Optimal Query Approach and Projection

Using JPA and Hibernate, you can choose between various ways to query your data, and each of them supports one or more kinds of projections. That provides you with lots of options to implement your persistence layer. But which one fits your use case? And which one should you avoid if you want to optimize your persistence layer for performance?

Entity Projections

For most teams, entity projections are the most commonly used ones. But that's often not the best approach. Entities might be easy to use and are the best fit for all write operations. But as I showed in a previous article, selecting and managing an entity creates an overhead that slows down your application.

So, if you need to optimize your persistence layer for performance, you should only use entity projections if you need them to implement write operations.

Select Entities by ID

You could, of course, implement your own query to fetch one or more entities by their primary key. But you don't have to. There are more comfortable options available.

If you've ever used JPA or Hibernate, you know the *find* method of the *EntityManager* interface. It provides an easy to use way to load one entity by its primary key.

```
Author a = em.find(Author.class, id);
```

On top of this, Hibernate provides an API that enables you to load more than one entity by its primary key. You just need to provide a *List* of ids and Hibernate loads all of them in one query.

```
MultiIdentifierLoadAccess<Book> multi = session.byMultipleIds(Book.class);
List<Book> books = multi.multiLoad(1L, 2L, 3L);
```

Not too Complex Queries Selecting Entities

If you can statically define a not too complex query and need to filter by non-primary key attributes in your WHERE condition, you should use a named JPQL query. JPQL is a query language similar to SQL. The main 2 differences are that you can define your query based on your domain model and that JPQL is not as feature-rich as SQL.

You can define named JPQL queries by annotating an entity class with one or more *@NamedQuery*. Since Hibernate 5 and JPA 2.2, you no longer need to wrap multiple *@NamedQuery*annotations in a *@NamedQueries* annotation.

The syntax of a JPQL query is pretty similar to SQL. The query in the following code snippet selects all *Author* entities with a given *firstname*.

```
@Entity
@NamedQuery(name = "Author.findAuthorByFirstname", query = "SELECT a
FROM Author a WHERE a.firstname = :firstname")
public class Author { ... }
```

You can instantiate this query by calling the *createNamedQuery* method of your *EntityManager*with the name of the query. This method returns a *Query* or *TypedQuery* interface. You can then use this interface to set bind parameter values and to execute the query.

```
TypedQuery<Author> q = em.createNamedQuery("Author.findByFirstname",
Author.class);
q.setParameter("firstname", firstname);
List<Author> authors = q.getResultList();
```

Dynamic Queries Selecting Entities

JPA's Criteria API enables you to create your query dynamically at runtime.

Here you can see an example that selects all *Author* entities with a given *firstname*. This is the same query as I showed you in the previous JPQL example. As you can see, the code block that uses the Criteria API is longer and harder reader.

To create a CriteriaQuery, you first need to get a *CriteriaBuilder* from the *EntityManager* and create a query that returns the entity class. You can then use this query to define the FROM and WHERE clause. After you've created the *CriteriaQuery*, you can use it to create a *TypedQuery*, set the bind parameter values, and execute it.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Author> cq = cb.createQuery(Author.class);
Root<Author> root = cq.from(Author.class);

ParameterExpression<String> paramFirstName = cb.parameter(String.class);
cq.where(cb.equal(root.get(Author_.firstName), paramFirstName));

TypedQuery<Author> query = em.createQuery(cq);
query.setParameter(paramFirstName, "Thorben");
List<Author> authors = query.getResultList();
```

The Criteria API supports the same features as JPQL. Explaining all of them in details would take too long for this article. If you want to learn more about it, please join my [Advanced Hibernate Online Training](#).

Complex and Database-Specific Queries Selecting Entities

If your query gets really complex or if you want to use database-specific features, you need to use a native SQL query. Hibernate takes the native query statement and sends it to the database without parsing it.

If your native query returns all columns mapped by your entity and if their names are identical to the ones used in the entity mapping, you just need to provide your entity class as the 2nd parameter to the *createNativeQuery* method. Hibernate will then apply the standard entity mapping to your query result.

