# JPA Kickstart Example:

Followings are the key concepts of JPA specifications (JSR 338):

1. JPA is a standard way to access database, to persist Java Objects to database, to retrieve Java Objects from database, taking care of transaction management and to perform other persistence related tasks.
2. A developer works with [javax.persistence](https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html) package. [EntityManager](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html) class is the entry point to use this API.
3. It provides database neutral way to talk to the underlying database via Java Persistence Query Language (JPQL). The query is still string based. We execute JPQL queries using method [EntityManager.createQuery(String)](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#createQuery-java.lang.String-) and overloaded methods [EntityManager.createNamedQuery()](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#createNamedQuery-java.lang.String-).
4. It provides a database neutral and type safe way to talk to the underlying database via Criteria API, defined in [javax.persistence.criteria](https://docs.oracle.com/javaee/7/api/javax/persistence/criteria/package-summary.html) package. Here we work with Java Objects rather than string based queries. The typical entry point to start with the criteria API is [EntityManager.getCriteriaBuilder()](https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html#getCriteriaBuilder--).
5. It provides Object/relational annotations (like [@Entity](https://docs.oracle.com/javaee/7/api/javax/persistence/Entity.html), [@Id](https://docs.oracle.com/javaee/7/api/javax/persistence/Id.html) etc). We use these annotations on our entity objects, so that underlying implementation of JPA will know how to perform [ORM](https://en.wikipedia.org/wiki/Object-relational_mapping) (mapping Java Objects/fields to database tables/columns).

## EntityMangerFactory and persistence-unit

An instance of the interface EntityManagerFactory is used to load a persistence-unit. We can get it via bootstrap class Persistence:

EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPersistenceUnit")

EntityManagerFactory is thread safe so a single instance can be shared by multiple threads.

### What is persistence-unit?

A persistence unit is a logical grouping that contains information like configuration of EntityManagerFactory, a set of entity classes, mapping metadata (can be loaded by scanning mapping annotations or from persistence.xml/orm.xml under META-INF directory). Each persistence-unit must have a unique name. An application can have one or more persistence units.

### persistence.xml

The root of the persistence-unit is defined by a persistence.xml file under META-INF directory in the classpath.

EntityManager : An instance of the interface javax. persistence. Entity Manager is used to persist, retrieve, update and remove entities to/from d db.

## EntityManager and persistence-context

An instance of the interface EntityManager is used to initiate a persistence-context. We can obtain EntityManager instance by calling Entity Manger Factory# create Entity Manager Factory:

EntityManagerFactory emf = Persistence.createEntityManagerFactory("testPersistenceUnit");

EntityManager entityManager = emf.createEntityManager();

### What is persistence-context?

An EntityManager instance is associated with a persistence-context. The persistence-context is a set of managed unique entity instances. EntityManger interacts with this context to manage entity instances and their lifecycle.

EntityManager is not thread-safe, so we should use only one instance per thread.

Each EntityManagerFactory instance provides EntityManager instances that are all configured in the same manner, i.e. all will be using the same persistence-unit. More than one EntityManagerFactory instances can be used per application which would probably be pointing to different set of entities and data-sources etc.

When the application has finished using the entity manager factory, and/or at application shutdown, the application should close the entity manager factory by calling emf.close(). Once an entity manager factory has been closed, all entity managers created from it are considered to be in the closed state as well.

Hib mappings:

*Deleting collection elements one by one can sometimes be extremely inefficient. Hibernate knows not to do that in the case of an newly-empty collection (if you called*list.clear()*, for example). In this case, Hibernate will issue a single*DELETE*.*

*Suppose you added a single element to a collection of size twenty and then remove two elements. Hibernate will issue one*INSERT*statement and two*DELETE*statements, unless the collection is a bag. This is certainly desirable.*

*However, suppose that we remove eighteen elements, leaving two and then add thee new elements. There are two possible ways to proceed*

* *delete eighteen rows one by one and then insert three rows*
* *remove the whole collection in one SQL*DELETE*and insert all five current elements one by one*

*Hibernate cannot know that the second option is probably quicker. It would probably be undesirable for Hibernate to be that intuitive as such behavior might confuse database triggers, etc.*

*Fortunately, you can force this behavior (i.e. the second strategy) at any time by discarding (i.e. dereferencing) the original collection and returning a newly instantiated collection with all the current elements.*

*One-shot-delete does not apply to collections mapped*inverse="true"*.*

The inverse="true" is for (Hibernate Mapping) XML. But in this post, we’ll see how “one-shot delete” works in JPA (with Hibernate as the provider).

We will try different approaches and see which one will result to a one-shot delete.

