## Ultimate Guide – Association Mappings with JPA and Hibernate

Association mappings are one of the key features of JPA and Hibernate. They model the relationship between two database tables as attributes in your domain model. That allows you to easily navigate the associations in your domain model and JPQL or Criteria queries.

JPA and Hibernate support the same associations as you know from your relational database model. You can use:

- one-to-one associations,
- many-to-one associations and
- many-to-many associations.

You can map each of them as a uni- or bidirectional association. That means you can either model them as an attribute on only one of the associated entities or on both. That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and JPQL or Criteria queries. I will explain that in more details in the first example.

### Many-to-One Associations

An order consists of multiple items, but each item belongs to only one order. That is a typical example for a **many-to-one association**. If you want to model this in your database model, **you need to store the primary key of the Order record as a foreign key in the OrderItem table.**

**With JPA and Hibernate, you can model this in 3 different ways**. You can either model it as a bidirectional association with an attribute on the Order and the OrderItem entity. Or you can model it as a unidirectional relationship with an attribute on the Order or the OrderItem entity.

## Unidirectional Many-to-One Association

Let's take a look at the unidirectional mapping on the OrderItem entity first. The OrderItem entity represents the many side of the relationship and the OrderItem table contains the foreign key of the record in the Order table.

As you can see in the following code snippet, you can model this association with an attribute of type Order and a @ManyToOne annotation. The Order order attribute models the association, and the annotation tells Hibernate how to map it to the database.

```
@Entity
public class OrderItem {

    @ManyToOne
    private Order order;
    …
}
```

That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column with the name order_id to store the foreign key to the Order entity.

**If you want to use a different column, you need to define the foreign key column name with a @JoinColumn annotation**. The example in the following code snippet tells Hibernate to use the column **fk_order** to store the foreign key.

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = "fk_order")
    private Order order;
    …
}
```

You can now use this association in your business code to get the Order for a given OrderItem and to add or remove an OrderItem to or from an existing Order.

```
Order o = em.find(Order.class, 1L);
OrderItem i = new OrderItem();
i.setOrder(o);
em.persist(i);
```

That's all about the mapping of unidirectional many-to-one associations for now. If you want to dive deeper, you should take a look at FetchTypes. I explained them in detail in my Introduction to JPA FetchTypes**(https://thoughts-on-java.org/entity-mappings-introduction-jpa-fetchtypes/).**

But now, let's continue with the association mappings and talk about unidirectional one-to-many relationships next. As you might expect, the mapping is very similar to this one.

### Unidirectional One-to-Many Association

The unidirectional one-to-many relationship mapping is not very common. In the example of this post, it only models the association on the Order entity and not on the OrderItem.

The basic mapping definition is very similar to the many-to-one association. It consist of the List items attribute which stores the associated entities and a @OneToMany association.

```
@Entity
public class Order {

    @OneToMany
    private List<OrderItem> items = new ArrayList<OrderItem>();
     …
}
```

But this is most likely not the mapping you're looking for because Hibernate uses an association table to map the relationship. If you want to avoid that, you need to use a @JoinColumn annotation to specify the foreign key column.

The following code snippet shows an example of such a mapping. The @JoinColumn annotation tells Hibernate to use the fk_order column in the OrderItem table to join the two database tables.

```
@Entity
public class Order {

   @OneToMany
   @JoinColumn(name = "fk_order")
   private List<OrderItem> items = new ArrayList<OrderItem>();
    …
}
```

You can now use the association in your business code to get all OrderItems of an Order and to add or remove an OrderItem to or from an Order.

```
Order o = em.find(Order.class, 1L);
OrderItem i = new OrderItem();
o.getItems().add(i);
em.persist(i);
```

**Bidirectional Many-to-One Associations**

The bidirectional Many-to-One association mapping is the most common way to model this relationship with JPA and Hibernate. It uses an attribute on the Order and the OrderItem entity. This allows you to navigate the association in both directions in your domain model and your JPQL queries.

The mapping definition consists of 2 parts:

- the to-many side of the association which owns the relationship mapping and
- the to-one side which just references the mapping

Let's take a look at the owning side first. You already know this mapping from the unidirectional Many-to-One association mapping. It consists of the Order order attribute, a @ManyToOne annotation and an optional @JoinColumn annotation.

```
@Entity
public class OrderItem {

   @ManyToOne
   @JoinColumn(name = "fk_order")
   private Order order;

   …
}
```

The owning part of the association mapping already provides all the information Hibernate needs to map it to the database. That makes the definition of the referencing part simple. You just need to reference the owning association mapping. You can do that by providing the name of the association-mapping attribute to the mappedBy attribute of the @OneToMany annotation. In this example, that's the order attribute of the OrderItem entity.

```
@Entity
public class Order {
```

```
    @OneToMany(mappedBy = "order")
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

You can now use this association in a similar way as the unidirectional relationships I showed you before. But adding and removing an entity from the relationship requires an additional step. You need to update both sides of the association.

```
Order o = em.find(Order.class, 1L);
OrderItem i = new OrderItem();
i.setOrder(o);
o.getItems().add(i);
em.persist(i);
```

That is an error-prone task, and a lot of developers prefer to implement it in a utility method which updates both entities.

```
@Entity
public class Order {
    …
    public void addItem(OrderItem item) {
        this.items.add(item);
        item.setOrder(this);
    }
    …
}
```

That's all about many-to-one association mapping for now. You should also take a look at FetchTypes and how they impact the way Hibernate loads entities from the database. I get into detail about them in my Introduction to JPA FetchTypes.

## Many-to-Many Associations

Many-to-Many relationships are another often used association type. On the database level, it requires an additional association table which contains the primary key pairs of the associated entities. But as you will see, you don't need to map this table to an entity.

A typical example for such a many-to-many association are Products and Stores. Each Store sells multiple Products and each Product gets sold in multiple Stores.

Similar to the many-to-one association, you can model a many-to-many relationship as a uni- or bidirectional relationship between two entities.

But there is an important difference that might not be obvious when you look at the following code snippets. When you map a many-to-many association, you should use a Set instead of a List as the attribute type. Otherwise, Hibernate will take a very inefficient approach to remove entities from the association. It will remove all records from the association table and re-insert the remaining ones. You can

avoid that by using a Set instead of a List as the attribute type.

OK let's take a look at the unidirectional mapping first.

## Unidirectional Many-to-Many Associations

Similar to the previously discussed mappings, the unidirectional many-to-many relationship mapping requires an entity attribute and a **@ManyToMany** annotation. The attribute models the association and you can use it to navigate it in your domain model or JPQL queries. The annotation tells Hibernate to map a **many-to-many** association.

Let's take a look at the relationship mapping between a Store and a Product. The Set products attribute models the association in the domain model and the **@ManyToMany** association tells Hibernate to map it as a many-to-many association.

And as I already explained, please note the difference to the previous many-to-one mappings. You should map the associated entities to a Set instead of a List.

```
@Entity
public class Store {

   @ManyToMany
   private Set<Product> products = new HashSet<Product>();

   …
}
```

If you don't provide any additional information, Hibernate uses its default mapping which expects an association table with the name of both entities and the primary key attributes of both entities. In this case, Hibernate uses the **Store_Product** table with the columns **store_id** and **product_id.**

You can customize that with a **@JoinTable** annotation and its attributes **joinColumns** and **inverse-JoinColumns**. The **joinColumns** attribute defines the **foreign key** columns for the entity on which you define the association mapping. The **inverseJoinColumns** attribute specifies the foreign key columns of the associated entity.

The following code snippet shows a mapping that tells Hibernate to use the **store_product** table with the **fk_product** column as the foreign key to the Product table and the **fk_store** column as the foreign key to the Store table.

```
@Entity
public class Store {

   @ManyToMany
   @JoinTable(name = "store_product",
         joinColumns = { @JoinColumn(name = "fk_store") },
         inverseJoinColumns = { @JoinColumn(name = "fk_product") })
   private Set<Product> products = new HashSet<Product>();

   …
```

```
}
```

That's all you have to do to define an unidirectional many-to-many association between two entities. You can now use it to get a Set of associated entities in your domain model or to join the mapped tables in a JPQL query.

```
Store s = em.find(Store.class, 1L);
Product p = new Product();
s.getProducts().add(p);
em.persist(p);
```

## Bidirectional Many-to-Many Associations

The bidirectional relationship mapping allows you to navigate the association in both directions. And after you've read the post this far, you're probably not surprised when I tell you that the mapping follows the same concept as the bidirectional mapping of a many-to-one relationship.

