1. Overview (/)

In this tutorial, we'll discuss what cascading is in JPA/Hibernate. Then we'll cover the various cascade types that are available, along with their semantics.

Further reading:

Introduction to Spring Data JPA (/the-persistence-layer-with-spring-data-jpa)

Introduction to Spring Data JPA with Spring 4 - the Spring config, the DAO, manual and generated queries and transaction management.
Read more (/the-persistence-layer-with-spring-data-jpa) →

Mapping Entity Class Names to SQL Table Names with JPA (/jpa-entity-table-names)

Learn how table names are generated by default and how to override that behavior.
Read more (/jpa-entity-table-names) →

2. What Is Cascading?

Entity relationships often depend on the existence of another entity, for example the *Person–Address* relationship. Without the *Person*, the *Address* entity doesn't have any meaning of its own. When we delete the *Person* entity, our *Address* entity should also get deleted.

Cascading is the way to achieve this. **When we perform some action on the target entity, the same action will be applied to the associated entity.**

2.1. JPA Cascade Type

All JPA-specific cascade operations are represented by the *jakarta.persistence.CascadeType* enum containing entries:

- *ALL*
- *PERSIST*
- *MERGE*
- *REMOVE*
- *REFRESH*
- *DETACH*

2.2. Hibernate Cascade Type

Hibernate supports three additional Cascade Types along with those specified by JPA. These Hibernate-specific Cascade Types are available in *org.hibernate.annotations.CascadeType*:

- *REPLICATE*
- *SAVE_UPDATE*
- *LOCK*

3. Difference Between the Cascade Types

3.1. *CascadeType.ALL*

CascadeType.ALL **propagates all operations — including Hibernate-specific ones — from a parent to a child entity.**

Let's see it in an example:

```
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;
    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Address> addresses;
}
```

Note that in *OneToMany* associations, we've mentioned cascade type in the annotation.

Now let's see the associated entity *Address*:

```
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String street;
    private int houseNumber;
    private String city;
    private int zipCode;
    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

3.2. *CascadeType.PERSIST*

The persist operation makes a transient instance persistent. **Cascade Type *PERSIST* propagates the persist operation from a parent to a child entity.** When we save the *person* entity, the *address* entity will also get saved.

Let's see the test case for a persist operation:

```
@Test                                (/)
public void whenParentSavedThenChildSaved() {
    Person person = new Person();
    Address address = new Address();
    address.setPerson(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    session.clear();
}
```

When we run the above test case, we'll see the following SQL:

```
Hibernate: insert into Person (name, id) values (?, ?)
Hibernate: insert into Address (
    city, houseNumber, person_id, street, zipCode, id) values (?, ?, ?,
?, ?, ?)
```

3.3. *CascadeType.MERGE*

The merge operation copies the state of the given object onto the persistent object with the same identifier. ***CascadeType.MERGE* propagates the merge operation from a parent to a child entity.**

Let's test the merge operation:

```
@Test
public void whenParentSavedThenMerged() {
    int addressId;
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    addressId = address.getId();
    session.clear();

    Address savedAddressEntity = session.find(Address.class, addressId);
    Person savedPersonEntity = savedAddressEntity.getPerson();
    savedPersonEntity.setName("devender kumar");
    savedAddressEntity.setHouseNumber(24);
    session.merge(savedPersonEntity);
    session.flush();
}
```

When we run the test case, the merge operation generates the following SQL:

```
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.houseNumber as houseNum3_0_0_, address0_.person_id as
person_i6_0_0_, address0_.street as street4_0_0_, address0_.zipCode as
zipCode5_0_0_ from Address address0_ where address0_.id=?
Hibernate: select person0_.id as id1_1_0_, person0_.name as name2_1_0_
from Person person0_ where person0_.id=?
Hibernate: update Address set city=?, houseNumber=?, person_id=?,
street=?, zipCode=? where id=?
Hibernate: update Person set name=? where id=?
```

Here, we can see that the merge operation first loads both *address* and *person* entities and then updates both as a result of *CascadeType.MERGE*.

3.4. *CascadeType.REMOVE*

As the name suggests, the remove operation removes the row corresponding to the entity from the database and also from the persistent context.

***CascadeType.REMOVE* propagates the remove operation from parent to child entity. Similar to JPA's *CascadeType.REMOVE*, we have *CascadeType.DELETE*, which is specific to Hibernate.** There is no difference

between the two.

Now it's time to test *CascadeType.Remove*:

```
@Test
public void whenParentRemovedThenChildRemoved() {
    int personId;
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    personId = person.getId();
    session.clear();