1. Bi-directional one-to-many
2. Uni-directional one-to-many (with join table)
3. Uni-directional one-to-many (with no join table)
4. Uni-directional one-to-many (using ElementCollection)

We’ll use a Cart entity with many CartItems.

Bi-directional One-to-Many

For this, we have references from *both* sides.

|  |  |
| --- | --- |
| 01 | @Entity |
| 02 | public class Cart { ... |

|  |  |
| --- | --- |
| 03 | @OneToMany(mappedBy="cart", cascade=ALL, orphanRemoval=true) |
| 04 | Collection<OrderItem> items; |

|  |  |
| --- | --- |
| 05 | } |
| 06 |  |

|  |  |
| --- | --- |
| 07 | @Entity |
| 08 | public class CartItem { ... |

|  |  |
| --- | --- |
| 09 | @ManyToOne Cart cart; |
| 10 | } |

To test this, we insert one row to the table for Cart, and three or more rows to the table for CartItem. Then, we run the test.

|  |  |
| --- | --- |
| 01 | public class CartTests { ... |
| 02 | @Test |

|  |  |
| --- | --- |
| 03 | public void testOneShotDelete() throws Exception { |
| 04 | Cart cart = entityManager.find(Cart.class, 53L); |

|  |  |
| --- | --- |
| 05 | for (CartItem item : cart.items) { |
| 06 | item.cart = null; // remove reference to cart |

|  |  |
| --- | --- |
| 07 | } |
| 08 | cart.items.clear(); // as indicated in Hibernate manual |

|  |  |
| --- | --- |
| 09 | entityManager.flush(); // just so SQL commands can be seen |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

The SQL commands shown had each item deleted individually (and not as a one-shot delete).

|  |  |
| --- | --- |
| 1 | delete from CartItem where id=? |
| 2 | delete from CartItem where id=? |

|  |  |
| --- | --- |
| 3 | delete from CartItem where id=? |

Discarding the original collection did not work either. It even caused an exception.

|  |  |
| --- | --- |
| 1 | public class CartTests { ... |
| 2 | @Test |

|  |  |
| --- | --- |
| 3 | public void testOneShotDelete() throws Exception { |
| 4 | Cart cart = entityManager.find(Cart.class, 53L); |

|  |  |
| --- | --- |
| 5 | // remove reference to cart |
| 6 | cart.items = new LinkedList<CartItem>(); // discard, and use new collection |

|  |  |
| --- | --- |
| 7 | entityManager.flush(); // just so SQL commands can be seen |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |
| 1 | javax.persistence.PersistenceException: |

|  |  |
| --- | --- |
| 2 | org.hibernate.HibernateException: |
| 3 | A collection with cascade="all-delete-orphan" was no longer referenced by the owning entity instance: ….Cart.items |

I tested this with Hibernate 4.3.11 and HSQL 2.3.2. If your results vary, please hit the [comments](https://www.javacodegeeks.com/2016/07/one-shot-delete-hibernate-jpa.html?utm_content=buffere0c70&utm_medium=social&utm_source=facebook.com&utm_campaign=buffer#comments).

Uni-directional One-to-Many (With Join Table)

For this, we make changes to the mapping. This causes a join table to be created.

|  |  |
| --- | --- |
| 01 | @Entity |
| 02 | public class Cart { ... |

|  |  |
| --- | --- |
| 03 | @OneToMany(cascade=ALL) |
| 04 | Collection<OrderItem> items; |

|  |  |
| --- | --- |
| 05 | } |
| 06 |  |

|  |  |
| --- | --- |
| 07 | @Entity |
| 08 | public class CartItem { ... |

|  |  |
| --- | --- |
| 09 | // no @ManyToOne Cart cart; |
| 10 | } |

Again, we insert one row to the table for Cart, and three or more rows to the table for CartItem. We also have to insert appropriate records to the join table (Cart\_CartItem). Then, we run the test.

|  |  |
| --- | --- |
| 1 | public class CartTests { ... |
| 2 | @Test |

|  |  |
| --- | --- |
| 3 | public void testOneShotDelete() throws Exception { |
| 4 | Cart cart = entityManager.find(Cart.class, 53L); |

|  |  |
| --- | --- |
| 5 | cart.items.clear(); // as indicated in Hibernate manual |
| 6 | entityManager.flush(); // just so SQL commands can be seen |

|  |  |
| --- | --- |
| 7 | } |
| 8 | } |

The SQL commands shown had the associated rows in the join table deleted (with one command). But the rows in the table for CartItem still exist (and did not get deleted).

|  |  |
| --- | --- |
| 1 | delete from Cart\_CartItem where cart\_id=? |
| 2 | // no delete commands for CartItem |

Hmmm, not exactly what we want, since the rows in the table for CartItem still exist.

