How to Decide Between JOIN and JOIN FETCH

by Anghel Leonard (/users/196910/anghelleonard.html) MVB CORE · Sep. 21, 20 · Database Zone (/database-sql-nosql-tutorials-tools-news) · Tutorial

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

```
@Entity
public class Author implements Serializable {

  private static final long serialVersionUID = 1L;

  @Id
  private Long id;
  private String name;
  private String genre;
  private int age;

  @OneToMany(cascade = CascadeType.ALL, mappedBy = "author", orphanRemoval = true)
  private List<Book> books = new ArrayList<>();
  ...
```

X

```
  private static final long serialVersionUID = 1L;
```

```
5
6   @Id private Long id;
7   private String title;
8   private String isbn;
9
10  @ManyToOne(fetch = FetchType.LAZY)
11  @JoinColumn(name = "author_id")
12  private Author author;
13  ...
14 }
```

Consider the following sample of data:

id	age	genre	name
1	23	Anthology	Mark Janel
2	43	Horror	Olivia Goy
3	51	Anthology	Quartis Young
4	34	History	Joana Nimar
5	38	Anthology	Alicia Tom
6	56	Anthology	Katy Loin

id	isbn	price	title	author_id
1	001-JN	36	A History of Ancient Prague	4
2	002-JN	41	A People's History	4
3	001-MJ	11	The Beatles Anthology	1
4	001-OG	23	Carrie	2

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
5   // INNER JOIN
6   @Query(value = "SELECT a FROM Author a INNER JOIN a.books b WHERE b.price > ?1")
7   List<Author> fetchAuthorsBooksByPriceInnerJoin(int price);
8
```

X

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

Java

```
public void fetchAuthorsBooksByPriceJoinFetch() {

  List<Author> authors = authorRepository.fetchAuthorsBooksByPriceJoinFetch(40);

  authors.forEach((e) -> System.out.println("Author name: "
      + e.getName() + ", books: " + e.getBooks()));
}

@Transactional(readOnly = true)
public void fetchAuthorsBooksByPriceInnerJoin() {

  List<Author> authors = authorRepository.fetchAuthorsBooksByPriceInnerJoin(40);

  authors.forEach((e) -> System.out.println("Author name: "
      + e.getName() + ", books: " + e.getBooks()));
}
```

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

```
SELECT
  author0_.id AS id1_0_0_,
  books1_.id AS id1_1_1_,
  author0_.age AS age2_0_0_,
  author0_.genre AS genre3_0_0_,
  author0_.name AS name4_0_0_,
  books1_.author_id AS author_i5_1_1_,
  books1_.isbn AS isbn2_1_1_,
  books1_.price AS price3_1_1_,
  books1_.title AS title4_1_1_,
  books1_.author_id AS author_i5_1_0__,
  books1_.id AS id1_1_0__
FROM author author0_
INNER JOIN book books1_
```

X

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

following data (display the author's names and books):

Textile

```
Author{name=Joana Nimar, genre=Horror, books: [Book{id=2, title=A People's History, isbn=002-JN, 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

```
SELECT
  author0_.id AS id1_0_,
  author0_.age AS age2_0_,
  author0_.genre AS genre3_0_,
  author0_.name AS name4_0_
FROM author author0_
INNER JOIN book books1_
  ON author0_.id = books1_.author_id
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

```
SELECT
  books0_.author_id AS author_i5_1_0_,
  books0_.id AS id1_1_0_,
  books0_.id AS id1_1_1_,
  books0_.author_id AS author_i5_1_1_,
  books0_.isbn AS isbn2_1_1_,
  books0_.price AS price3_1_1_,
  books0_.title AS title4_1_1_
FROM book books0_
```

X

W

Java

Display the author name and the fetched books:

Textile

```
Author name: Joana Nimar,
       books: [
                Book{id=1, title=A History of Ancient Prague, isbn=001-JN, price=36},
                Book{id=2, title=A People's History, isbn=002-JN, price=41}
               ]
```

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 `JOIN`s and a `JOIN FETCH` query:

Java

```
@Repository
@Transactional(readOnly = true)
public interface BookRepository extends JpaRepository<Book, Long> {

  // INNER JOIN BAD
  @Query(value = "SELECT b FROM Book b INNER JOIN b.author a")

  // JOIN FETCH
```

```
13   // JOIN FETCH
14   @Query(value = "SELECT b FROM Book b JOIN FETCH b.author a")
15   List<Book> fetchBooksAuthorsJoinFetch();
16 }
```


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

Java

```
public void fetchBooksAuthorsJoinFetch() {

  List<Book> books = bookRepository.fetchBooksAuthorsJoinFetch();

  books.forEach((e) -> System.out.println("Book title: " + e.getTitle()
     + ", Isbn:" + e.getIsbn() + ", author: " + e.getAuthor()));
}

@Transactional(readOnly = true)
public void fetchBooksAuthorsInnerJoinBad/Good() {

  List<Book> books = bookRepository.fetchBooksAuthorsInnerJoinBad/Good();

  books.forEach((e) -> System.out.println("Book title: " + e.getTitle()
     + ", Isbn: " + e.getIsbn() + ", author: " + e.getAuthor()));
}
```

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

```
SELECT
  book0_.id AS id1_1_0_,
  author1_.id AS id1_0_1_,
  book0_.author_id AS author_i5_1_0_,
  book0_.isbn AS isbn2_1_0_,
  book0_.price AS price3_1_0_,
  book0_.title AS title4_1_0_,
  author1_.age AS age2_0_1_,
  author1_.genre AS genre3_0_1_,
  author1_.name AS name4_0_1_
FROM book book0_
INNER JOIN author author1_
```


Running this SQL against the database will output (display only the book title, ISBN, and author):

```
Book title: A History of Ancient Prague, Isbn:001-JN,
     author: Author{id=4, name=Joana Nimar, genre=History, age=34}

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

```
SELECT
  book0_.id AS id1_1_,
  book0_.author_id AS author_i5_1_,
  book0_.isbn AS isbn2_1_,
  book0_.price AS price3_1_,
  book0_.title AS title4_1_
FROM book book0_
INNER JOIN author author1_
  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

```
SELECT
```

X

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

Displaying the title, ISBN, and author of each book will output:

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

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:

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

C
th

```
SELECT a FROM Author a JOIN FETCH a.books.title
```

Causes: *org.hibernate.QueryException: illegal attempt to dereference collection [author0_.id.books] with element property reference [title]*

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)

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-ways-synchronising?fromrel=true)**
- **Top Tips for Powerful Data-Driven Storytelling (/articles/top-tips-for-powerful-data-driven-storytelling?fromrel=true)**

CONTACT US
600 Park Offices Drive
Suite 300
Durham, NC 27709
support@dzone.com (mailto:support@dzone.com)
+1 (919) 678-0300 (tel:+19196780300)