Entities or DTOs – When should you use which projection?

By Thorben Janssen — 30 Comments

JPA and Hibernate allow you to use DTOs and entities as projections in your JPQL and Criteria queries. When I talk about Hibernate performance in my online training or at a workshop, I get often asked, if it matters which projection you use.

The answer is: YES! Choosing the right projection for your use case can have a huge performance impact.

And I'm not talking about selecting only the data that you need. It should be obvious that selecting unnecessary information will not provide you any performance benefits.

The Main Difference Between DTOs And Entities

There is another, often ignored difference between entities and DTOs. Your persistence context manages the entities.

That is a great thing when you want to update an entity. You just need to call a setter method with the new value. Hibernate will take care of the required SQL statements and write the changes to the database.

That's comfortable to use, but you don't get it for free. Hibernate has to perform dirty checks on all managed entities to find out if it needs to store any changes in the database. That takes time and is completely unnecessary when you just want to send a few information to the client.

You also need to keep in mind that Hibernate and any other JPA implementation, stores all managed entities in the 1st level cache. That seems to be a great thing. It prevents the execution of duplicate queries and is required for Hibernate's write behind optimization. But managing the 1st level cache takes time and can even become a problem if you select hundreds or thousands of entities.

So, using entities creates an overhead, which you can avoid when you use DTOs. But does that mean that you shouldn't use entities?

No, it doesn't.

The Hibernate Universe - Beyond known CRUD-Galaxies

Looking for an on-site training?

FEATURED POST

Getting Started With Hibernate

12 Java YouTube Channels You Should Follow In 2018

Join the Thoughts on Java Library

Get free access to:

- 2 Ebooks about JPA and Hibernate
- More than 50 Cheat Sheets
- More than 60 printable Hibernate Tips
- A 3-Part Video Course about Finding and Fixing N+1 Select Issues

Signup For Free:

Email

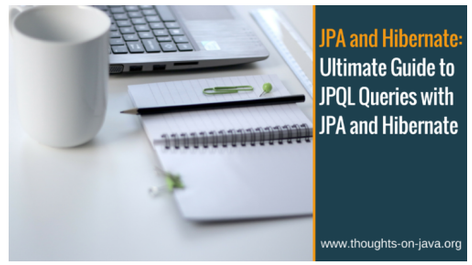
Password

Password Confirmation

JOIN NOW!

Already a member? Login here.

Ultimate Guide – Association Mappings with JPA and Hibernate



Ultimate Guide to JPQL Queries with JPA and Hibernate

Hibernate Best Practices

RECENT POSTS

Entity projections are great for all write operations. Hibernate and any other JPA implementation manages the state of your entities and creates the required SQL statements to persist your changes in the database. That makes the implementation of most create, update and remove operations very easy and efficient.

```
1  EntityManager em = emf.createEntityManager();
2  em.getTransaction().begin();
3
4  Author a = em.find(Author.class, 1L);
5  a.setFirstName("Thorben");
6
7  em.getTransaction().commit();
8  em.close();
```

Projections For Read Operations

But read-only operations should be handled differently. Hibernate doesn't need to manage any states or perform dirty checks if you just want to read some data from the database.

So, from a theoretical point of view, DTOs should be the better projection for reading your data. But does it make a real difference?

I did a small performance test to answer this question.

Hibernate Tips: How to map the latest element of an association

5 Common Hibernate Mistakes That Cause Dozens of Unexpected Queries

Hibernate Tips: How to map an entity to multiple tables

Hibernate Tips: Easiest way to manage bi-directional associations

Hibernate & jOOQ – A Match Made in Heaven

Hibernate Tips: How to avoid Hibernate's MultipleBagFetchException

Getting Started with jOOQ – Building SQL Queries in Java

Composition vs. Inheritance with JPA and Hibernate

are associated by a many-to-one association. So, each *Book* was written by 1 *Author*.

```
1  @Entity
2  public class Author {
3
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
6      @Column(name = "id", updatable = false, nullable = false)
7      private Long id;
8
9      @Version
10     private int version;
11
12     private String firstName;
13
14     private String lastName;
15
16     @OneToMany(mappedBy = "author")
17     private List bookList = new ArrayList();
18
19     ...
20 }
```

To make sure that Hibernate doesn't fetch any extra data, I set the *FetchType* for the *@ManyToOne* association on the *Book* entity to *LAZY*. You can read more about the different FetchTypes and their effect in my Introduction to JPA FetchTypes.

```
1  @Entity
2  public class Book {
3
4      @Id
5      @GeneratedValue(strategy = GenerationType.AUTO)
```

```
9       @Version
10      private int version;
11
12      private String title;
13
14      @ManyToOne(fetch = FetchType.LAZY)
15      @JoinColumn(name = "fk_author")
16      private Author author;
17
18      ...
19  }
```

And I created a test database with 10 *Author*s. Each of them wrote 10 *Book*s. So the database contains 100 *Book*s in total.

