Vlad Mihalcea's Blog

Teaching is my way of learning

A beginner's guide to JPA and Hibernate Cascade Types

MARCH 5, 2015SEPTEMBER 29, 2017 / VLADMIHALCEA

Introduction

JPA (http://en.wikipedia.org/wiki/Java_Persistence_API) translates entity state transitions (/2014/07/30/a-beginners-guide-to-jpahibernate-entity-state-transitions/) to database DML (http://en.wikipedia.org/wiki/Data_manipulation_language) statements. Because it's common to operate on entity graphs, *JPA* allows us to propagate entity state changes from *Parents* to *Child* entities.

This behavior is configured through the CascadeType (http://docs.oracle.com/javaee/7/api/javax/persistence/CascadeType.html) mappings.

JPA vs *Hibernate Cascade Types*

Hibernate supports all *JPA* Cascade Types and some additional legacy cascading styles. The following table draws an association between *JPA* Cascade Types and their *Hibernate* native *API* equivalent:

***JPA EntityManager* action**	***JPA C***
detach(entity) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#detach%28java.lang.Object%29)	DETA (http:
merge(entity) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#merge%28T%29)	MERG (http:
persist(entity) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#persist%28java.lang.Object%29)	PERSI (http:
refresh(entity) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#refresh%28java.lang.Object%29)	REFR (http:
remove(entity) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#remove%28java.lang.Object%29)	REMO (http:
lock(entity, lockModeType) (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#lock%28java.lang.Object,%20javax.persistence.LockModeType%29)	
All the above EntityManager methods	ALL (

From this table we can conclude that:

- There's no difference between calling *persist*, *merge* or *refresh* on the *JPA* EntityManager (http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html) or the *Hibernate Session* (https://docs.jboss.org/hibernate/orm/4.3/javadocs/org/hibernate/Session.html).
- The *JPA remove* and *detach* calls are delegated to *Hibernate delete* and *evict* native operations.
- Only *Hibernate* supports *replicate* and *saveOrUpdate*. While *replicate* is useful for some very specific scenarios (when the exact entity state needs to be mirrored between two distinct *DataSources*), the *persist* and *merge* combo is always a better alternative than the native *saveOrUpdate* operation.

As a rule of thumb, you should always use *persist* for *TRANSIENT* entities and merge for *DETACHED* ones.

The *saveOrUpdate* shortcomings (when passing a detached entity snapshot to a *Session* already managing this entity) had lead to the *merge* operation predecessor: the now extinct saveOrUpdateCopy (https://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/classic/Session.html#saveOrUpdateCopy%28java.lang.Object%29) operation.

- The *JPA* lock method shares the same behaviour with *Hibernate* lock request method.
- The *JPA* CascadeType.ALL (http://docs.oracle.com/javaee/7/api/javax/persistence/CascadeType.html#ALL) doesn't only apply to *EntityManager* state change operations, but to all *Hibernate* CascadeTypes (https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/annotations/CascadeType.html) as well.

  So if you mapped your associations with *CascadeType.ALL*, you can still cascade *Hibernate* specific events. For example, you can cascade the *JPA* lock operation (although it behaves as reattaching, instead of an actual lock request propagation), even if *JPA* doesn't define a *CascadeType.LOCK*.

Cascading best practices

Cascading only makes sense only for *Parent* – *Child* associations (the *Parent* entity state transition being cascaded to its Child entities). Cascading from *Child* to *Parent* is not very useful and usually, it's a mapping code smell.

Next, I'm going to take analyse the cascading behaviour of all *JPA Parent* – *Child* associations.

