SPRINGBOOT ASSIGNMENT

CASCADE:

When we perform some action on the target entity, the same action will be applied to the associated entity. This is achieved with cascade. 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.

JPA(JAVA PERSISTENCE API) CASCADE TYPE:

1 ALL
2 PERSIST
3 MERGE
4 REMOVE
5 REFRESH
6 DETACH

Difference Between the Cascade Types:

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;
}
```

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.

```
@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();
}
```

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();
}
```

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.

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.

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.

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.

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.

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.