In each test, I will use a different projection to select all 100 *Book*s and measure the time required to execute the query and the transaction. To reduce the impact of any side-effects, I do this 1000 times and measure the average time.

OK, so let's get started.

Selecting An Entity

Entity projections are the most popular ones in most applications. You already have the entity and JPA makes it easy to use them as a projection.

```
long timeTx = 0;
long timeQuery = 0;
long iterations = 1000;
// Perform 1000 iterations
for (int i = 0; i < iterations; i++) {
    EntityManager em = emf.createEntityManager();

    long startTx = System.currentTimeMillis();
    em.getTransaction().begin();

    // Execute Query
    long startQuery = System.currentTimeMillis();
    List<Book> books = em.createQuery("SELECT b FROM Book b").getResu
    long endQuery = System.currentTimeMillis();
    timeQuery += endQuery - startQuery;

    em.getTransaction().commit();
    long endTx = System.currentTimeMillis();

    em.close();
    timeTx += endTx - startTx;
}
System.out.println("Transaction: total " + timeTx + " per iteration "
System.out.println("Query: total " + timeQuery + " per iteration " +
```

On average, it takes 2ms to execute the query, retrieve the result and map it to 100 *Book* entities. And 2.89ms if you include the transaction handling. Not bad for a small and not so new laptop.

```
Transaction: total 2890 per iteration 2.89
Query: total 2000 per iteration 2.0
```

additional queries. By default, the *FetchtType* of a to-one association is EAGER which tells Hibernate to initialize the association immediately.

That requires additional queries and has a huge performance impact if your query selects multiple entities. Let's change the *Book* entity to use the default *FetchType* and perform the same test.

```
1  @Entity
2  public class Book {
3
4      @ManyToOne
5      @JoinColumn(name = "fk_author")
6      private Author author;
7
8      ...
9  }
```

That little change more than tripled the execution time of the test case. Instead of 2ms it now took 7.797ms to execute the query and map the result. And the time per transaction went up to 8.681ms instead of 2.89ms.

```
Transaction: total 8681 per iteration 8.681
Query: total 7797 per iteration 7.797
```

So, better make sure to set the *FetchType* to LAZY for your to-one associations.

question is: Does a query that returns entities annotated with *@Immutable* perform better?

Hibernate knows that it doesn't have to perform any dirty checks on these entities because they're immutable. That could result in a better performance. So, let's give it a try.

I added the following *ImmutableBook* entity to the test.

```
1  @Entity
2  @Table(name = "book")
3  @Immutable
4  public class ImmutableBook {
5
6      @Id
7      @GeneratedValue(strategy = GenerationType.AUTO)
8      @Column(name = "id", updatable = false, nullable = false)
9      private Long id;
10
11     @Version
12     private int version;
13
14     private String title;
15
16     @ManyToOne(fetch = FetchType.LAZY)
17     @JoinColumn(name = "fk_author")
18     private Author author;
19
20     ...
21 }
```

It's a copy of the *Book* entity with 2 additional annotations. The *@Immutable* annotation tells Hibernate that this entity can't be changed. And the *@Table(name = "book")* maps the entity to

```
long timeTx = 0;
long timeQuery = 0;
long iterations = 1000;
// Perform 1000 iterations
for (int i = 0; i < iterations; i++) {
    EntityManager em = emf.createEntityManager();

    long startTx = System.currentTimeMillis();
    em.getTransaction().begin();

    // Execute Query
    long startQuery = System.currentTimeMillis();
    List<Book> books = em.createQuery("SELECT b FROM ImmutableBook b'
            .getResultList();
    long endQuery = System.currentTimeMillis();
    timeQuery += endQuery - startQuery;

    em.getTransaction().commit();
    long endTx = System.currentTimeMillis();

    em.close();
    timeTx += endTx - startTx;
}
System.out.println("Transaction: total " + timeTx + " per iteration '
System.out.println("Query: total " + timeQuery + " per iteration " +
```

Interestingly enough, it doesn't make any difference, if the entity is immutable or not. The measured average execution time for the transaction and the query are almost identical to the previous test.