One-To-One

The most common *One-To-One* bidirectional association looks like this:

```java
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @OneToOne(mappedBy = "post",
        cascade = CascadeType.ALL, orphanRemoval = true)
    private PostDetails details;

    public Long getId() {
        return id;
    }

    public PostDetails getDetails() {
        return details;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void addDetails(PostDetails details) {
        this.details = details;
        details.setPost(this);
    }

    public void removeDetails() {
        if (details != null) {
            details.setPost(null);
        }
        this.details = null;
    }
}

@Entity
public class PostDetails {

    @Id
    private Long id;

    @Column(name = "created_on")
    @Temporal(TemporalType.TIMESTAMP)
    private Date createdOn = new Date();

    private boolean visible;

    @OneToOne
    @JoinColumn(name = "id")
    @MapsId
    private Post post;

    public Long getId() {
        return id;
    }

    public void setVisible(boolean visible) {
        this.visible = visible;
    }

    public void setPost(Post post) {
        this.post = post;
    }
}
```

The *Post* entity plays the *Parent* role and the *PostDetails* is the *Child*.

> The bidirectional associations should always be updated on both sides, therefore the *Parent* side should contain the *addChild* and *removeChild* combo. These methods ensure we always synchronize both sides of the association, to avoid object or relational data corruption issues.

In this particular case, the *CascadeType.ALL* and orphan removal make sense because the *PostDetails* life-cycle is bound to that of its *Post Parent* entity.

Cascading the *one-to-one* persist operation

The *CascadeType.PERSIST* comes along with the *CascadeType.ALL* configuration, so we only have to persist the *Post* entity, and the associated *PostDetails* entity is persisted as well:

```
Post post = new Post();
post.setName("Hibernate Master Class");

PostDetails details = new PostDetails();

post.addDetails(details);

session.persist(post);
```

Generating the following output:

```
INSERT INTO post(id, NAME)
VALUES (DEFAULT, Hibernate Master Class'')

insert into PostDetails (id, created_on, visible)
values (1, '2015-03-03 10:17:19.14', false)
```

Cascading the *one-to-one* merge operation

The *CascadeType.MERGE* is inherited from the *CascadeType.ALL* setting, so we only have to merge the *Post* entity and the associated *PostDetails* is merged as well:

```
Post post = newPost();
post.setName("Hibernate Master Class Training Material");
post.getDetails().setVisible(true);

doInTransaction(session -> {
    session.merge(post);
});
```

The merge operation generates the following output:

```
SELECT onetooneca0_.id     AS id1_3_1_,
   onetooneca0_.NAME       AS name2_3_1_,
   onetooneca1_.id         AS id1_4_0_,
   onetooneca1_.created_on AS created_2_4_0_,
   onetooneca1_.visible    AS visible3_4_0_
FROM   post onetooneca0_
LEFT OUTER JOIN postdetails onetooneca1_
    ON onetooneca0_.id = onetooneca1_.id
WHERE  onetooneca0_.id = 1

UPDATE postdetails SET
    created_on = '2015-03-03 10:20:53.874', visible = true
WHERE  id = 1

UPDATE post SET
    NAME = 'Hibernate Master Class Training Material'
WHERE  id = 1
```

Cascading the *one-to-one* delete operation

The *CascadeType.REMOVE* is also inherited from the *CascadeType.ALL* configuration, so the *Post* entity deletion triggers a *PostDetails* entity removal too:

```
Post post = newPost();

doInTransaction(session -> {
    session.delete(post);
});
```

Generating the following output:

```
delete from PostDetails where id = 1
delete from Post where id = 1
```

The *one-to-one delete orphan* cascading operation

If a *Child* entity is dissociated from its *Parent*, the *Child Foreign Key* is set to *NULL*. If we want to have the *Child* row deleted as well, we have to use the *orphan removal* support.

```
doInTransaction(session -> {
    Post post = (Post) session.get(Post.class, 1L);
    post.removeDetails();
});
```

The *orphan removal* generates this output:

```
SELECT onetooneca0_.id         AS id1_3_0_,
       onetooneca0_.NAME       AS name2_3_0_,
       onetooneca1_.id         AS id1_4_1_,
       onetooneca1_.created_on AS created_2_4_1_,
       onetooneca1_.visible    AS visible3_4_1_
FROM   post onetooneca0_
LEFT OUTER JOIN postdetails onetooneca1_
    ON onetooneca0_.id = onetooneca1_.id
WHERE  onetooneca0_.id = 1

delete from PostDetails where id = 1
```

Unidirectional *one-to-one* association

Most often, the *Parent* entity is the inverse side (e.g. *mappedBy*), the *Child* controlling the association through its Foreign Key. But the cascade is not limited to bidirectional associations, we can also use it for unidirectional relationships:

```
@Entity
public class Commit {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String comment;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinTable(
        name = "Branch_Merge_Commit",
        joinColumns = @JoinColumn(
            name = "commit_id",
            referencedColumnName = "id"),
        inverseJoinColumns = @JoinColumn(
            name = "branch_merge_id",
            referencedColumnName = "id")
    )
    private BranchMerge branchMerge;

    public Commit() {
    }

    public Commit(String comment) {
        this.comment = comment;
    }

    public Long getId() {
        return id;
    }

    public void addBranchMerge(
        String fromBranch, String toBranch) {
        this.branchMerge = new BranchMerge(
             fromBranch, toBranch);
    }

    public void removeBranchMerge() {
        this.branchMerge = null;
    }
}

@Entity
public class BranchMerge {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String fromBranch;

    private String toBranch;

    public BranchMerge() {
    }

    public BranchMerge(
        String fromBranch, String toBranch) {
        this.fromBranch = fromBranch;
        this.toBranch = toBranch;
    }

    public Long getId() {
        return id;
    }
}
```

Cascading consists in propagating the *Parent* entity state transition to one or more *Child* entities, and it can be used for both unidirectional and bidirectional associations.

One-To-Many

The most common *Parent* – *Child* association consists of a *one-to-many* and a *many-to-one* relationship, where the cascade being useful for the *one-to-many* side only:

```
@Entity
public class Post {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @OneToMany(cascade = CascadeType.ALL,
        mappedBy = "post", orphanRemoval = true)
    private List<Comment> comments = new ArrayList<>();

    public void setName(String name) {
        this.name = name;
    }

    public List<Comment> getComments() {
        return comments;
    }

    public void addComment(Comment comment) {
        comments.add(comment);
        comment.setPost(this);
    }

    public void removeComment(Comment comment) {
        comment.setPost(null);
        this.comments.remove(comment);
    }
}

@Entity
public class Comment {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @ManyToOne
    private Post post;

    private String review;

    public void setPost(Post post) {
        this.post = post;
    }

    public String getReview() {
        return review;
    }

    public void setReview(String review) {
        this.review = review;
    }
}
```

Like in the *one-to-one* example, the *CascadeType.ALL* and orphan removal are suitable because the *Comment* life-cycle is bound to that of its *Post Parent* entity.

Cascading the *one-to-many* persist operation

We only have to persist the *Post* entity and all the associated *Comment* entities are persisted as well:

```
Post post = new Post();
post.setName("Hibernate Master Class");

Comment comment1 = new Comment();
comment1.setReview("Good post!");
Comment comment2 = new Comment();
comment2.setReview("Nice post!");

post.addComment(comment1);
post.addComment(comment2);

session.persist(post);
```

The persist operation generates the following output:

```
insert into Post (id, name)
values (default, 'Hibernate Master Class')

insert into Comment (id, post_id, review)
values (default, 1, 'Good post!')

insert into Comment (id, post_id, review)
values (default, 1, 'Nice post!')
```

Cascading the *one-to-many* merge operation

Merging the *Post* entity is going to merge all *Comment* entities as well:

```
Post post = newPost();
post.setName("Hibernate Master Class Training Material");

post.getComments()
    .stream()
    .filter(comment -> comment.getReview().toLowerCase()
         .contains("nice"))
    .findAny()
    .ifPresent(comment ->
        comment.setReview("Keep up the good work!")
);

doInTransaction(session -> {
    session.merge(post);
});
```

Generating the following output:

```
SELECT onetomanyc0_.id    AS id1_1_1_,
       onetomanyc0_.NAME  AS name2_1_1_,
       comments1_.post_id AS post_id3_1_3_,
       comments1_.id      AS id1_0_3_,
       comments1_.id      AS id1_0_0_,
       comments1_.post_id AS post_id3_0_0_,
       comments1_.review  AS review2_0_0_
FROM   post onetomanyc0_
LEFT OUTER JOIN comment comments1_
    ON onetomanyc0_.id = comments1_.post_id
WHERE  onetomanyc0_.id = 1

update Post set
    name = 'Hibernate Master Class Training Material'
where id = 1

update Comment set
    post_id = 1,
    review='Keep up the good work!'
where id = 2
```