```
Book b = (Book) em.createNativeQuery("SELECT * FROM book b WHERE id =
1", Book.class).getSingleResult();
```

You can customize this mapping using a *@SqlResultSetMapping* annotation. I explained this annotation and how you can use it to define all kinds of mappings in a series of articles:

- Basic SQL ResultSet Mappings
- Complex SQL ResultSet Mappings
- Mapping DTO Projections
- Hibernate-specific Mappings

Scalar Value Projections

Scalar value projections are my least favorite ones. In almost all situations, I prefer DTO projections, which I will show you in the following section. Scalar value projections can be a good option if you need to read and immediately process 1-5 database columns for which you don't have a matching DTO projection.

The main downside of scalar value projections is that they are very uncomfortable to use. You can use this projection with a JPQL, Criteria, or native SQL query. In all 3 cases, your query returns an *Object[]*. When you use this array, you need to remember the position of each column and cast the array element to the correct type.

Here is an example of a JPQL query that uses a scalar value projection.

```
TypedQuery<Object[]> q = em.createQuery("SELECT b.title, b.publisher.name
FROM Book b WHERE b.id = :id", Object[].class);
q.setParameter("id", 1L);
Object[] result = q.getSingleResult();
```

Please take a look at the following articles, if you want to use this projection with a Criteria or native SQL query:

- Select scalar values in a Criteria Query
- Complex SQL ResultSet Mappings

DTO Projections

From a performance point of view, DTO projections are almost as good as scalar value projections. They provide the best performance for read operations. But the strong typing and the descriptive attribute names make this projection so much easier to use.

You can use DTO projections in JPQL, Criteria, and native SQL queries.

Not too Complex Queries Selecting DTOs

Named JPQL queries are a good fit for all queries that are not too complex and that you want to define based on your domain model.

The definition of a JPQL query that uses a DTO projection is pretty simple. You define a constructor call in your SELECT clause using the keyword *new* followed by the fully qualified name of your DTO class and a list of parameters.

```
TypedQuery<BookValue> q = em.createQuery("SELECT new
org.thoughts.on.java.model.BookValue(b.id, b.title, b.publisher.name) FROM
Book b WHERE b.id = :id", BookValue.class);
q.setParameter("id", 1L);
BookValue b = q.getSingleResult();
```

Dynamic Queries Selecting DTOs

JPA's Criteria API enables you to create your query dynamically at runtime. As I explained early, this is a little bit slower than using a JPQL query, and the code is hard to read. So, better use a JPQL query if you can define your query statically.

You define and execute the CriteriaQuery in almost the same way as the CriteriaQuery I showed you early. The only difference is that you now need to call the *construct* method on the *CriteriaBuilder* to define the constructor call.

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<AuthorValue> q = cb.createQuery(AuthorValue.class);
Root<Author> root = q.from(Author.class);
q.select(cb.construct(AuthorValue.class, root.get(Author_.firstName),
root.get(Author_.lastName)));

TypedQuery<AuthorValue> query = em.createQuery(q);
List<AuthorValue> authors = query.getResultList();
```

Complex and Database-Specific Queries Selecting DTOs

If your query is too complex for JPQL, you can use a native SQL query and an *@SqlResultSetMapping* using a *@ConstructorResult* annotation.

Hibernate then executes the native query and iterates through the result set. For each record, it calls the constructor defined by the *@ConstructorResult* annotation.

Here you can see the definition of a constructor call of the *BookValue* class. Hibernate will provide the value of the *title* column as the 1st and the value of the *date* column as the 2nd parameter.

```
@Entity
@SqlResultSetMapping(name = "BookValueMapping",
        classes = @ConstructorResult(
                targetClass = BookValue.class,
                columns = { @ColumnResult(name = "title"),
                        @ColumnResult(name = "date")}
                )
)
public class Book { ... }
```

To use this *@SqlResultSetMapping* with your query, you need to provide its name as the 2nd parameter to the *createNativeQuery* method.

```
BookValue b = (BookValue) em.createNativeQuery("SELECT b.publishingDate as
date, b.title, b.id FROM book b WHERE b.id = 1",
"BookValueMapping").getSingleResult();
```

Conclusion

When using JPA and Hibernate, you can choose between various ways to read the required information.

Choose the Best Kind of Query for the Use Case

You can use JPQL queries if they are static and not too complex. The Criteria API enables you to define your query dynamically using a

Java API. And if you need to use the full feature set of your database, you need to use a native SQL query.

Choose the Optimal Projection

You should use entity projections only if you need to implement write operations. JPA and Hibernate provide APIs to load one or more entities by their primary key. You can also use entity projections with JPQL, Criteria, and native queries.

Scalar value projections are not very comfortable to use, and you should better use a DTO projection.

DTO projections are the best option for read-only operations. They are strongly typed, easy to use, and provide the best performance. You can use them with JPQL, Criteria, and native queries.