Selecting An Entity With *QueryHints.HINT_READONLY*

Andrew Bourgeois suggested to include a test with a read-only query. So, here it is.

This test uses the *Book* entity that I showed you at the beginning of the post. But it requires a change to test case.

JPA and Hibernate support a set of query hints which allow you to provide additional information about the query and how it should be executed. The query hint *QueryHints.HINT_READONLY* tells Hibernate to select the entities in read-only mode. So, Hibernate doesn't need to perform any dirty checks on them, and it can apply other optimizations.

You can set this hint by calling the *setHint* method on the *Query* interface.

```
1  long timeTx = 0;
2  long timeQuery = 0;
3  long iterations = 1000;
4  // Perform 1000 iterations
5  for (int i = 0; i < iterations; i++) {
6      EntityManager em = emf.createEntityManager();
7
8      long startTx = System.currentTimeMillis();
9      em.getTransaction().begin();
10
```

```
14        query.setHint(QueryHints.HINT_READONLY, true);
15        query.getResultList();
16        long endQuery = System.currentTimeMillis();
17        timeQuery += endQuery - startQuery;
18
19        em.getTransaction().commit();
20        long endTx = System.currentTimeMillis();
21
22        em.close();
23        timeTx += endTx - startTx;
24    }
25    System.out.println("Transaction: total " + timeTx + " per iteration '
26    System.out.println("Query: total " + timeQuery + " per iteration " +
```

You might expect that setting the query to read-only provides a noticeable performance benefit. Hibernate has to perform less work so it should be faster.

But as you can see below, the execution times are almost identical to the previous tests. At least in this test scenario, setting *QueryHints.HINT_READONLY* to true doesn't improve the performance.

```
Transaction: total 2842 per iteration 2.842
Query: total 2006 per iteration 2.006
```

Selecting A DTO

And you can, of course, also use constructor expressions in your Criteria queries.

```
long timeTx = 0;
long timeQuery = 0;
long iterations = 1000;
// Perform 1000 iterations
for (int i = 0; i < iterations; i++) {
    EntityManager em = emf.createEntityManager();

    long startTx = System.currentTimeMillis();
    em.getTransaction().begin();

    // Execute the query
    long startQuery = System.currentTimeMillis();
    List<BookValue> books = em.createQuery("SELECT new org.thoughts.c
    long endQuery = System.currentTimeMillis();
    timeQuery += endQuery - startQuery;

    em.getTransaction().commit();
    long endTx = System.currentTimeMillis();

    em.close();

    timeTx += endTx - startTx;
}
System.out.println("Transaction: total " + timeTx + " per iteration '
System.out.println("Query: total " + timeQuery + " per iteration " +
```

As expected, the DTO projection performs much better than the entity projection.

On average, it took 1.143ms to execute the query and 1.678ms to perform the transaction. That's is a performance improvement of ~43% for the query and ~42% for the transaction.

Not bad for a small change that just takes a minute to implement.

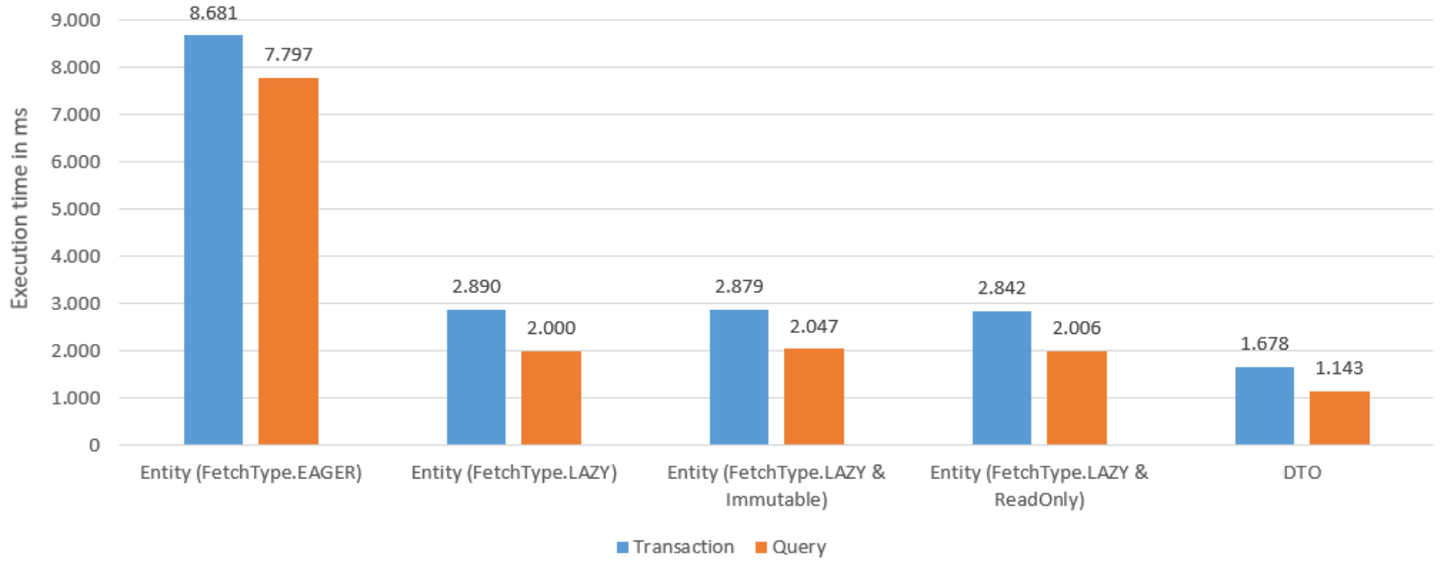
And in most projects, the performance improvement of the DTO projection will be even higher. It allows you to select the data that you need for your use case and not just all the attributes mapped by the entity. And selecting less data almost always results in a better performance.

