mapping between your query result and the *Book* and *Author* entity. If you provide this to the *createNativeQuery(String sqlString, String resultSetMapping)* method of the *EntityManager*, you get a *List<Object[]>*.

OK, that might not look like what we wanted to achieve in the first place. We wanted to get rid of these *Object[]*. If you have a more detailed look at the *Objects* in the array, you see that these are no longer the different columns of the query but the *Book* and *Author* entities. And as the *EntityManager* knows that these two entities are related to each other, the relation to the Book entity is already initialized.

```
List<Object[]> results = this.em.createNativeQuery("SELECT b.id, b.title, b.auth\
or_id, b.version, a.id as authorId, a.firstName, a.lastName, a.version as author\
Version FROM Book b JOIN Author a ON b.author_id = a.id", "BookAuthorMapping").g\
etResultList();

results.stream().forEach((record) -> {
    Book book = (Book)record[0];
    Author author = (Author)record[1];
    // do something useful
});
```

How to map additional columns

Another very handy feature is the mapping of additional columns in the query result. If you want to select all *Authors* and their number of *Books*, you can get this information with the following query.

```
SELECT a.id, a.firstName, a.lastName, a.version, count(b.id) as bookCount FROM B\
ook b JOIN Author a ON b.author_id = a.id GROUP BY a.id, a.firstName, a.lastName\
, a.version
```

So how do you map this query result to an *Author* entity and an additional *Long* value?

That is quite simple; you just need to combine a mapping for the *Author* entity with a *@ColumnResult* definition. The mapping of the *Author* entity has to define the mapping of all columns, even if you do not change anything as in the example below. The *@ColumnResult* specifies the name of the column that shall be mapped and can optionally provide the Java type to which it shall be converted. I used it to convert the *BigInteger*, that the query returns by default, to a *Long*.

```
@SqlResultSetMapping(
        name = "AuthorBookCountMapping",
        entities = @EntityResult(
                entityClass = Author.class,
                fields = {
                    @FieldResult(name = "id", column = "id"),
                    @FieldResult(name = "firstName", column = "firstName"),
                    @FieldResult(name = "lastName", column = "lastName"),
                    @FieldResult(name = "version", column = "version")
                }
        ),
        columns = @ColumnResult(name = "bookCount", type = Long.class)
)
```

As before, this mapping can also be defined with a similar looking XML configuration.

```
<sql-result-set-mapping name="AuthorBookCountMappingXml">
    <entity-result entity-class="org.thoughts.on.java.jpa.model.Author">
        <field-result name="id" column="id"/>
        <field-result name="firstName" column="firstName"/>
        <field-result name="lastName" column="lastName"/>
        <field-result name="version" column="version"/>
    </entity-result>
    <column-result name="bookCount" class="java.lang.Long" />
</sql-result-set-mapping>
```

If you use this mapping in the *createNativeQuery(String sqlString, String resultSetMapping)* of the *EntityManager*, you get a *List* that contains the initialized *Author* entity and the number of her/his *Books* as a *Long*.