One of the two entities owns the association and provides all mapping information. The other entity just references the association mapping so that Hibernate knows where it can get the required information.

Let's start with the entity that owns the relationship. The mapping is identical to the unidirectional many-to-many association mapping. You need an attribute that maps the association in your domain model and a @ManyToMany association. If you want to adapt the default mapping, you can do that with a @JoinColumn annotation.

```
@Entity
public class Store {

   @ManyToMany
   @JoinTable(name = "store_product",
         joinColumns = { @JoinColumn(name = "fk_store") },
         inverseJoinColumns = { @JoinColumn(name = "fk_product") })
   private Set<Product> products = new HashSet<Product>();

   …
}
```

The mapping for the referencing side of the relationship is a lot easier. Similar to the bidirectional many-to-one relationship mapping, you just need to reference the attribute that owns the association.

You can see an example of such a mapping in the following code snippet. The List products attribute of the Store entity owns the association. So, you only need to provide the String "products" to the mappedBy attribute of the @ManyToMany annotation.

```
@Entity
public class Product{
    @ManyToMany(mappedBy="products")
   private Set<Store> stores = new HashSet<Store>();
```

```
    …
}
```

That's all you need to do to define a bidirectional many-to-many association between two entities. But there is another thing you should do to make it easier to use the bidirectional relationship.

You need to update both ends of a bidirectional association when you want to add or remove an entity. Doing that in your business code is verbose and error-prone. It's, therefore, a good practice to provide helper methods which update the associated entities.

```
@Entity
public class Store {

    public void addProduct(Product p) {
        this.products.add(p);
        p.getStores().add(this);
    }

    public void removeProduct(Product p) {
        this.products.remove(p);
        p.getStores().remove(this);
    }
    …
}
```

OK, now we're done with the definition of the many-to-many association mappings. Let's take a look at the third and final kind of association: The one-to-one relationship.

## One-to-One Associations

One-to-one relationships are rarely used in relational table models. You, therefore, won't need this mapping too often. But you will run into it from time to time. So it's good to know that you can map it in a similar way as all the other associations.

An example for a one-to-one association could be a Customer and the ShippingAddress. Each Customer has exactly one ShippingAddress and each ShippingAddress belongs to one Customer. On the database level, this mapped by a foreign key column either on the ShippingAddress or the Customer table.

Let's take a look at the unidirectional mapping first.

## Unidirectional One-to-One Associations

As in the previous unidirectional mapping, you only need to model it for the entity for which you want to navigate the relationship in your query or domain model. Let's say you want to get from the Customer to the ShippingAddress entity.

The required mapping is similar to the previously discussed mappings. You need an entity attribute that represents the association, and you have to annotate it with an @OneToOne annotation.

When you do that, Hibernate uses the name of the associated entity and the name of its primary key attribute to generate the name of the foreign key column. In this example, it would use the column shippingaddress_id. You can customize the name of the foreign key column with a @JoinColumn annotation. The following code snippet shows an example of such a mapping.

```
@Entity
public class Customer{

   @OneToOne
   @JoinColumn(name = "fk_shippingaddress")
   private ShippingAddress shippingAddress;

   …
}
```

That's all you need to do to define a one-to-one association mapping. You can now use it in your business to add or remove an association, to navigate it in your domain model or to join it in a JPQL query.

```
Customer c = em.find(Customer.class, 1L);
ShippingAddress sa = c.getShippingAddress();
```

## Bidirectional One-to-One Associations

The bidirectional one-to-one relationship mapping extends the unidirectional mapping so that you can also navigate it in the other direction. In this example, you also model it on the ShippingAddress entity so that you can get the Customer for a giving ShippingAddress.

Similar to the previously discussed bidirectional mappings, the bidirectional one-to-one relationship consists of an owning and a referencing side. The owning side of the association defines the mapping, and the referencing one just links to that mapping.

The definition of the owning side of the mapping is identical to the unidirectional mapping. It consists of an attribute that models the relationship and is annotated with a @OneToOne annotation and an optional @JoinColumn annotation.

```
@Entity
public class Customer{

   @OneToOne
   @JoinColumn(name = "fk_shippingaddress")
   private ShippingAddress shippingAddress;

   …
}
```

The referencing side of the association just links to the attribute that owns the relationship. Hibernate gets all information from the referenced mapping, and you don't need to provide any additional information. You can define that with the mappedBy attribute of the @OneToOne annotation. The following code snippet shows an example of such a mapping.

```
@Entity
public class ShippingAddress{

   @OneToOne(mappedBy = “shippingAddress”)
   private Customer customer;
    …
}
```

**Summary**

The relational table model uses many-to-many, many-to-one and one-to-one associations to model the relationship between database records. You can map the same relationships with JPA and Hibernate, and you can do that in an unidirectional or bidirectional way.

The unidirectional mapping defines the relationship only on 1 of the 2 associated entities, and you can only navigate it in that direction. The bidirectional mapping models the relationship for both entities so that you can navigate it in both directions.

The concept for the mapping of all 3 kinds of relationships is the same.

If you want to create an unidirectional mapping, you need an entity attribute that models the association and that is annotated with a *@ManyToMany*, *@ManyToOne*, *@OneToMany* or *@OneToOne* annotation. Hibernate will generate the name of the required foreign key columns and tables based on the name of the entities and their primary key attributes.

The bidirectional associations consist of an owning and a referencing side. The owning side of the association is identical to the unidirectional mapping and defines the mapping. The referencing side only links to the attribute that owns the association.

## Entity Mappings: Introduction to JPA FetchTypes

The FetchType defines when Hibernate gets the related entities from the database, and it is one of the crucial elements for a fast persistence tier. In general, you want to fetch the entities you use in your business tier as efficiently as possible. But that's not that easy. You either get all relationships with one query or you fetch only the root entity and initialize the relationships as soon as you need them.

I'll explain both approaches in more detail during this post and also provide you some links to more advanced solutions that combine flexibility and efficiency.

## Default FetchTypes and how to change it

When you started with Hibernate, you most, likely either didn't know about FetchTypes or you were told to always use FetchType.LAZY. In general, that's a good recommendation. But what does it exactly mean? And what is the default if you don't define the FetchType?

The default depends on the cardinality of the relationship. All to-one relationships use FetchType.EAGER and all to-many relationships FetchType.LAZY.

Even the best default doesn't fit for all use cases, and you sometimes want to change it. You can do this by providing your preferred FetchType to the relationship annotation as you can see in the following code snippet.

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

  @OneToMany(mappedBy = "order", fetch = FetchType.EAGER)
  private Set<OrderItem> items = new HashSet<OrderItem>();
    ...
 }
```

## FetchType.EAGER – Fetch it so you'll have it when you need it

The FetchType.EAGER tells Hibernate to get all elements of a relationship when selecting the root entity. As I explained earlier, this is the default for to-one relationships, and you can see it in the following code snippets.

I use the default FetchType (EAGER) for the many-to-one relationship between the OrderItem and Product entity.

```
@Entity
public class OrderItem implements Serializable {
  @ManyToOne
  private Product product;
  ...
}
```

When I now fetch an OrderItem entity from the database, Hibernate will also get the related Product entity.

```
OrderItem orderItem = em.find(OrderItem.class, 1L);
```

```
log.info("Fetched OrderItem: "+orderItem);
Assert.assertNotNull(orderItem.getProduct());
```

05:01:24,504 DEBUG SQL:92 - select orderitem0_.id as id1_0_0_, orderitem0_.order_id as order_id4_0_0_, orderitem0_.product_id as product_5_0_0_, orderitem0_.quantity as quantity2_0_0_, orderitem0_.version as version3_0_0_, order1_.id as id1_2_1_, order1_.orderNumber as orderNum2_2_1_, order1_.version as version3_2_1_, product2_.id as id1_1_2_, product2_.name as name2_1_2_, product2_.price as price3_1_2_, product2_.version as version4_1_2_ from OrderItem orderitem0_ left outer join purchaseOrder order1_ on orderitem0_.order_id=order1_.id left outer join Product product2_ on orderitem0_.product_id=product2_.id where orderitem0_.id=?
05:01:24,557 INFO FetchTypes:77 - Fetched OrderItem: OrderItem , quantity: 100

This seems to be very useful in the beginning. Joining the required entities and getting all of them in one query is very efficient.

But keep in mind, that Hibernate will ALWAYS fetch the Product entity for your OrderItem, even if you don't use it in your business code. If the related entity isn't too big, this is not an issue for to-one relationships. But it will most likely slow down your application if you use it for a to-many relationship that you don't need for your use case. Hibernate then has to fetch tens or even hundreds of additional entities which creates a significant overhead.

## FetchType.LAZY – Fetch it when you need it

The FetchType.LAZY tells Hibernate to only fetch the related entities from the database when you use the relationship. This is a good idea in general because there's no reason to select entities you don't need for your uses case. You can see an example of a lazily fetched relationship in the following code snippets.

The one-to-many relationship between the Order and the OrderItem entities uses the default FetchType for to-many relationships which is lazy.