Summary

Choosing the right projection for your use case is easier and more important than you might have thought.

When you want to implement a write operation, you should use an entity as your projection. Hibernate will manage its state, and you just have to update its attributes within your business logic. Hibernate will then take care of the rest.

You've seen the results of my small performance test. My laptop might not be the best environment to run these tests and it is definitely slower than your production environment. But the performance improvement is so big that it's obvious which projection you should use.



The query that used a DTO projection was ~40% faster than the one that selected entities. So, better spend the additional effort to create a DTO for your read-only operations and use it as the projection.

And you should also make sure to use *FetchType.LAZY* for all associations. As you've seen in the test, even one eagerly fetched to-one association might triple the execution time of your query. So, better use *FetchType.LAZY* and initialize the relationships that you need for your use case.

Related Posts:

1. Standardized schema generation and data loading with JPA 2.1

4. Why you should avoid CascadeType.REMOVE for to-many associations and what to do instead

---

Filed Under: Hibernate, JPA

Comments

July 24, 2017 at 12:16 pm

Excellent Article Thorben.
My biggest concern with DTOs is when we have large entities with a lot of fields. The constructor would start to get big, and hard to create queries.

What about @Immutable entities?
It would be a good comparison in your article to add @Immutable entities to the tests.
What do you think?

Thanks,

Reply

Thorben Janssen says
July 24, 2017 at 2:30 pm

Thanks for the kind words and the interesting question, Joao!

I added an immutable annotation to the test: http://www.thoughts-on-java.org/entities-dtos-use-projection/#immutable
And the result is interesting. I expected that it would perform better than a “normal”

Regarding the constructor: Yes, that's definitely a downside of the constructor expression. If you're selecting a lot of columns, the constructor gets huge and the query gets hard to read and maintain.

Reply

Joao Charnet says
July 24, 2017 at 4:12 pm

WOW! That was fast! 🙂
Thanks, Thorben.
And too bad the immutable entity does not perform better than a mutable one!
This would be very interesting to solve complex and large entities queries.

Thanks again!

Reply

Christian Beikov says
July 26, 2017 at 6:20 am

persistence#entity-view-usage

Reply

Andrew Bourgeois says
July 24, 2017 at 7:58 pm

1) what about:
query.setHint(QueryHints.HINT_READONLY, true);
when doing an entity projection? I use it but I have yet to benchmark it.

2) try this:
List books = em.createQuery("SELECT new org.thoughts.on.java.model.BookValue(b.id, b.title, a) FROM Book b JOIN FETCH b.author a").getResultList();
and enjoy this issue: https://hibernate.atlassian.net/browse/HHH-3345.

Reply

Thanks for the suggestion Andrew. I added a test using the read-only query hint here: http://www.thoughts-on-java.org/entities-dtos-use-projection/#readonly
Similar to the @Immutable test, you could expect better performance. But at least in this test scenario, it doesn't provide any noticeable performance improvement.

And yes, the constructor expression has several limitations. With Hibernate's current implementation, you can't reference entities (= you need to reference entity attributes) or attributes that map associations. You also can't use multiple constructor expressions.
I hope that they will improve that in Hibernate 6.

Reply

Clement Ojo says
July 25, 2017 at 4:48 pm

Thanks Thorben, you always have nice tutorials.