Cascading the *one-to-many* delete operation

When the *Post* entity is deleted, the associated *Comment* entities are deleted as well:

```
Post post = newPost();

doInTransaction(session -> {
    session.delete(post);
});
```

Generating the following output:

```
delete from Comment where id = 1
delete from Comment where id = 2
delete from Post where id = 1
```

The *one-to-many* delete orphan cascading operation

The *orphan-removal* allows us to remove the *Child* entity whenever it's no longer referenced by its *Parent*:

```
newPost();

doInTransaction(session -> {
    Post post = (Post) session.createQuery(
        "select p " +
                "from Post p " +
                "join fetch p.comments " +
                "where p.id = :id")
        .setParameter("id", 1L)
        .uniqueResult();
    post.removeComment(post.getComments().get(0));
});
```

The Comment is deleted, as we can see in the following output:

```
SELECT onetomanyc0_.id    AS id1_1_0_,
       comments1_.id      AS id1_0_1_,
       onetomanyc0_.NAME  AS name2_1_0_,
       comments1_.post_id AS post_id3_0_1_,
       comments1_.review  AS review2_0_1_,
       comments1_.post_id AS post_id3_1_0__,
       comments1_.id      AS id1_0_0__
FROM   post onetomanyc0_
INNER JOIN comment comments1_
    ON onetomanyc0_.id = comments1_.post_id
WHERE  onetomanyc0_.id = 1

delete from Comment where id = 1
```

Many-To-Many

The *many-to-many* relationship is tricky because each side of this association plays both the *Parent* and the *Child* role. Still, we can identify one side from where we'd like to propagate the entity state changes.

We shouldn't default to *CascadeType.ALL* because the *CascadeType.REMOVE* might end-up deleting more than we're expecting (as you'll soon find out):

```
@Entity
public class Author {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name = "full_name", nullable = false)
    private String fullName;

    @ManyToMany(mappedBy = "authors",
        cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    private List<Book> books = new ArrayList<>();

    private Author() {}

    public Author(String fullName) {
        this.fullName = fullName;
    }

    public Long getId() {
        return id;
    }

    public void addBook(Book book) {
        books.add(book);
        book.authors.add(this);
    }

    public void removeBook(Book book) {
        books.remove(book);
        book.getAuthors().remove(this);
    }

    public void remove() {
        for(Book book : new ArrayList<>(books)) {
            removeBook(book);
        }
    }
}

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name = "title", nullable = false)
    private String title;

    @ManyToMany(cascade =
        {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable(name = "Book_Author",
        joinColumns = {
            @JoinColumn(
                name = "book_id",
                referencedColumnName = "id"
            )
        },
        inverseJoinColumns = {
            @JoinColumn(
                name = "author_id",
                referencedColumnName = "id"
            )
        }
    )
    private List<Author> authors = new ArrayList<>();

    private Book() {}

    public Book(String title) {
        this.title = title;
    }

    public List<Author> getAuthors() {
        retrun authors;
    }
}
```

Cascading the *many-to-many* persist operation

Persisting the *Author* entities will persist the *Books* as well:

```
Author _John_Smith = new Author("John Smith");
Author _Michelle_Diangello =
    new Author("Michelle Diangello");
Author _Mark_Armstrong =
    new Author("Mark Armstrong");

Book _Day_Dreaming = new Book("Day Dreaming");
Book _Day_Dreaming_2nd =
    new Book("Day Dreaming, Second Edition");

_John_Smith.addBook(_Day_Dreaming);
_Michelle_Diangello.addBook(_Day_Dreaming);

_John_Smith.addBook(_Day_Dreaming_2nd);
_Michelle_Diangello.addBook(_Day_Dreaming_2nd);
_Mark_Armstrong.addBook(_Day_Dreaming_2nd);

session.persist(_John_Smith);
session.persist(_Michelle_Diangello);
session.persist(_Mark_Armstrong);
```