```
@Entity
@Table(name = "purchaseOrder")
public class Order implements Serializable {

  @OneToMany(mappedBy = "order")
  private Set<OrderItem> items = new HashSet<OrderItem>();
   ...
}
```

The used FetchType has no influence on the business code. You can call the getOrderItems() method just as any other getter method.

```
Order newOrder = em.find(Order.class, 1L);
log.info("Fetched Order: "+newOrder);
Assert.assertEquals(2, newOrder.getItems().size());
```

Hibernate handles the lazy initialization transparently and fetches the OrderItem entities as soon as the getter method gets called.

05:03:01,504 DEBUG SQL:92 - select order0_.id as id1_2_0_, order0_.orderNumber as orderNum2_2_0_, order0_.version as version3_2_0_ from purchaseOrder order0_ where order0_.id=?
05:03:01,545 INFO FetchTypes:45 - Fetched Order: Order orderNumber: order1
05:03:01,549 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?

Handling lazy relationships in this way is perfectly fine if you work on a single Order entity or a small list of entities. But it becomes a performance problem when you do it on a large list of entities. As you can see in the following log messages, Hibernate has to perform an additional SQL statement for each Order entity to fetch its OrderItems.

05:03:40,936 DEBUG ConcurrentStatisticsImpl:411 - HHH000117: HQL: SELECT o FROM Order o, time: 41ms, rows: 3
05:03:40,939 INFO FetchTypes:60 - Fetched all Orders
05:03:40,942 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?
05:03:40,957 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?
05:03:40,959 DEBUG SQL:92 - select items0_.order_id as order_id4_0_0_, items0_.id as id1_0_0_, items0_.id as id1_0_1_, items0_.order_id as order_id4_0_1_, items0_.product_id as product_5_0_1_, items0_.quantity as quantity2_0_1_, items0_.version as version3_0_1_, product1_.id as id1_1_2_, product1_.name as name2_1_2_, product1_.price as price3_1_2_, product1_.version as version4_1_2_ from OrderItem items0_ left outer join Product product1_ on items0_.product_id=product1_.id where items0_.order_id=?

This behavior is called n+1 select issue, and it's the most common performance problem. It is so common that you most likely have it if you didn't explicitly search for it. If you're not sure how to do that, signup for my free, 3-part video course about finding and fixing n+1 select issues.

There are two ways to avoid these issues:

You can use FetchType.EAGER if you know that all of your use cases that fetch an Order entity also need to process the related OrderItem entities. That will almost never be the case.

If there are some use cases which only work on Order entities (which is most likely the case), you

should use FetchType.LAZY in your entity mapping and use one of these options to initialize the relationship when you need them.

## Summary and cheat sheet

As I said in the beginning, you need to make sure to use the right FetchType for your use case to avoid common Hibernate performance issues. For most use cases, the FetchType.LAZY is a good choice. But make sure that you don't create any n+1 select issues.

Let's quickly summarize the different FetchTypes.

EAGER fetching tells Hibernate to get the related entities with the initial query. This can be very efficient because all entities are fetched with only one query. But in most cases it just creates a huge overhead because you select entities you don't need in your use case.

You can prevent this with FetchType.LAZY. This tells Hibernate to delay the initialization of the relationship until you access it in your business code. The drawback of this approach is that Hibernate needs to execute an additional query to initialize each relationship.

## Hibernate One To Many Mapping Example Annotation

Today we will look into One To Many Mapping in Hibernate. We will look into Hibernate One To Many Mapping example using Annotation and XML configuration.

## One To Many Mapping in Hibernate

In simple terms, one to many mapping means that one row in a table can be mapped to multiple rows in another table. For example, think of a Cart system where we have another table for Items. A cart can have multiple items, so here we have one to many mapping. We will use Cart-Items scenario for our hibernate one to many mapping example.

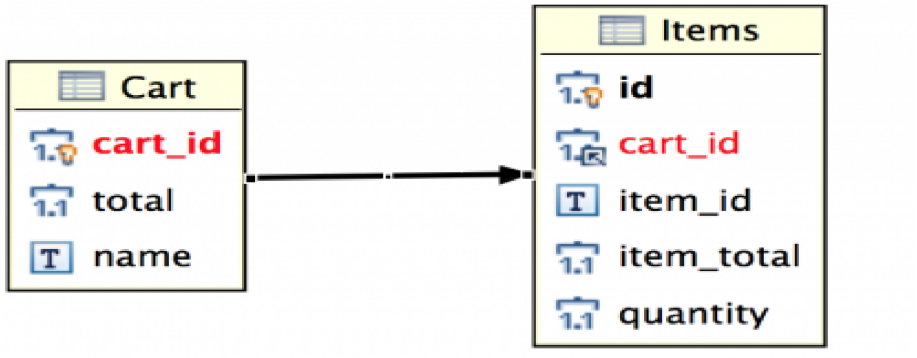
## One To Many Mapping in Hibernate – Database Setup

We can use foreign key constraint for one to many mapping. Below is our database script for Cart and Items table. I am using MySQL database for Hibernate one to many mapping example.

**setup.sql**

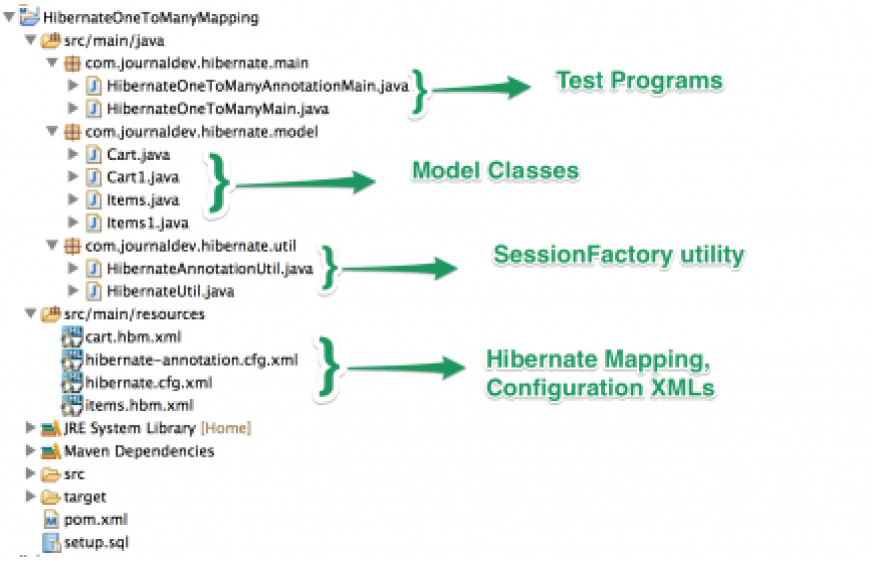
```
CREATE TABLE `Cart` (
  `cart_id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `total` decimal(10,0) NOT NULL,
  `name` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`cart_id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;

CREATE TABLE `Items` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `cart_id` int(11) unsigned NOT NULL,
  `item_id` varchar(10) NOT NULL,
  `item_total` decimal(10,0) NOT NULL,
  `quantity` int(3) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `cart_id` (`cart_id`),
  CONSTRAINT `items_ibfk_1` FOREIGN KEY (`cart_id`) REFERENCES `Cart` (`cart_id`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8;
```

Below is the ER diagram of the Cart and Items table.



Our database setup is ready, let's move on to creating hibernate One to Many Mapping example project. First of all, we will use XML based configuration and then we will implement one to many mapping using Hibernate and JPA annotation.

**Hibernate One To Many Mapping Project Structure**
Create a simple Maven project in Eclipse or you favorite IDE, the final project structure will look like below image.




**Hibernate Maven Dependencies**
Our final pom.xml file contains dependencies for Hibernate and MySQL driver. Hibernate uses JBoss logging and it automatically gets added as transitive dependencies.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.journaldev.hibernate</groupId>
  <artifactId>HibernateOneToManyMapping</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>4.3.5.Final</version>
        </dependency>
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.0.5</version>
        </dependency>
```

```
 </dependencies>

</project>
```

Note that I am using latest Hibernate version 4.3.5.Final and MySQL driver version based on my database installation.

**Hibernate One To Many Mapping Model Classes**
For our tables Cart and Items, we have model classes to reflect them.

Cart.java

```
package com.java.hibernate.model;

import java.util.Set;

public class Cart {
private long id;
private double total;
private String name;
private Set<Items> items;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public String getName() {
return name;
}

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

public Set<Items> getItems() {
return items;
}

public void setItems(Set<Items> items) {
this.items = items;
}
}
```

I am using Set of Items, so that every record is unique. We can also use List or Array for one to many mapping in hibernate.

Items.java

```
package com.java.hibernate.model;

public class Items {
private long id;
private String itemId;
private double itemTotal;
private int quantity;
private Cart cart;

// Hibernate requires no-args constructor
public Items() {
}

public Items(String itemId, double total, int qty, Cart c) {
this.itemId = itemId;
this.itemTotal = total;
this.quantity = qty;
this.cart = c;
}

public String getItemId() {
return itemId;
}

public void setItemId(String itemId) {
this.itemId = itemId;
}

public double getItemTotal() {
return itemTotal;
}

public void setItemTotal(double itemTotal) {
this.itemTotal = itemTotal;
}

public int getQuantity() {
return quantity;
}

public void setQuantity(int quantity) {
this.quantity = quantity;
}

public Cart getCart() {
return cart;
}

public void setCart(Cart cart) {
this.cart = cart;
}

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
```

```
}
}
```

Items have many to one relationship to Cart, so we don't need to have Collection for Cart object.

**Hibernate SessionFactory Utility Class**
We have a utility class for creating Hibernate SessionFactory.

**HibernateUtil.java**

```
package com.java.hibernate.util;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class HibernateUtil {
private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {
try {
// Create the SessionFactory from hibernate.cfg.xml
Configuration configuration = new Configuration();
configuration.configure("/resources/Hibernate-cfg.xml");

System.out.println("Hibernate Configuration loaded");

ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySet-
tings(configuration.getProperties()).build();
System.out.println("Hibernate serviceRegistry created");

SessionFactory sessionFactory = configuration.buildSessionFactory(serviceRegistry);
return sessionFactory;
} catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
}

public static SessionFactory getSessionFactory() {
if(sessionFactory == null) {
sessionFactory = buildSessionFactory();
}

return sessionFactory;
}
}
```

**Hibernate Configuration XML File**
Our hibernate configuration xml file contains database information and mapping resource details.

**hibernate.cfg.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDri-
ver</property>
        <property name="hibernate.connection.password">Admin@123</property>
        <property name="hibernate.connection.url">jdbc:oracle:thin:@local-
host:1521:XE</property>
        <property name="hibernate.connection.username">system</property>
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle9Dia-
lect</property>

        <property name="hibernate.current_session_context_class">thread</property>
        <property name="hibernate.show_sql">true</property>

        <mapping resource="cart.hbm.xml"/>
        <mapping resource="items.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

**Hibernate One To Many Mapping Example – XML Configuration**

This is the most important part of tutorial, let's see how we have to map both Cart and Items classes for one to many mapping in hibernate.

cart.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.hibernate.m2o.model">
<class name="Cart" table="O2M_CART">
<id name="id" type="long">
<column name="cart_id"/>
<generator class="identity"/>
</id>

<property name="total" type="double">
<column name="total"/>
</property>

<property name="name" type="string">
<column name="name"/>
</property>

<set name="items" table="O2M_ITEMS" fetch="select">
<key>
<column name="cart_id" not-null="true"/>
</key>
<one-to-many class="Items"/>
</set>
</class>

</hibernate-mapping>
```

The important part is the set element and one-to-many element inside it. Notice that we are providing key to be used for one to many mapping i.e cart_id.

**items.hbm.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd" >

<hibernate-mapping package="com.hibernate.m2o.model">
<class name="Items" table="O2M_ITEMS">
<id name="id" type="long" column="ID">
<generator class="identity"/>
</id>
<property name="itemId" type="string">
<column name="item_id"></column>
</property>
<property name="itemTotal" type="double">
<column name="item_total"></column>
</property>
<property name="quantity" type="integer">
<column name="quantity"></column>
</property>

<many-to-one name="cart" class="Cart">
<column name="cart_id" not-null="true"></column>
</many-to-one>
</class>
</hibernate-mapping>
```

Notice that from items to cart, it's many to one relationship. So we need to use many-to-one element for cart and we are providing column name that will be mapped with the key. So based on the Cart hibernate mapping configuration, it's key cart_id will be used for mapping.

Our project for Hibernate One To Many Mapping Example using XML mapping is ready, let's write a test program and check if it's working fine or not.

Hibernate One To Many Mapping Example – Test Program

**HibernateOneToManyMain.java**

```
package com.java.hibernate;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.java.hibernate.model.Cart;
import com.java.hibernate.model.Items;
import com.java.hibernate.util.HibernateUtil;

public class HibernateOneToManyMain {
public static void main(String[] args) {
Cart cart = new Cart();
cart.setName("SamCart");

Items item1 = new Items("I3", 30, 3, cart);
Items item2 = new Items("I4", 40, 4, cart);
Set<Items> itemSet = new HashSet<>();
itemSet.add(item1);
itemSet.add(item2);
```

```
cart.setItems(itemSet);
cart.setTotal(10*1 + 20*2);

SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;

try {
// Get Session Factory
sessionFactory = HibernateUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
System.out.println("Session created");
//start transaction
tx = session.beginTransaction();

//Save the Model objects
session.save(cart);
session.save(item1);
session.save(item2);

//Commit transaction
tx.commit();

}catch(Exception e){
System.out.println("Exception occured. "+e.getMessage());
e.printStackTrace();
}finally{
if(!sessionFactory.isClosed()){
System.out.println("Closing SessionFactory");
sessionFactory.close();
}
}
}
}
```

Notice that we need to save both Cart and Items objects one by one. Hibernate will take care of updating the foreign keys in Items table. When we execute above program, we get following output.

**Output**

```
Hibernate Configuration loaded
Hibernate serviceRegistry created
Session created
Hibernate: insert into CART (total, name) values (?, ?)
Hibernate: insert into ITEMS (item_id, item_total, quantity, cart_id) values (?, ?, ?, ?)
Hibernate: insert into ITEMS (item_id, item_total, quantity, cart_id) values (?, ?, ?, ?)
Hibernate: update ITEMS set cart_id=? where id=?
Hibernate: update ITEMS set cart_id=? where id=?
Cart ID=6
Closing SessionFactory
```

Notice that Hibernate is using Update query to set the cart_id in ITEMS table.

## Hibernate One To Many Mapping Annotation

Now that we have seen how to implement One To Many mapping in Hibernate using XML based configurations, let's see how we can do the same thing using JPA annotations.

### Hibernate One To Many Mapping Example Annotation

Hibernate configuration file is almost same, except that mapping element changes because we are using Classes for hibernate one to many mapping using annotation.

**hibernate-annotation.cfg.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="hibernate.connection.driver_class">oracle.jdbc.driver.OracleDri-
ver</property>
        <property name="hibernate.connection.password">Admin@123</property>
        <property name="hibernate.connection.url">jdbc:oracle:thin:@local-
host:1521:XE</property>
        <property name="hibernate.connection.username">system</property>
        <property name="hibernate.dialect">org.hibernate.dialect.Oracle9Dia-
lect</property>

        <property name="hibernate.current_session_context_class">thread</property>
        <!-- <property name="hibernate.show_sql">true</property>
        <property name="hbm2ddl.auto">create</property> -->

        <mapping class="com.java.hibernate.model.Cart"/>
        <mapping class="com.java.hibernate.model.Items"/>
</session-factory>
</hibernate-configuration>
```

### Hibernate SessionFactory Utility Class

SessionFactory utility class is almost same, we just need to use the new hibernate configuration file.

**HibernateAnnotationUtil.java**

```
package com.java.hibernate.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateAnnotationUtil {
private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {
try {
// Create the SessionFactory from hibernate.cfg.xml
Configuration configuration = new Configuration();
configuration.configure("/resources/hibernate-annotation.cfg.xml");

System.out.println("Hibernate Configuration loaded");

// ServiceRegistry serviceRegistry = new
// StandardServiceRegistryBuilder().applySettings(configuration.getProper-
ties()).build();
```

```
System.out.println("Hibernate serviceRegistry created");

SessionFactory sessionFactory = configuration.buildSessionFactory();
return sessionFactory;
} catch (Throwable ex) {
System.err.println("Initial SessionFactory creation failed." + ex);
ex.printStackTrace();
throw new ExceptionInInitializerError(ex);
}
}

public static SessionFactory getSessionFactory() {
if (sessionFactory == null) {
sessionFactory = buildSessionFactory();
}

return sessionFactory;
}
}
```

**Hibernate One To Many Mapping Annotation Model Classes**

Since we don't have xml based mapping files, all the mapping related configurations will be done using JPA annotations in the model classes. If you understand the xml based mapping, it's very simple and similar.

**Cart1.java**

```
package com.java.hibernate.model;

import java.util.Set;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "O2M_CART1")
public class Cart1 {

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "cart_id")
private long id;

@Column(name = "TOTAL")
private double total;

@Column(name = "NAME")
private String name;

@OneToMany(mappedBy = "cart1")
private Set<Items1> items1;

public long getId() {
return id;
```

```
}

public void setId(long id) {
this.id = id;
}

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public String getName() {
return name;
}

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

public Set<Items1> getItems1() {
return items1;
}

public void setItems1(Set<Items1> items1) {
this.items1 = items1;
}
}
```

Important point to note is the OneToMany annotation where mappedBy variable is used to define the property in Items1 class that will be used for the mapping purpose. So we should have a property named “cart1” in Items1 class. Don’t forget to include all the getter-setter methods.

**Items1.java**

```
package com.java.hibernate.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "O2M_ITEMS1")
public class Items1 {

@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "ID")
private long id;

@Column(name = "item_id")
private String itemId;
```

```java
@Column(name = "item_total")
private double itemTotal;

@Column(name = "quantity")
private int quantity;

@ManyToOne
@JoinColumn(name = "cart_id", nullable = false)
private Cart1 cart1;

// Hibernate requires no-args constructor
public Items1() {
}

public Items1(String itemId, double total, int qty, Cart1 c) {
this.itemId = itemId;
this.itemTotal = total;
this.quantity = qty;
this.cart1 = c;
}

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public String getItemId() {
return itemId;
}

public void setItemId(String itemId) {
this.itemId = itemId;
}

public double getItemTotal() {
return itemTotal;
}

public void setItemTotal(double itemTotal) {
this.itemTotal = itemTotal;
}

public int getQuantity() {
return quantity;
}

public void setQuantity(int quantity) {
this.quantity = quantity;
}

public Cart1 getCart1() {
return cart1;
}

public void setCart1(Cart1 cart1) {
```

```
this.cart1 = cart1;
}
}
```

Most important point in above class is the ManyToOne annotation on Cart1 class variable and JoinColumn annotation to provide the column name for mapping.

That's it for one to many mapping in hibernate using annotation in model classes. Compare it with XML based configurations, you will find them very similar.

Let's write a test program and execute it.

## Hibernate One To Many Mapping Annotation Example Test Program

Our test program is just like xml based configuration, we are just using the new classes for getting Hibernate Session and saving the model objects into database.

### HibernateOneToManyAnnotationMain.java

```
package com.java.hibernate;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.java.hibernate.model.Cart1;
import com.java.hibernate.model.Items1;
import com.java.hibernate.util.HibernateAnnotationUtil;

public class HibernateOneToManyAnnotationMain {
public static void main(String[] args) {

Cart1 cart = new Cart1();
cart.setName("MyCart1");

Items1 item1 = new Items1("I10", 10, 1, cart);
Items1 item2 = new Items1("I20", 20, 2, cart);
Set<Items1> itemsSet = new HashSet<Items1>();
itemsSet.add(item1);
itemsSet.add(item2);

cart.setItems1(itemsSet);
cart.setTotal(10 * 1 + 20 * 2);

SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try {
// Get Session
sessionFactory = HibernateAnnotationUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
System.out.println("Session created");
// start transaction
tx = session.beginTransaction();
// Save the Model object
```

```
session.save(cart);
session.save(item1);
session.save(item2);
// Commit transaction
tx.commit();
System.out.println("Cart1 ID=" + cart.getId());
System.out.println("item1 ID=" + item1.getId() + ", Foreign Key Cart ID=" +
item1.getCart1().getId());
System.out.println("item2 ID=" + item2.getId() + ", Foreign Key Cart ID=" +
item1.getCart1().getId());

} catch (Exception e) {
System.out.println("Exception occured. " + e.getMessage());
e.printStackTrace();
} finally {
if (!sessionFactory.isClosed()) {
System.out.println("Closing SessionFactory");
sessionFactory.close();
}
}
}
}
```

When we execute above hibernate one to many mapping annotation example test program, we get following output.

**Output**

```
Hibernate Annotation Configuration loaded
Hibernate Annotation serviceRegistry created
Session created
Hibernate: insert into CART (name, total) values (?, ?)
Hibernate: insert into ITEMS (cart_id, item_id, item_total, quantity) values (?, ?, ?, ?)
Hibernate: insert into ITEMS (cart_id, item_id, item_total, quantity) values (?, ?, ?, ?)
Cart1 ID=7
item1 ID=9, Foreign Key Cart ID=7
item2 ID=10, Foreign Key Cart ID=7
Closing SessionFactory
```

That’s all for hibernate one to many mapping, download the sample project from below link and do some more experiments.

## Hibernate Many To Many Mapping – Join Tables

Today we will look into Hibernate Many to Many Mapping using XML and annotation configurations. Earlier we looked how to implement One To One and One To Many mapping in Hibernate.

**Hibernate Many to Many**

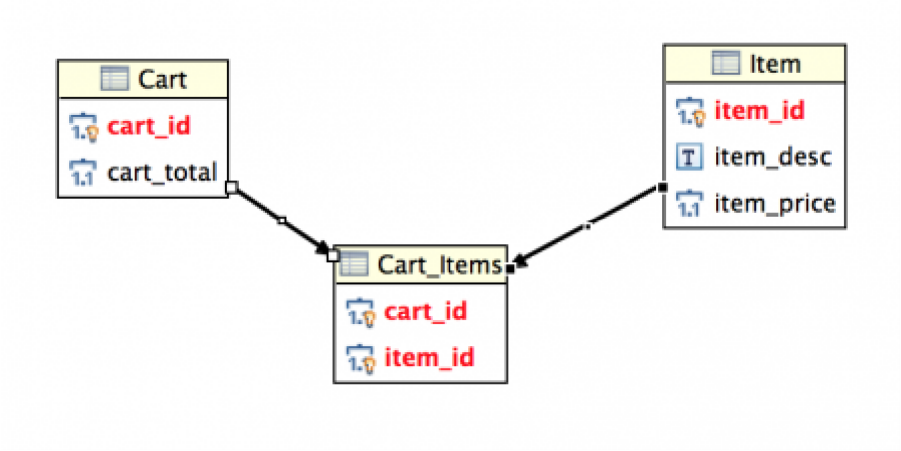
Many-to-Many mapping is usually implemented in database using a Join Table. For example we can have Cart and Item table and Cart_Items table for many-to-many mapping. Every cart can have multiple items and every item can be part of multiple carts, so we have a many to many mapping here.
Hibernate Many to Many Mapping Database Setup

Below script can be used to create our many-to-many example database tables, these scripts are for MySQL database. If you are using any other database, you might need to make small changes to get it working.

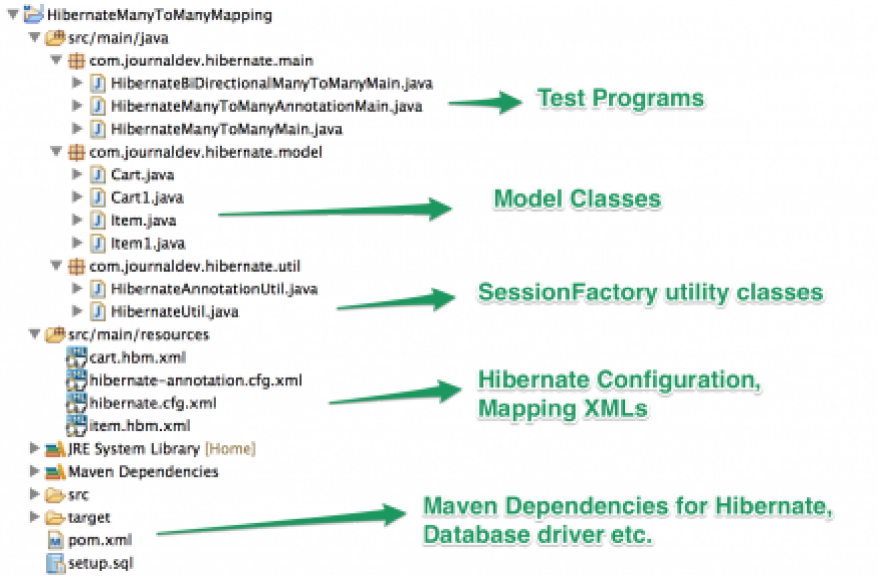
```
DROP TABLE IF EXISTS `Cart_Items`;
DROP TABLE IF EXISTS `Cart`;
DROP TABLE IF EXISTS `Item`;

CREATE TABLE `Cart` (
  `cart_id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `cart_total` decimal(10,0) NOT NULL,
  PRIMARY KEY (`cart_id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;

CREATE TABLE `Item` (
  `item_id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `item_desc` varchar(20) NOT NULL,
  `item_price` decimal(10,0) NOT NULL,
  PRIMARY KEY (`item_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `Cart_Items` (
  `cart_id` int(11) unsigned NOT NULL,
  `item_id` int(11) unsigned NOT NULL,
  PRIMARY KEY (`cart_id`,`item_id`),
  CONSTRAINT `fk_cart` FOREIGN KEY (`cart_id`) REFERENCES `Cart` (`cart_id`),
  CONSTRAINT `fk_item` FOREIGN KEY (`item_id`) REFERENCES `Item` (`item_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Notice that Cart_Items table doesn't have any extra columns, actually it doesn't make much sense to have extra columns in many-to-many mapping table. But if you have extra columns, the implementation changes little bit and we will look into that in another post.

Below diagram shows the Entity relationship between these tables.
Our Database setup is ready now, let's move on to create the hibernate many-to-many mapping project.




**Hibernate Many To Many Mapping Project Structure**
Create a maven project in Eclipse or your favorite IDE, below image shows the structure and different components in the application.




We will first look into the XML based mapping implementations and then move over to using JPA annotations.

**Hibernate Maven Dependencies**
Our final pom.xml contains Hibernate dependencies with latest version 4.3.5.Final and mysql driver dependencies.

**pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.journaldev.hibernate</groupId>
```

```
	<artifactId>HibernateManyToManyMapping</artifactId>
	<version>0.0.1-SNAPSHOT</version>

	<dependencies>
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-core</artifactId>
			<version>4.3.5.Final</version>
		</dependency>
		<dependency>
			<groupId>mysql</groupId>
			<artifactId>mysql-connector-java</artifactId>
			<version>5.0.5</version>
		</dependency>
	</dependencies>

</project>
```

## Hibernate Many to Many XML Configuration Model Classes

### Cart.java

```
package com.java.hibernate.model;

import java.util.Set;

public class Cart {
private long id;
private double total;
private Set<Items> items;

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public Set<Item> getItems() {
return items;
}

public void setItems(Set<Item> items) {
this.items = items;
}
}
```

**Items.java**

```
package com.java.hibernate.model;

import java.util.Set;

public class Items {
private long id;
private double price;
private String description;
private Set<Cart> carts;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public double getPrice() {
return price;
}

public void setPrice(double price) {
this.price = price;
}

public String getDescription() {
return description;
}

public void setDescription(String description) {
this.description = description;
}

public Set<Cart> getCarts() {
return carts;
}

public void setCarts(Set<Cart> carts) {
this.carts = carts;
}
}
```

Notice that Cart has set of Item and Item has set of Cart, this way we are implementing Bi-Directional associations. It means that we can configure it to save Item when we save Cart and vice versa.

For one-directional mapping, usually we have set in one of the model class. We will use annotations for one-directional mapping.

**Hibernate Many To Many Mapping XML Configuration**

Let's create hibernate many to many mapping xml configuration files for Cart and Item. We will implement bi-directional many-to-many mapping.

**cart.hbm.xml**

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.java.hibernate.model">
<class name="Cart" table="M2M_CART">
<id name="ID" type="long">
<column name="cart_id"/>
<generator class="native"/>
</id>

<property name="total" type="double" column="cart_total" />
<set name="items" table="M2M_CART_ITEMS" fetch="select" cascade="all">
<key column="cart_id"/>
<many-to-many class="Items" column="item_id"/>
</set>
</class>
</hibernate-mapping>
```

Notice that set of items is mapped to CART_ITEMS table. Since Cart is the primary object, cart_id is the key and many-to-many mapping is using Item class item_id column.

**item.hbm.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="com.java.hibernate.model">
<class name="Items" table="M2M_ITEMS">
<id name="id" type="long">
<column name="item_id" />
<generator class="identity" />
</id>
<property name="description" type="string" column="item_desc" />
<property name="price" type="double" column="item_price" />

<set name="carts" table="M2M_CART_ITEMS" fetch="select" cascade="all">
<key column="item_id" />
<many-to-many class="Cart" column="cart_id" />
</set>
</class>
</hibernate-mapping>
```

As you can see from above, the mapping is very similar to Cart mapping configurations.

**Hibernate Configuration for XML Based Many to Many Mapping**

Our hibernate configuration file looks like below.

**hibernate.cfg.xml**

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">Admin@123</property>
```

```
<!-- JDBC connection pool settings ... using built-in test pool -->
<property name="connection.pool_size">5</property>

<!-- Select our SQL dialect -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<!-- <property name="show_sql">true</property> -->
<!-- <property name="hbm2ddl.auto">create</property> -->

 <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>

        <mapping resource = "/resources/cart.hbm.xml"/>
        <mapping resource = "/resources/item.hbm.xml"/>
</session-factory>

</hibernate-configuration>
```

Hibernate SessionFactory Utility Class for XML Based Mapping

**HibernateUtil.java**
It's a simple utility class that works as a factory for SessionFactory.

**Hibernate Many To Many Mapping XML Configuration Test Program**

Our hibernate many to many mapping setup is ready, let's test it out. We will write two program, one is to save Cart and see that Item and Cart_Items information is also getting saved. Another one to save item data and check that corresponding Cart and Cart_Items are saved.

```
package com.java.hibernate.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
        private static SessionFactory sessionFactory;

        private static SessionFactory buildSessionFactory() {
                try {
                        // Create the SessionFactory from hibernate.cfg.xml
                        Configuration configuration = new Configuration();
                        configuration.configure("/resources/hibernate.cfg.xml");

                        System.out.println("Hibernate Configuration loaded");

                        //ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().ap-
plySettings(configuration.getProperties()).build();
                        System.out.println("Hibernate serviceRegistry created");

                        SessionFactory sessionFactory = configuration.buildSessionFactory();
                        return sessionFactory;
                } catch (Throwable ex) {
```

```
            System.err.println("Initial SessionFactory creation failed." + ex);
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
            }

            public static SessionFactory getSessionFactory() {
                    if(sessionFactory == null) {
                            sessionFactory = buildSessionFactory();
                    }

                    return sessionFactory;
            }
}
```

it's a simple utility class that works as a factory for SessionFactory.

**Hibernate Many To Many Mapping XML Configuration Test Program**

Our hibernate many to many mapping setup is ready, let's test it out. We will write two program, one is to save Cart and see that Item and Cart_Items information is also getting saved. Another one to save item data and check that corresponding Cart and Cart_Items are saved.

**HibernateManyToManyMain.java**

```
package com.java.hibernate;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.java.hibernate.model.Cart;
import com.java.hibernate.model.Items;
import com.java.hibernate.util.HibernateUtil;

public class HibernateManyToManyMain {
        // Saving many-to-many where Cart is primary
        public static void main(String[] args) {

                Items iphone = new Items();
                iphone.setPrice(100);
                iphone.setDescription("iPhone");

                Items ipod = new Items();
                ipod.setPrice(50);
                ipod.setDescription("iPod");

                Set<Items> items = new HashSet<Items>();
```

```
		items.add(iphone);
		items.add(ipod);

		Cart cart = new Cart();
		cart.setItems(items);
		cart.setTotal(150);

		Cart cart1 = new Cart();
		Set<Items> items1 = new HashSet<>();
		items1.add(iphone);
		cart1.setItems(items1);
		cart1.setTotal(100);

		SessionFactory sessionFactory = null;
		try {
			sessionFactory = HibernateUtil.getSessionFactory();
			Session session = sessionFactory.getCurrentSession();
			Transaction tx = session.beginTransaction();
			session.save(cart);
			session.save(cart1);
			System.out.println("Before committing transaction");
			tx.commit();
			sessionFactory.close();

			System.out.println("Cart ID=" + cart.getId());
			System.out.println("Cart1 ID=" + cart1.getId());
			System.out.println("Item1 ID=" + iphone.getId());
			System.out.println("Item2 ID=" + ipod.getId());

		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			if (sessionFactory != null && !sessionFactory.isClosed())
				sessionFactory.close();
		}
	}
}
```

When we execute above hibernate many to many mapping example program, we get following output.

```
Hibernate Configuration loaded
Hibernate serviceRegistry created
Hibernate: insert into CART (cart_total) values (?)
Hibernate: insert into ITEM (item_desc, item_price) values (?, ?)
Hibernate: insert into ITEM (item_desc, item_price) values (?, ?)
Hibernate: insert into CART (cart_total) values (?)
Before committing transaction
Hibernate: insert into CART_ITEMS (cart_id, item_id) values (?, ?)
Hibernate: insert into CART_ITEMS (cart_id, item_id) values (?, ?)
```

Hibernate: insert into CART_ITEMS (cart_id, item_id) values (?, ?)
Cart ID=1
Cart1 ID=2
Item1 ID=1
Item2 ID=2

Note that once the item data is saved through first cart, the item_id is generated and while saving second cart, it's not saved again.

Another important point to note is that many-to-many join table data is getting saved when we are committing the transaction. It's done for better performance incase we choose to rollback the transaction

## Hibernate Many To Many Mapping Annotation

Now that we have seen how to configure many-to-many mapping using hibernate xml configurations, let's see an example of implementing it through annotations. We will implement one-directional many-to-many mapping using JPA annotations.

### Hibernate Configuration XML File

Our annotations based hibernate configuration file looks like below.

**hibernate-annotation.cfg.xml**

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">Admin@123</property>

<!-- JDBC connection pool settings ... using built-in test pool -->
<property name="connection.pool_size">5</property>

<!-- Select our SQL dialect -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<!-- <property name="show_sql">true</property> -->
<!-- <property name="hbm2ddl.auto">create</property> -->

 <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>

        <mapping class="com.java.hibernate.model.Cart1"/>
        <mapping class="com.java.hibernate.model.Items1"/>
</session-factory>

</hibernate-configuration>
```

**Hibernate SessionFactory utility class**
Our utility class to create SessionFactory looks like below.

**HibernateAnnotationUtil.java**

```
package com.java.hibernate.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateAnnotationUtil {
private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {
try {
// Create the SessionFactory from hibernate.cfg.xml
Configuration configuration = new Configuration();
configuration.configure("/resources/hibernate-annotation.cfg.xml");

System.out.println("Hibernate Configuration loaded");

//ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySet-
tings(configuration.getProperties()).build();
System.out.println("Hibernate serviceRegistry created");

SessionFactory sessionFactory = configuration.buildSessionFactory();
return sessionFactory;
} catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
}

public static SessionFactory getSessionFactory() {
if(sessionFactory == null) {
sessionFactory = buildSessionFactory();
}

return sessionFactory;
}
}
```

**Hibernate Many to Many Mapping Annotation Model Classes**
This is the most important part for annotation based mapping, let's first look at the Item table model class and then we will look into Cart table model class.

**Item1.java**

```
package com.java.hibernate.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
```

```
@Table(name="ITEM")
public class Items1 {

@Id
@Column(name="item_id")
@GeneratedValue(strategy=GenerationType.IDENTITY)
private long id;

@Column(name="item_price")
private double price;

@Column(name="item_desc")
private String description;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public double getPrice() {
return price;
}

public void setPrice(double price) {
this.price = price;
}

public String getDescription() {
return description;
}

public void setDescription(String description) {
this.description = description;
}
}
```

Item1 class looks simple, there is no relational mapping here.

**Cart1.java**

```
package com.java.hibernate.model;

import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table(name = "M2M_CART_1")
public class Cart1 {
```

```
@Id
@Column(name = "cart_id")
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;

@Column(name = "cart_total")
private double total;

@ManyToMany(targetEntity=Items1.class, cascade={CascadeType.ALL})
@JoinTable(name="M2M_CART_ITEMS", joinColumns={@JoinColumn(name="cart_id")}, inver-
seJoinColumns={@JoinColumn(name="item_id")})
private Set<Items1> items;

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public Set<Items1> getItems() {
return items;
}

public void setItems(Set<Items1> items) {
this.items = items;
}
}
```

Most important part here is the use of ManyToMany annotation and JoinTable annotation where we provide table name and columns to be used for many-to-many mapping.

**Hibernate Many To Many Annotation Mapping Test Program**
Here is a simple test program for our hibernate many to many mapping annotation based configuration.

**HibernateManyToManyAnnotationMain.java**

```
package com.java.hibernate;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
```

```
import com.java.hibernate.model.Cart1;
import com.java.hibernate.model.Items1;
import com.java.hibernate.util.HibernateAnnotationUtil;

public class HibernateManyToManyAnnotationMain {
	public static void main(String[] args) {
		Items1 item1 = new Items1();
		item1.setDescription("samsung");
		item1.setPrice(300);
		Items1 item2 = new Items1();
		item2.setDescription("nokia");
		item2.setPrice(200);
		Cart1 cart = new Cart1();
		cart.setTotal(500);
		Set<Items1> items = new HashSet<Items1>();
		items.add(item1);
		items.add(item2);
		cart.setItems(items);

		SessionFactory sessionFactory = null;
		try {
			sessionFactory = HibernateAnnotationUtil.getSessionFactory();
			Session session = sessionFactory.getCurrentSession();
			Transaction tx = session.beginTransaction();
			session.save(cart);
			System.out.println("Before committing transaction");
			tx.commit();
			sessionFactory.close();

			System.out.println("Cart ID=" + cart.getId());
			System.out.println("Item1 ID=" + item1.getId());
			System.out.println("Item2 ID=" + item2.getId());

		} catch (Exception e) {
			e.printStackTrace();
		} finally {
			if (sessionFactory != null && !sessionFactory.isClosed())
				sessionFactory.close();
		}
	}
}
```

When we execute above program, it produces following output.

```
Hibernate Annotation Configuration loaded
Hibernate Annotation serviceRegistry created
Hibernate: insert into CART (cart_total) values (?)
Hibernate: insert into ITEM (item_desc, item_price) values (?, ?)
```