Reply

Thanks Clement 🙂

Reply

Weliff Lima says
July 28, 2017 at 5:28 pm

Great article! Thanks

Reply

Thorben Janssen says
July 30, 2017 at 4:10 am

Thank you Weliff

Reply

Ajay says
July 28, 2017 at 9:14 pm

nice explanation. Thank you.

Reply

> Thorben Janssen says
> July 30, 2017 at 4:10 am
>
> Thanks Ajay
>
> Reply

Ibanga Enoobong Ime says
July 29, 2017 at 1:30 pm

Excellent article my good man, I usually do do this em.createQuery(“select b.name from Book b, String.class”).getResultList(); Optimal or not?

Thorben Janssen says
July 30, 2017 at 4:10 am

Yes, that's also OK.

Reply

Mohammed Salman says
November 15, 2017 at 4:14 am

Very illustrative article! I'm an beginner and these things are so much helpful for newbies like me. One silly question, do we need to provide parameterized constructor for using these constructor queries?

Reply

Thorben Janssen says
December 13, 2017 at 12:34 pm

Yes, your class needs a constructor that matches the constructor expression.
The constructor expression that you can use in JPQL or CriteriaQuery just describe a constructor call which Hibernate will execute for each record in the result set.

Regards,
Thorben

Reply

Michel Kapel says
November 24, 2017 at 4:45 pm

Many thakns for this really interesting article.
I wonder, what about ResultMappings for DTOs?
If you map the result straight to a DTO does the performance stand or does the fact that ResultMappings is still one more annotation on an Entity (and hence falls back on EntityManager work) just take you back to square one.

Will have to test this sometime

Reply

December 1, 2017 at 5:38 pm

Hi Michel,

I haven't tested it but I expect that the ResultMapping should provide a similar performance as the constructor expression.
I'll put it on my list and extend the test and this post in the future.

Regards,
Thorben

Reply

Leandro Bianchini says
December 12, 2017 at 2:04 pm

Hi Thorben,

Nice article, as always.
I would like to ask you how to handle, using DTO approach, with the following scenarios:

Even better, fetch data from database with a single query using a Author DTO class that has a List of BookDTO class as its property.

Should I use more than just one query to do that? Filling the property member using another query inside a loop of the main's query result?

Thanks in advance!!

Reply

Thorben Janssen says
December 13, 2017 at 12:20 pm

Hi Leandro,

that depends on your use case and the number of records you're selecting.

In the 1st case (BookDTO + 1 AuthorDTO), you can implement a constructor on the BookDTO which instantiates the BookDTO and the AuthorDTO. Unfortunately, you can't use more than 1 constructor expression in your projection. So, this is the only option to instantiate both DTOs in 1 query.

If you use DTO's, you most often need to perform too many queries to fetch the associated BookDTOs. The additional queries then outweigh the performance benefit of the DTO projection.

Regards,
Thorben

Reply

Leandro Bianchini says
December 31, 2017 at 4:09 am

Thanks for your answer!

Reply

Manuel says
January 6, 2018 at 9:44 am

"constructor on the BookDTO which instantiates the BookDTO and the AuthorDTO" means using ResultTransformer?

Thorben Janssen says
January 9, 2018 at 5:39 pm

A ResultTransformer is also an option. But if each Book was written by 1 Author, you can also implement a constructor on the BookDTO which takes the attributes of the BookDTO and the AuthorDTO. You can then instantiate the AuthorDTO within the constructor of the BookDTO.
That's not the most beautiful solution, but you can't use more than 1 constructor expression per query. So, it's a necessary workaround.

Reply

Manuel says
January 6, 2018 at 9:04 am

Nice article, I have a question. It is fair to think that eager fetching is a bad thing but we do DTO join fetch on multiple tables?

Reply

Thorben Janssen says
January 9, 2018 at 5:35 pm

Hi Manuel,

The DTO is your best option as long as you load only the data that you need for the specific use case.

Regards,
Thorben

Reply

anayisse says
April 7, 2018 at 2:30 pm

Maybe a stupid question from a newbie is Book class and BookValue class are the same?

Thorben Janssen says
April 21, 2018 at 2:52 pm

Hi,

no, they are not the same.
The Book class is an entity which gets managed by the persistence context. E.g., when you load a Book entity from the database and change any of its attributes, Hibernate will generate an SQL UPDATE statement to persist that change in the database.
The BookValue class is a simple DTO which just gets loaded from the database. It's a simple class which doesn't get managed by the persistence context.

Regards,
Thorben

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT