

REFCARDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars)

ZONES ~

DZone (/) > Database Zone (/database-sql-nosql-tutorials-tools-news) > How to Decide Between JOIN and JOIN FETCH

How to Decide Between JOIN and JOIN FETCH



by Anghel Leonard (/users/196910/anghelleonard.html) RMVB ⋄ CORE · Sep. 21, 20

· Database Zone (/database-sql-nosql-tutorials-tools-news) · Tutorial

```
⚠ Like (9) Comment (0)
                        ☆ Save
                                ৺ Tweet
```

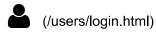
Typically, JOIN and JOIN FETCH come into play when the application has lazy associations but some data must be fetched eagerly. Relying on FetchType.EAGER at the entities-level is a code smell.

Consider the Author and Book entities that are involved in a bidirectional-lazy @OneToMany association:

```
Java
 1 @Entity
 2 public class Author implements Serializable {
 4
     private static final long serialVersionUID = 1L;
 5
 6
 7
     private Long id;
     private String name;
 8
 9
     private String genre;
10
     private int age;
11
     @OneToMany(cascade = CascadeType.ALL, mappedBy = "author", orphanRemoval = true)
12
     private List<Book> books = new ArrayList<>();
13
14
                                                  X
```

private static final long serialVersionUID = 1L;



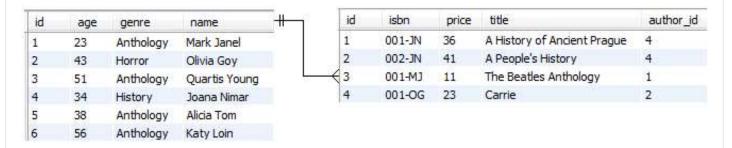


REFEARDZ (Atefcatetz)ng TREND REPORTS (/trendreports) EVENTS (/webinars)

ZONES ~

```
10
     @ManyToOne(fetch = FetchType.LAZY)
     @JoinColumn(name = "author_id")
11
     private Author author;
13
14 }
```

Consider the following sample of data:



And, the goal is to fetch the following data as entities:

- all Author and their Book that are more expensive than the given price
- all the Book and their Author

Fetch All Authors and Their Books that Are More Expensive than The Given Price

To satisfy the first query (fetch all the Author and their Book that are more expensive than the given price) write a Spring repository, AuthorRepository, and add a JOIN and a JOIN FETCH query meant to fetch the same data:

```
Java
1 @Repository
2 @Transactional(readOnly = true)
3 public interface AuthorRepository extends JpaRepository<Author, Long> {
4
    // INNER JOIN
5
    @Query(value = "SELECT a FROM Author a INNER JOIN a.books b WHERE b.price > ?1")
6
7
    List<Author> fetchAuthorsBooksByPriceInnerJoin(int price);
8
```

REFCARDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars) ZONES ~

```
1 public void fetchAuthorsBooksByPriceJoinFetch() {
     List<Author> authors = authorRepository.fetchAuthorsBooksByPriceJoinFetch(40);
 3
 4
 5
     authors.forEach((e) -> System.out.println("Author name: "
         + e.getName() + ", books: " + e.getBooks()));
 6
 7 }
 8
 9 @Transactional(readOnly = true)
10 public void fetchAuthorsBooksByPriceInnerJoin() {
11
     List<Author> authors = authorRepository.fetchAuthorsBooksByPriceInnerJoin(40);
12
13
     authors.forEach((e) -> System.out.println("Author name: "
14
         + e.getName() + ", books: " + e.getBooks()));
15
16 }
```

How JOIN FETCH Will Act

JOIN FETCH is specific to JPA and it allows associations to be initialized along with their parent objects using a single SELECT . As you will see soon, this is particularly useful for fetching associated collections. This means that calling

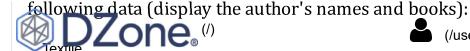
fetchAuthorsBooksByPriceJoinFetch() will trigger a single SELECT as follows:

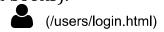
```
SQL
```

```
1 SELECT
     author0 .id AS id1 0 0 ,
     books1 .id AS id1 1 1,
 3
 4
     author0_.age AS age2_0_0_,
     author0_.genre AS genre3_0_0_,
 5
 6
     author0_.name AS name4_0_0_,
 7
     books1_.author_id AS author_i5_1_1_,
 8
     books1_.isbn AS isbn2_1_1_,
 9
     books1_.price AS price3_1_1_,
10
     books1_.title AS title4_1_1_,
     books1_.author_id AS author_i5_1_0__,
11
     books1 .id AS id1 1 0
12
13 FROM author author0_
14 INNER JOIN book books1_
```

X

Running this SQL against the data sample for a given price of 40 dollars will fetch the





REFCARDZr(/refrardZ)partREND REPORTS (/trendreports) EVENTS (/webinars) ZONES V DOOKS. [BOOK(1d-2, Little-A Feople's History, isbn-002-5N, price-41)]

This looks correct! There is a single book in the database expensive than 40 dollars and its author is Joana Nimar.