Uni-directional One-to-Many (No Join Table)

Starting with JPA 2.0, the join table can be avoided in a uni-directional one-to-many by specifying a @JoinColumn.

|  |  |
| --- | --- |
| 01 | @Entity |
| 02 | public class Cart { ... |

|  |  |
| --- | --- |
| 03 | @OneToMany(cascade=CascadeType.ALL, orphanRemoval=true) |
| 04 | @JoinColumn(name="cart\_id", updatable=false, nullable=false) |

|  |  |
| --- | --- |
| 05 | Collection<OrderItem> items; |
| 06 | } |

|  |  |
| --- | --- |
| 07 |  |
| 08 | @Entity |

|  |  |
| --- | --- |
| 09 | public class CartItem { ... |
| 10 | // no @ManyToOne Cart cart; |

|  |  |
| --- | --- |
| 11 | } |

Again, we insert one row to the table for Cart, and three or more rows to the table for CartItem. Then, we run the test.

|  |  |
| --- | --- |
| 1 | public class CartTests { ... |
| 2 | @Test |

|  |  |
| --- | --- |
| 3 | public void testOneShotDelete() throws Exception { |
| 4 | Cart cart = entityManager.find(Cart.class, 53L); |

|  |  |
| --- | --- |
| 5 | cart.items.clear(); // as indicated in Hibernate manual |
| 6 | entityManager.flush(); // just so SQL commands can be seen |

|  |  |
| --- | --- |
| 7 | } |
| 8 | } |

Discarding the original collection also did not work either. It also caused the same exception (as with bi-directional one-to-many).

|  |  |
| --- | --- |
| 1 | javax.persistence.PersistenceException: |
| 2 | org.hibernate.HibernateException: |

|  |  |
| --- | --- |
| 3 | A collection with cascade="all-delete-orphan" was no longer referenced by the owning entity instance: ….Cart.items |

Uni-directional One-to-Many (with ElementCollection)

JPA 2.0 introduced @ElementCollection. This allows one-to-many relationships to be established with the many-side being either @Basic or @Embeddable (i.e. not an @Entity).

|  |  |
| --- | --- |
| 01 | @Entity |
| 02 | public class Cart { ... |

|  |  |
| --- | --- |
| 03 | @ElementCollection // @OneToMany for basic and embeddables |
| 04 | @CollectionTable(name="CartItem") // defaults to "Cart\_items" if not overridden |

|  |  |
| --- | --- |
| 05 | Collection<OrderItem> items; |
| 06 | } |

|  |  |
| --- | --- |
| 07 |  |
| 08 | @Embeddable // not an entity! |

|  |  |
| --- | --- |
| 09 | public class CartItem { |
| 10 | // no @Id |

|  |  |
| --- | --- |
| 11 | // no @ManyToOne Cart cart; |
| 12 | private String data; // just so that there are columns we can set |

|  |  |
| --- | --- |
| 13 | } |

Again, we insert one row to the table for Cart, and three or more rows to the table for CartItem. Then, we run the test.

|  |  |
| --- | --- |
| 1 | public class CartTests { ... |
| 2 | @Test |

|  |  |
| --- | --- |
| 3 | public void testOneShotDelete() throws Exception { |
| 4 | Cart cart = entityManager.find(Cart.class, 53L); |

|  |  |
| --- | --- |
| 5 | cart.items.clear(); // as indicated in Hibernate manual |
| 6 | entityManager.flush(); // just so SQL commands can be seen |

|  |  |
| --- | --- |
| 7 | } |
| 8 | } |

Yey! The associated rows for CartItem were deleted in one shot.

|  |  |
| --- | --- |
| 1 | delete from CartItem where Cart\_id=? |

Closing Thoughts

One-shot delete occurs with uni-directional one-to-many using ElementCollection (where the many-side is an embeddabled, and *not* an entity).

In the uni-directional one-to-many with join table scenario, deleting entries in a join table doesn’t add much value.

I’m not sure why one-shot delete works (or why it works this way) in Hibernate. But I do have a guess. And that is the underlying JPA provider could not do a one-shot delete because it could not ensure that the many-side entity is not referenced by other entities. Unlike the ElementCollection, the many-side is not an entity and cannot be referenced by other entities.

Now, this *does not* mean that you have to use ElementCollection all the time. Perhaps the one-shot delete only applies to aggregate roots. In those cases, using Embeddable and ElementCollection might be appropriate for a collection of value objects that make up an aggregate. When the aggregate root is removed, then it would be good to see that the “child” objects should be removed as well (and in an efficient manner).

I wish there was a way in JPA to indicate that the child entities are privately owned and can be safely removed when the parent entity is removed (e.g. similar to @PrivateOwned in EclipseLink). Let’s see if it will be included in a future version of the API.