The *Book* and the *Book_Author* rows are inserted along with the *Authors*:

```
insert into Author (id, full_name)
values (default, 'John Smith')

insert into Book (id, title)
values (default, 'Day Dreaming')

insert into Author (id, full_name)
values (default, 'Michelle Diangello')

insert into Book (id, title)
values (default, 'Day Dreaming, Second Edition')

insert into Author (id, full_name)
values (default, 'Mark Armstrong')

insert into Book_Author (book_id, author_id) values (1, 1)
insert into Book_Author (book_id, author_id) values (1, 2)
insert into Book_Author (book_id, author_id) values (2, 1)
insert into Book_Author (book_id, author_id) values (2, 2)
insert into Book_Author (book_id, author_id) values (2, 3)
```

Dissociating one side of the *many-to-many* association

To delete an *Author*, we need to dissociate all *Book_Author* relations belonging to the removable entity:

```
doInTransaction(session -> {
    Author _Mark_Armstrong =
        getByName(session, "Mark Armstrong");
    _Mark_Armstrong.remove();
    session.delete(_Mark_Armstrong);
});
```

This use case generates the following output:

```
SELECT manytomany0_.id        AS id1_0_0_,
       manytomany2_.id        AS id1_1_1_,
       manytomany0_.full_name AS full_nam2_0_0_,
       manytomany2_.title     AS title2_1_1_,
       books1_.author_id      AS author_i2_0_0__,
       books1_.book_id        AS book_id1_2_0__
FROM   author manytomany0_
INNER JOIN book_author books1_
    ON manytomany0_.id = books1_.author_id
INNER JOIN book manytomany2_
    ON books1_.book_id = manytomany2_.id
WHERE  manytomany0_.full_name = 'Mark Armstrong'

SELECT books0_.author_id  AS author_i2_0_0_,
       books0_.book_id    AS book_id1_2_0_,
       manytomany1_.id    AS id1_1_1_,
       manytomany1_.title AS title2_1_1_
FROM   book_author books0_
INNER JOIN book manytomany1_
    ON books0_.book_id = manytomany1_.id
WHERE  books0_.author_id = 2

delete from Book_Author where book_id = 2

insert into Book_Author (book_id, author_id) values (2, 1)
insert into Book_Author (book_id, author_id) values (2, 2)

delete from Author where id = 3
```

The *many-to-many* association generates way too many redundant *SQL* statements and often, they are very difficult to tune. Next, I'm going to demonstrate the *many-to-many CascadeType.REMOVE* hidden dangers.

The *many-to-many CascadeType.REMOVE* gotchas

The *many-to-many CascadeType.ALL* is another code smell, I often bump into while reviewing code. The *CascadeType.REMOVE* is automatically inherited when using *CascadeType.ALL*, but the entity removal is not only applied to the link table, but to the other side of the association as well.

Let's change the *Author* entity *books many-to-many* association to use the *CascadeType.ALL* instead:

```
@ManyToMany(mappedBy = "authors",
    cascade = CascadeType.ALL)
private List<Book> books = new ArrayList<>();
```

When deleting one *Author*:

```
doInTransaction(session -> {
    Author _Mark_Armstrong =
        getByName(session, "Mark Armstrong");
    session.delete(_Mark_Armstrong);
    Author _John_Smith =
        getByName(session, "John Smith");
    assertEquals(1, _John_Smith.books.size());
});
```

All books belonging to the deleted *Author* are getting deleted, even if other *Authors* we're still associated to the deleted *Books*:

```
SELECT manytomany0_.id        AS id1_0_,
       manytomany0_.full_name AS full_nam2_0_
FROM   author manytomany0_
WHERE  manytomany0_.full_name = 'Mark Armstrong'

SELECT books0_.author_id  AS author_i2_0_0_,
       books0_.book_id    AS book_id1_2_0_,
       manytomany1_.id    AS id1_1_1_,
       manytomany1_.title AS title2_1_1_
FROM   book_author books0_
INNER JOIN book manytomany1_ ON
       books0_.book_id = manytomany1_.id
WHERE  books0_.author_id = 3

delete from Book_Author where book_id=2
delete from Book where id=2
delete from Author where id=3
```

Most often, this behavior doesn't match the business logic expectations, only being discovered upon the first entity removal.

We can push this issue even further, if we set the *CascadeType.ALL* to the *Book* entity side as well:

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(name = "Book_Author",
    joinColumns = {
        @JoinColumn(
            name = "book_id",
            referencedColumnName = "id"
        )
    },
    inverseJoinColumns = {
        @JoinColumn(
            name = "author_id",
            referencedColumnName = "id"
        )
    }
)
```

This time, not only the *Books* are being deleted, but *Authors* are deleted as well:

```
doInTransaction(session -> {
    Author _Mark_Armstrong =
        getByName(session, "Mark Armstrong");
    session.delete(_Mark_Armstrong);
    Author _John_Smith =
        getByName(session, "John Smith");
    assertNull(_John_Smith);
});
```

The *Author* removal triggers the deletion of all associated *Books*, which further triggers the removal of all associated *Authors*. This is a very dangerous operation, resulting in a massive entity deletion that's rarely the expected behavior.

```
SELECT manytomany0_.id        AS id1_0_,
       manytomany0_.full_name AS full_nam2_0_
FROM   author manytomany0_
WHERE  manytomany0_.full_name = 'Mark Armstrong'

SELECT books0_.author_id  AS author_i2_0_0_,
       books0_.book_id    AS book_id1_2_0_,
       manytomany1_.id    AS id1_1_1_,
       manytomany1_.title AS title2_1_1_
FROM   book_author books0_
INNER JOIN book manytomany1_
   ON books0_.book_id = manytomany1_.id
WHERE  books0_.author_id = 3

SELECT authors0_.book_id      AS book_id1_1_0_,
       authors0_.author_id    AS author_i2_2_0_,
       manytomany1_.id        AS id1_0_1_,
       manytomany1_.full_name AS full_nam2_0_1_
FROM   book_author authors0_
INNER JOIN author manytomany1_
   ON authors0_.author_id = manytomany1_.id
WHERE  authors0_.book_id = 2

SELECT books0_.author_id  AS author_i2_0_0_,
       books0_.book_id    AS book_id1_2_0_,
       manytomany1_.id    AS id1_1_1_,
       manytomany1_.title AS title2_1_1_
FROM   book_author books0_
INNER JOIN book manytomany1_
   ON books0_.book_id = manytomany1_.id
WHERE  books0_.author_id = 1

SELECT authors0_.book_id      AS book_id1_1_0_,
       authors0_.author_id    AS author_i2_2_0_,
       manytomany1_.id        AS id1_0_1_,
       manytomany1_.full_name AS full_nam2_0_1_
FROM   book_author authors0_
INNER JOIN author manytomany1_
   ON authors0_.author_id = manytomany1_.id
WHERE  authors0_.book_id = 1

SELECT books0_.author_id  AS author_i2_0_0_,
       books0_.book_id    AS book_id1_2_0_,
       manytomany1_.id    AS id1_1_1_,
       manytomany1_.title AS title2_1_1_
FROM   book_author books0_
INNER JOIN book manytomany1_
   ON books0_.book_id = manytomany1_.id
WHERE  books0_.author_id = 2

delete from Book_Author where book_id=2
delete from Book_Author where book_id=1
delete from Author where id=2
delete from Book where id=1
delete from Author where id=1
delete from Book where id=2
delete from Author where id=3
```

This use case is wrong in so many ways. There are a plethora of unnecessary *SELECT* statements and eventually we end up deleting all Authors and all their Books. That's why *CascadeType.ALL* should raise your eyebrow, whenever you spot it on a *many-to-many* association.

When it comes to *Hibernate* mappings, you should always strive for simplicity. The Hibernate documentation (http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch26.html) confirms this assumption as well:

> Practical test cases for real *many-to-many* associations are rare. Most of the time you need additional information stored in the *link table*. In this case, it is much better to use two *one-to-many* associations to an intermediate link class. In fact, most associations are *one-to-many* and *many-to-one*. For this reason, you should proceed cautiously when using any other association style.