How JOIN Will Act

On the other hand, JOIN **doesn't** allow associated collections to be initialized along with their parent objects using a single SELECT. This means that calling fetchAuthorsBooksByPriceInnerJoin() will result in the following SELECT (the SQL reveals that no book was loaded):

```
SQL

1 SELECT

2 author0_.id AS id1_0_,

3 author0_.age AS age2_0_,

4 author0_.genre AS genre3_0_,

5 author0_.name AS name4_0_

6 FROM author author0_

7 INNER JOIN book books1_

8 ON author0_.id = books1_.author_id

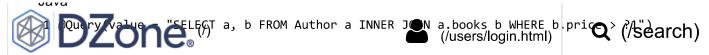
9 WHERE books1_.price > ?
```

Running this SQL against the data sample will fetch a single author (Joana Nimar) which is correct. Attempting to display the books written by Joana Nimar via getBooks() will trigger an additional SELECT as follows:

```
SQL
1 SELECT
2
    books0_.author_id AS author_i5_1_0_,
3
    books0 .id AS id1 1 0 ,
    books0_.id AS id1_1_1_,
4
5
    books0_.author_id AS author_i5_1_1_,
6
    books0_.isbn AS isbn2_1_1_,
7
    books0_.price AS price3_1_1_,
    books0_.title AS title4_1_1_
9 FROM book books0_
```

X

W



REFCARDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars) ZONES ~

Display the author name and the fetched books:

Two things must be highlighted here: an important drawback and potential confusion.

First, the drawback. Notice that JOIN has fetched the books in an additional SELECT. This can be considered a performance penalty in comparison with JOIN FETCH which needs a single SELECT, therefore a single database roundtrip.

Second, the potential confusion. Pay extra attention to the interpretation of the WHERE books1_.price > ? clause in the first SELECT. While the application fetches only the authors that have written books more expensive than 40 dollars, when calling getBooks(), the application fetches all books of these authors not only the books more expensive than 40 dollars. This is normal since, when getBooks() is called, the WHERE clause is not there anymore. Therefore, in this case, JOIN produced a different result than JOIN FETCH.

Fetch All Book and Their Author

To satisfy the second query (all the Book and their Author) write a Spring repository, BookRepository, and add two JOINs and a JOIN FETCH query:

```
Java
1  @Repository
2  @Transactional(readOnly = true)
3  public interface BookRepository extends JpaRepository<Book, Long> {
4
5    // INNER JOIN BAD
6    @Query(value = "SELECT b FROM Book b INNER JOIN b.author a")
```

X

REFCÁRDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars)

ZONES ~

Calling the above methods and displaying the fetched data to the console can be done as follows:

```
Java
1 public void fetchBooksAuthorsJoinFetch() {
2
3
     List<Book> books = bookRepository.fetchBooksAuthorsJoinFetch();
4
    books.forEach((e) -> System.out.println("Book title: " + e.getTitle()
5
        + ", Isbn:" + e.getIsbn() + ", author: " + e.getAuthor()));
6
7 }
8
9 @Transactional(readOnly = true)
10 public void fetchBooksAuthorsInnerJoinBad/Good() {
11
     List<Book> books = bookRepository.fetchBooksAuthorsInnerJoinBad/Good();
12
13
     books.forEach((e) -> System.out.println("Book title: " + e.getTitle()
14
        + ", Isbn: " + e.getIsbn() + ", author: " + e.getAuthor()));
15
16 }
```

How JOIN FETCH Will Act

Calling fetchBooksAuthorsJoinFetch() will trigger a single SQL triggered as follows (all authors and books are fetched in a single SELECT):

```
SQL
 1 SELECT
 2
     book0_.id AS id1_1_0_,
     author1_.id AS id1_0_1_,
 3
     book0_.author_id AS author_i5_1_0_,
 4
 5
     book0 .isbn AS isbn2 1 0 ,
 6
     book0_.price AS price3_1_0_,
 7
     book0_.title AS title4_1_0_,
     author1_.age AS age2_0_1_,
 8
 9
     author1_.genre AS genre3_0_1_,
     author1_.name AS name4_0_1_
10
11 FROM book book0_
12 INNER JOIN author author1
```

X

RL.....,,,,,,,,,,,,, ISBN, and author):





1 Book title: A History of Ancient Prague, Isbn:001-JN,

REFGARDZ (/refgardz) uth REND REPORTS (/webjnars)

ZONES ~

```
Book title: A People's History, Isbn:002-JN,
author: Author{id=4, name=Joana Nimar, genre=History, age=34}

Book title: The Beatles Anthology, Isbn:001-MJ,
author: Author{id=1, name=Mark Janel, genre=Anthology, age=23}

Book title: Carrie, Isbn:001-OG,
author: Author{id=2, name=Olivia Goy, genre=Horror, age=43}
```

Everything looks as expected! There are four books and each of them has an author.

How JOIN Will Act

On the other hand, calling fetchBooksAuthorsInnerJoinBad() will trigger a single SQL as follows (the SQL reveals that no author was loaded):

```
SQL

1 SELECT

2 book0_.id AS id1_1_,

3 book0_.author_id AS author_i5_1_,

4 book0_.isbn AS isbn2_1_,

5 book0_.price AS price3_1_,

6 book0_.title AS title4_1_

7 FROM book book0_

8 INNER JOIN author author1_

9 ON book0_.author_id = author1_.id
```

The returned List<Book> contains four Book. Looping this list and fetching the author of each book via getAuthor() will trigger three additional SELECT statements (there are three SELECT statements instead of four because two of the books have the same author, therefore, for the second of these two books, the author will be fetched from the Persistence Context). So, the below SELECT is triggered three times with different id value:

```
SQL
1 SELECT
```

X

```
7 WHERE author0_.id = ?
```





Displaying the title, ISBN, and author of each book will output: REFCARDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars) Z

ZONES .

```
Textile
 1 Book title: A History of Ancient Prague, Isbn: 001-JN,
        author: Author{id=4, name=Joana Nimar, genre=History, age=34}
 2
 3
 4 Book title: A People's History, Isbn: 002-JN,
        author: Author{id=4, name=Joana Nimar, genre=History, age=34}
 5
 6
 7 Book title: The Beatles Anthology, Isbn: 001-MJ,
        author: Author{id=1, name=Mark Janel, genre=Anthology, age=23}
 8
 9
10 Book title: Carrie, Isbn: 001-OG,
        author: Author{id=2, name=Olivia Goy, genre=Horror, age=43}
11
```

In this case, the performance penalty is obvious. While JOIN FETCH needs a single SELECT, JOIN needs four SELECT statements.

How about calling fetchBooksAuthorsInnerJoinGood()? Well, this will produce the exact same query and result as JOIN FETCH. This is working because the fetched association is not a collection. So, in this case, you can use JOIN or JOIN FETCH.

As a rule of thumb, use JOIN FETCH (not JOIN) whenever the data should be fetched as entities (because the application plans to modify them) and Hibernate should include the associations in the SELECT clause. This is particularly useful for fetching associated collections. In such scenarios, using JOIN is prone to N+1 performance penalties. On the other hand, whenever fetching read-only data (don't plan to modify it), better rely on JOIN + DTO instead of JOIN FETCH.

Pay attention that while a query as SELECT a FROM Author a JOIN FETCH a.books is correct, the following attempts will not work:

X

SELECT a.age as age FROM Author a JOIN FETCH a.books

Ca

th

Seleci a FROM Author a JOIN FEICH a.DOOKS.TITTE

REFCARDZ (/refcardz) TREND REPORTS (/trendreports) EVENTS (/webinars) ZONES

The source code is available on GitHub

(https://github.com/AnghelLeonard/Hibernate-SpringBoot/tree/master/HibernateSpringBootJoinVSJoinFetch).

If you liked this article, then you'll my book containing 150+ performance items - **Spring Boot Persistence Best Practices** (https://www.amazon.com/Spring-Boot-Persistence-Best-Practices-dp-1484256255/dp/1484256255/).

This book helps every Spring Boot developer to squeeze the performances of the persistence layer.

Fetch (FTP Client) Joins (Concurrency Library) Database Book Spring Framework Sql Data (Computing)

Opinions expressed by DZone contributors are their own.

Popular on DZone

- One Hundred Years of Hate [Comic] (/articles/one-hundred-years-of-hate-comic? fromrel=true)
- Modern Enterprise Data Architecture (/articles/modern-enterprise-data-architecture? fromrel=true)
- Five Ways of Synchronising Multithreaded Integration Tests (/articles/five-wayssynchronising?fromrel=true)
- Top Tips for Powerful Data-Driven Storytelling (/articles/top-tips-for-powerful-data-driven-storytelling?fromrel=true)

X





(/users/login.html)

Q (/search)

(mailto:support@dzone.com) etcardz) TREND REPORTS (/trendreports) EVENTS (/webinars) ://careers.dzone.com/)

Sitemap (/sitemap)

ADVERTISE

Advertise with DZone (https://advertise.dzone.com)

CONTRIBUTE ON DZONE

Article Submission Guidelines (/articles/dzones-article-submission-guidelines)

MVB Program (/pages/mvb)

Become a Contributor (/pages/contribute)

Visit the Writers' Zone (/writers-zone)

LEGAL

Terms of Service (/pages/tos) Privacy Policy (/pages/privacy)

CONTACT US 600 Park Offices Drive Suite 300

Durham, NC 27709

support@dzone.com (mailto:support@dzone.com)

+1 (919) 678-0300 (tel:+19196780300)

Let's be friends:

(/paglest/fristills/sittles/vlar/stables/instables/pagles/fried/pagles/fried/stables/instables/pagles/fried/stable

DZone.com is powered by

(https://devada.com/answerhub/)