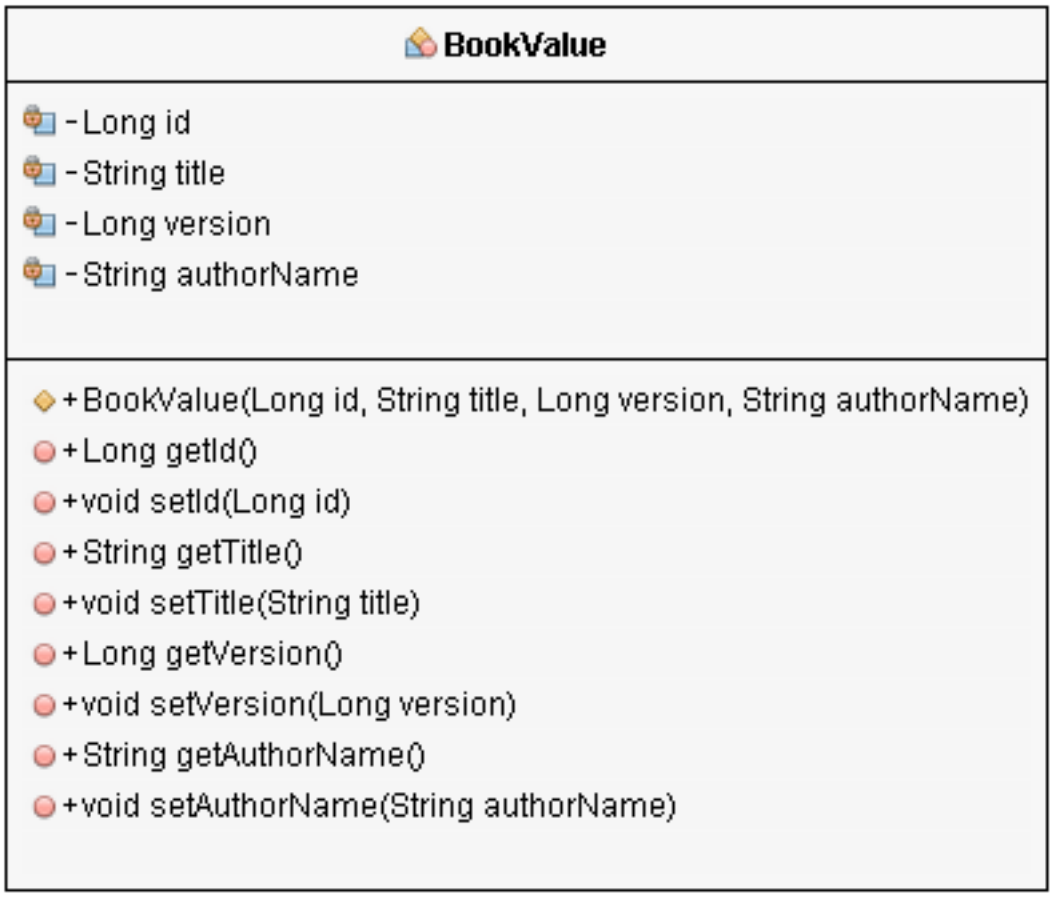
```
List<Object[]> results = this.em.createNativeQuery("SELECT a.id, a.firstName, a.\
lastName, a.version, count(b.id) as bookCount FROM Book b JOIN Author a ON b.aut\
hor_id = a.id GROUP BY a.id, a.firstName, a.lastName, a.version", "AuthorBookCou\
ntMapping").getResultList();

results.stream().forEach((record) -> {
    Author author = (Author)record[0];
    Long bookCount = (Long)record[1];
    System.out.println("Author: ID ["+author.getId()+"] firstName ["+author.getF\
irstName()+"] lastName ["+author.getLastName()+"] number of books ["+bookCount+"\
]");
});
```

This kind of mapping is pretty useful if your query result has no exact mapping to your entity model. Reasons for this can be additional attributes calculated by the database, as you have seen in the example above, or queries that select only some specific columns from related tables.

Constructor Result Mappings

Selecting entities and returning a tree of objects to the caller is not always the best approach. The caller often needs only a subset of the provided information, and a particular value object would be much more efficient. For these situations, JPQL supports constructor expressions that can be specified in the select part of the JPQL query and define the constructor call for each selected record. Since JPA 2.1, you can use a constructor result mapping to do the same with the result of a native query.

I will use the same entities as in the previous examples. But as I want to map the query results to a POJO, I will use the class *BookValue* with an id, a version, a title and the name of the author.



Class Diagram BookValue

How to map to a value object

The query that returns the required columns to initialize a *BookValue* object is quite simple. It joins the *book* and *author* table and selects the id, title and version columns of the *book* table and concatenates the firtName and lastName columns of the *author* table.

```
SELECT b.id, b.title, b.version, a.firstName || a.lastName as authorName FROM Bo\
ok b JOIN Author a ON b.author_id = a.id
```

You now just have to define a mapping that uses the query result to call the constructor of the *BookValue*. As in the previous examples, you can do that with a *@SqlResultSetMapping* annotation. The following code snippet shows an example of such a mapping.

```
@SqlResultSetMapping(
        name = "BookValueMapping",
        classes = @ConstructorResult(
                targetClass = BookValue.class,
                columns = {
                    @ColumnResult(name = "id", type = Long.class),
                    @ColumnResult(name = "title"),
                    @ColumnResult(name = "version", type = Long.class),
                    @ColumnResult(name = "authorName")
                }
        )
)
```

The name of the mapping, *BookValueMapping* in this example, will later be used to tell the *EntityManager* which mapping to use. The *@ConstructorResult* annotation defines the constructor call for a given target class. In this example, it's the *BookValue* class. The array of *@ColumnResult* annotations specifies the columns of the query result that will be used as constructor parameters with their type and order. The type attribute is optional. You only need to provide it, if the type of the column is different to the type of the constructor parameter. In this case, the default types of the id and version columns are *BigInteger* and need to be converted to *Long*.

Similar to the mapping of multiple entities, the *classes* attribute of the *@SqlResultSetMapping* accepts an array of *@ConstructorResult* annotations. If the mapping maps to multiple value objects or entities, each column can be used multiple times.

And like all the mapping definitions before, you can also define the constructor result mapping in a mapping XML file. The easiest way to do this is to use the default mapping file called *orm.xml*.

```
<sql-result-set-mapping name="BookValueMappingXml">
    <constructor-result target-class="org.thoughts.on.java.jpa.value.BookValue">
        <column name="id" class="java.lang.Long"/>
        <column name="title"/>
        <column name="version" class="java.lang.Long"/>
        <column name="authorName"/>
    </constructor-result>
</sql-result-set-mapping>
```

The usage of the constructor mapping is identical to the other SQL result set mappings. You just need to provide its name to the *createNativeQuery(String sqlString, String resultSetMapping)* method of the *EntityManager*.

```
List<BookValue> results = this.em.createNativeQuery("SELECT b.id, b.title, b.ver\
sion, a.firstName || a.lastName as authorName FROM Book b JOIN Author a ON b.aut\
hor_id = a.id", "BookValueMapping").getResultList();
```

Hibernate Specific Mappings

All previous examples used features defined by the JPA specification. Hibernate implements all of them and also provides its own API for mapping query results. While this creates a vendor lock and makes migration to another framework difficult, it also offers some interesting features. As always, you need to decide which trade-off you want to make.

How to use Hibernate Specific Features

The previous examples used JPA standard features and therefore the *EntityManager* to perform native queries. If you want to use Hibernate specific features, and you need to use the Hibernate *Session* instead. You can get it in a Java EE environment via the *EntityManager.unwarp(Class<T> cls)* method as shown in the following code snippet:

```
@PersistenceContext
private EntityManager em;

...

public void queryWithAuthorBookCountHibernateMapping() {
  Session session = this.em.unwrap(Session.class);
  ...
}
```

Aliases make the mapping easier

Hibernate provides its own API that supports a similar set of features as the JPA standard. But using the Hibernate API is sometimes more convenient as the result mappings you have seen in the previous examples.

One example for this is the following code snippet in which all *Books* and *Authors* are selected from the database and mapped to the corresponding entities. In a real world project, you would probably not use a native query for such a simple select. But it is good enough to explain the result mapping.

```
List<Object[]> results = session.createSQLQuery("SELECT {b.*}, {a.*} FROM Book b\
 JOIN Author a ON b.author_id = a.id").addEntity("b", Book.class).addEntity("a",\
 Author.class).list();
results.stream().forEach((record) -> {
    Book book = (Book) record[0];
    Author author = (Author) record[1];
    System.out.println("Author: ID [" + author.getId() + "] firstName [" + autho\
r.getFirstName() + "] lastName [" + author.getLastName() + "]");
    System.out.println("Book: ID [" + book.getId() + "] title[" + book.getTitle(\
) + "]");
});
```

The syntax of the query might look strange at the beginning, but it provides a very easy way to select all attributes of an entity. Instead of selecting all attributes in the select part of the query and map them one by one to the entity attributes, as you have seen in the previous example, you now use {a.*} and {b.*} to select them. The mapping from the aliases a and b to the entity classes is done by calling *addEntity(String tableAlias, Class entityType)*.

The following snippet shows a similar result mapping. This time, it selects an Author entity and the number of her/his books as a scalar value. You already know this query from a previous example that used an *@SqlResultSetMapping* annotation of the JPA standard to map the result.

```
List<Object[]> results = session.createSQLQuery("SELECT {a.*}, count(b.id) as bo\
okCount FROM Book b JOIN Author a ON b.author_id = a.id GROUP BY a.id, a.firstNa\
me, a.lastName, a.version").addEntity(Author.class).addScalar("bookCount", Stand\
ardBasicTypes.LONG).list();
results.stream().forEach((record) -> {
    Author author = (Author) record[0];
    Long bookCount = (Long) record[1];
    System.out.println("Author: ID [" + author.getId() + "] firstName [" + autho\
r.getFirstName() + "] lastName [" + author.getLastName() + "] number of books ["\
 + bookCount + "]");
});
```

You could create similar mappings with JPA. But from my point of view, the Hibernate API is a little bit easier to use, if the result mapping is specific to one query. But if there are no other reasons to create a dependency to Hibernate instead of JPA, I would still use JPA. Additionally, the result mapping annotations (or XML configuration) of the JPA standard can be used to map the results of multiple queries.

ResultTransformer for more flexibility

Another and more powerful way to transform the query result is *ResultTransformer*. It provides the option to define the result mapping in Java code.

OK, you might say that this is what you tried to avoid in the beginning, and you are right about that. But as you can see in the JavaDoc, Hibernate provides quite a list of different implementations of this interface. So in most cases, there is no need to implement the mapping yourself. Otherwise, the *ResultTransformer* provides only minimal benefits compared to a programmatic mapping using the *Stream* API.

One of the provided *ResultTransformer* is the *AliasToBeanResultTransformer*, which maps the query result to a Java Bean. But instead of using a constructor call, as I showed earlier, the transformer uses the setter methods or fields to populate the object. This can be beneficial, if the class has lots of fields and you would need to create a constructor with a parameter for each of them or if you would need multiple constructors because multiple query results need to be mapped to the same class. The following code snippet shows an example of the AliasToBeanResultTransformer:

```
List<BookValue> results = session.createSQLQuery("SELECT b.id, b.title, b.versio\
n, a.firstName || ' ' || a.lastName as authorName FROM Book b JOIN Author a ON b\
.author_id = a.id")
    .addScalar("id", StandardBasicTypes.LONG).addScalar("title").addScalar("vers\
ion", StandardBasicTypes.LONG).addScalar("authorName")
    .setResultTransformer(new AliasToBeanResultTransformer(BookValue.class)).lis\
t();

results.stream().forEach((book) -> {
    System.out.println("Book: ID [" + book.getId() + "] title [" + book.getTitle\
() + "] authorName [" + book.getAuthorName() + "]");
});
```

The *AliasToBeanResultTransformer* uses the default constructor of the *BookValue* to instantiate an object and searches the getter methods based on the alias and type of the returned column. You, therefore, need to use the *addScalar()* method to rename the columns and change the types of the *id* and *version* column.

Summary

The JPA standard provides different options to define result mappings. If you select all columns mapped by an entity and provide the right column aliases, you just need to provide the class of the entity to use the default mapping. Hibernate will then use the mapping definition to map the columns of the result set to the entity attributes.

You can also provide your own mapping definition with a *@SqlResultSetMapping* annotation or an XML configuration. This allows you to map a query result to multiple entities or POJOs or to map additional scalar values.

Hibernate implements the JPA result set mappings and provides its own API. It supports aliases to define the mapping between the query result and the Java entities or value objects. Besides being

easier to use, this also provides the advantage, that all information is in the same place. There is no need to search for the mapping definition in some annotations or XML files. On the other hand, it requires more work to define the mapping, and it is not as easy to reuse as the JPA standard approach.

The ResultTransformer, on the other hand, can provide some real benefits compared to the standard mapping. These can be used to do more complex mappings, and Hibernate already provides a list of ResultTransformer implementations. If none of the existing transformation implementation provides the required functionality, there is also the option to implement your own one. But in this case, I would prefer to use the Streams API to map the query results inside my business code.