    Person savedPersonEntity = session.find(Person.class, personId);
    session.remove(savedPersonEntity);
    session.flush();
}
```

When we run the test case, we'll see the following SQL:

```
Hibernate: delete from Address where id=?
Hibernate: delete from Person where id=?
```

The *address* associated with the *person* also got removed as a result of *CascadeType.REMOVE*.

3.5. *CascadeType.DETACH*

The detach operation removes the entity from the persistent context. **When we use *CascadeType.DETACH*, the child entity will also get removed from the persistent context.**

Let's see it in action:

```
@Test                                    (/)
public void whenParentDetachedThenChildDetached() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();

    assertThat(session.contains(person)).isTrue();
    assertThat(session.contains(address)).isTrue();

    session.detach(person);
    assertThat(session.contains(person)).isFalse();
    assertThat(session.contains(address)).isFalse();
}
```

Here, we can see that after detaching *person*, neither *person* nor *address* exists in the persistent context.

3.6. *CascadeType.LOCK*

Unintuitively, *CascadeType.LOCK* reattaches the entity and its associated child entity with the persistent context again.

Let's see the test case to understand *CascadeType.LOCK*:

```
@Test
public void whenDetachedAndLockedThenBothReattached() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();

    assertThat(session.contains(person)).isTrue();
    assertThat(session.contains(address)).isTrue();

    session.detach(person);
    assertThat(session.contains(person)).isFalse();
    assertThat(session.contains(address)).isFalse();
    session.unwrap(Session.class)
      .buildLockRequest(new LockOptions(LockMode.NONE))
      .lock(person);

    assertThat(session.contains(person)).isTrue();
    assertThat(session.contains(address)).isTrue();
}
```

As we can see, when using *CascadeType.LOCK*, we attached the entity *person* and its associated *address* back to the persistent context.

3.7. *CascadeType.REFRESH*

Refresh operations **reread the value of a given instance from the database.** In some cases, we may change an instance after persisting in the database, but later we need to undo those changes.

In that kind of scenario, this may be useful. **When we use this operation with Cascade Type *REFRESH*, the child entity also gets reloaded from the database whenever the parent entity is refreshed.**

For better understanding, let's see a test case for *CascadeType.REFRESH*:

```
@Test                                (/)
public void whenParentRefreshedThenChildRefreshed() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    person.setName("Devender Kumar");
    address.setHouseNumber(24);
    session.refresh(person);

    assertThat(person.getName()).isEqualTo("devender");
    assertThat(address.getHouseNumber()).isEqualTo(23);
}
```

Here, we made some changes in the saved entities *person* and *address*. When we refresh the *person* entity, the *address* also gets refreshed.

3.8. *CascadeType.REPLICATE*

The replicate operation is used when we have more than one data source and we want the data in sync. With *CascadeType.REPLICATE*, a sync operation also propagates to child entities whenever performed on the parent entity.

Now let's test *CascadeType.REPLICATE*:

```
@Test
public void whenParentReplicatedThenChildReplicated() {
    Person person = buildPerson("devender");
    person.setId(2);
    Address address = buildAddress(person);
    address.setId(2);
    person.setAddresses(Arrays.asList(address));
    session.unwrap(Session.class).replicate(person,
ReplicationMode.OVERWRITE);
    session.flush();

    assertThat(person.getId()).isEqualTo(2);
    assertThat(address.getId()).isEqualTo(2);
}
```

Because of *CascadeType.REPLICATE*, when we replicate the *person* entity, its associated *address* also gets replicated with the identifier we set.

3.9. *CascadeType.SAVE_UPDATE*

CascadeType.SAVE_UPDATE propagates the same operation to the associated child entity. It's useful when we use **Hibernate-specific operations like *save*, *update* and *saveOrUpdate*.**

Let's see *CascadeType.SAVE_UPDATE* in action:

```
@Test
public void whenParentSavedThenChildSaved() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.saveOrUpdate(person);
    session.flush();
}
```

Because of *CascadeType.SAVE_UPDATE*, when we run the above test case, we can see that the *person* and *address* both got saved.

Here's the resulting SQL:

```
Hibernate: insert into Person (name, id) values (?, ?)
Hibernate: insert into Address (
    city, houseNumber, person_id, street, zipCode, id) values (?, ?, ?,
?, ?, ?)
```

4. Conclusion

In this article, we discussed cascading and the different cascade type options available in JPA and Hibernate.

The source code for the article is available on GitHub (https://github.com/eugenp/tutorials/tree/master/persistence-modules/jpa-hibernate-cascade-type).