```
Hibernate: insert into ITEM (item_desc, item_price) values (?, ?)
Before committing transaction
Hibernate: insert into CART_ITEMS (cart_id, item_id) values (?, ?)
Hibernate: insert into CART_ITEMS (cart_id, item_id) values (?, ?)
Cart ID=5
Item1 ID=6
Item2 ID=5
```

It’s clear that saving cart is also saving data into Item and Cart_Items table. If you will save only item information, you will notice that Cart and Cart_Items data is not getting saved.

That’s all for Hibernate Many-To-Many mapping example tutorial, you can download the sample project from below link and play around with it to learn more.

# Hibernate One to One Mapping Example Annotation

## One to One Mapping in Hibernate

one to one mapping in hibernate, hibernate one to one mapping, hibernate one to one mapping annotation example

Most of the times, database tables are associated with each other. There are many forms of association – one-to-one, one-to-many and many-to-many are at the broad level. These can be further divided into unidirectional and bidirectional mappings. Today we will look into implementing Hibernate One to One Mapping using XML configuration as well as using annotation configuration.

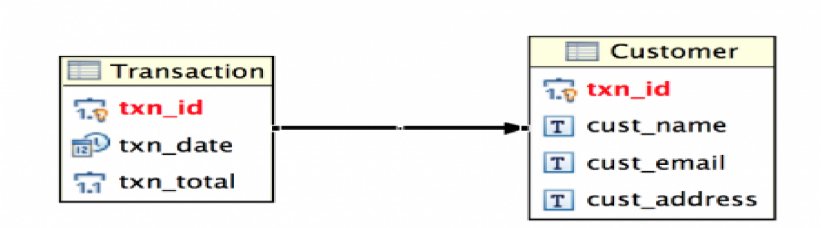
Hibernate One to One Mapping Example Database Setup

First of all we would need to setup One to One mapping in database tables. We will create two tables for our example – Transaction and Customer. Both of these tables will have one to one mapping. Transaction will be the primary table and we will be using Foreign Key in Customer table for one-to-one mapping.

I am providing MySQL script, that is the database I am using for this tutorial. If you are using any other database, make sure to change the script accordingly.

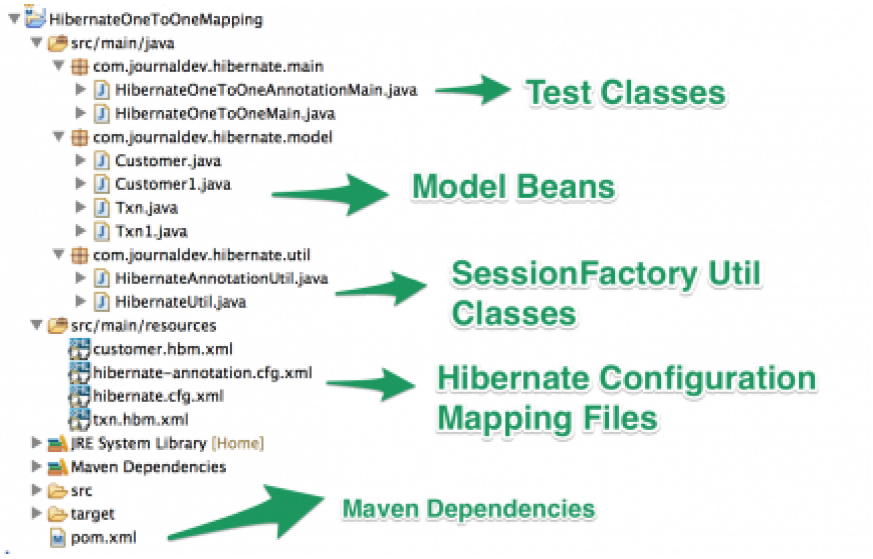
```
-- Create Transaction Table
CREATE TABLE `Transaction` (
  `txn_id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `txn_date` date NOT NULL,
  `txn_total` decimal(10,0) NOT NULL,
  PRIMARY KEY (`txn_id`)
) ENGINE=InnoDB AUTO_INCREMENT=16 DEFAULT CHARSET=utf8;

-- Create Customer table
CREATE TABLE `Customer` (
  `txn_id` int(11) unsigned NOT NULL,
  `cust_name` varchar(20) NOT NULL DEFAULT '',
  `cust_email` varchar(20) DEFAULT NULL,
  `cust_address` varchar(50) NOT NULL DEFAULT '',
  PRIMARY KEY (`txn_id`),
    CONSTRAINT `customer_ibfk_1` FOREIGN KEY (`txn_id`) REFERENCES `Transaction` (`txn_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Entity Relation Diagram (ERD) of above one-to-one mapping between tables looks like below image.



Our database setup is ready, let's move on the Hibernate One to One Example Project now.

**Hibernate One to One Mapping Example Project Structure**
Create a simple Maven project in your Java IDE, I am using Eclipse. Our final project structure will look like below image.




First of all we will look into XML Based Hibernate One to One Mapping example and then we will implement the same thing using annotation.

**Hibernate Maven Dependencies**
Our final pom.xml file looks like below.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.journaldev.hibernate</groupId>
  <artifactId>HibernateOneToOneMapping</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
        <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                <version>4.3.5.Final</version>
        </dependency>
        <dependency>
                <groupId>mysql</groupId>
                <artifactId>mysql-connector-java</artifactId>
                <version>5.0.5</version>
        </dependency>
  </dependencies>
</project>
```

Dependencies are just for hibernate and mysql java driver. Note that I am using Hibernate latest version 4.3.5.Final and MySQL java driver based on my MySQL database server version (5.0.5).

Hibernate 4 uses JBoss logging and it gets imported automatically as transitive dependency. You can confirm it in the maven dependencies of the project. If you are using Hibernate older versions, you might have to add slf4j dependencies.

**Hibernate One to One Mapping Model Classes**
Model classes for Hibernate One to One mapping to reflect database tables would be like below.

**Txn.java**

```
package com.java.hibernate.model;

import java.util.Date;

public class Txn {
private long id;
private Date date;
private double total;
private Customer customer;

@Override
public String toString() {
return id + ", " + total + ", " + customer.getName() + ", " + customer.getEmail() +
", "
+ customer.getAddress();
}

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public Date getDate() {
return date;
}

public void setDate(Date date) {
this.date = date;
}

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public Customer getCustomer() {
return customer;
}

public void setCustomer(Customer customer) {
```

```
this.customer = customer;
}
}
```

**Customer.java**

```
package com.java.hibernate.model;

public class Customer {
private long id;
private String name;
private String email;
private String address;

private Txn txn;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public String getName() {
return name;
}

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

public String getEmail() {
return email;
}

public void setEmail(String email) {
this.email = email;
}

public String getAddress() {
return address;
}

public void setAddress(String address) {
this.address = address;
}

public Txn getTxn() {
return txn;
}

public void setTxn(Txn txn) {
this.txn = txn;
}
}
```

Since we are using XML based configuration for mapping, above model classes are simple POJO classes or Java Beans with getter-setter methods. I am using class name as Txn to avoid confusion because Hibernate API have a class name as Transaction.

**Hibernate One to One Mapping Configuration**
Let's create hibernate one to one mapping configuration files for Txn and Customer tables.

**txn.hbm.xml**

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.java.hibernate.model">
<class name="Txn" table="TRANSACTION">
<id name="id" type="long" column="txn_id">
<generator class="native"></generator>
</id>

<property name="date" type="date"><column name="txn_date" /></property>
<property name="total" type="double"><column name="txn_total" /></property>

<one-to-one name="customer" class="Customer" cascade="save-update"/>
</class>
</hibernate-mapping>
```

The important point to note above is the hibernate one-to-one element for customer property.

**customer.hbm.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.org/dtd/hibernate-mapping-3.0.dtd" >
<hibernate-mapping package="com.java.hibernate.model">
<class name="Customer" table="O2O_CUSTOMER_1">
<id name="id" type="long">
<column name="txn_id" />
<generator class="foreign">
<param name="property">txn</param>
</generator>
</id>

<one-to-one name="txn" class="Txn" constrained="true" />
<property name="name" type="string">
<column name="cust_name"></column>
</property>
<property name="email" type="string">
<column name="cust_email"></column>
</property>
<property name="address" type="string">
<column name="cust_address"></column>
</property>
</class>
</hibernate-mapping>
```

generator class=”foreign” is the important part that is used for hibernate foreign key implementation.

**Hibernate Configuration File**
Here is the hibernate configuration file for XML based hibernate mapping configuration.

hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">Admin@123</property>

<!-- JDBC connection pool settings ... using built-in test pool -->
<property name="connection.pool_size">5</property>

<!-- Select our SQL dialect -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<!-- <property name="show_sql">true</property> -->
<property name="hbm2ddl.auto">create</property>

 <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>

        <mapping resource = "/resources/customer.hbm.xml"/>
        <mapping resource = "/resources/txn.hbm.xml"/>
</session-factory>

</hibernate-configuration>
```

Hibernate configuration file is simple, it has database connection properties and hibernate mapping resources.

**Hibernate SessionFactory Utility**

Here is the utility class to create hibernate SessionFactory instance.

```
package com.java.hibernate;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
        Configuration configuration = new Configuration();
        configuration.configure("/resources/hibernate-cfg.xml");
        System.out.println("Hibernate Configuration loaded");

        SessionFactory sessionFactory = configuration.buildSessionFactory();

            return sessionFactory;
        }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
```

```
    }

public static SessionFactory getSessionFactory() {
if(sessionFactory == null) sessionFactory = buildSessionFactory();
        return sessionFactory;
    }
}
```

That's it, lets write a test program to test the hibernate one to one mapping xml based configuration.

**Hibernate One to One Mapping XML Configuration Test Program**

In the hibernate one to one mapping example test program, first we will create Txn object and save it. Once it's saved into database, we will use the generated id to retrieve the Txn object and print it.

```
package com.java.hibernate;

import java.util.Date;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.java.hibernate.model.Customer;
import com.java.hibernate.model.Txn;
import com.java.hibernate.util.HibernateUtil;

public class HibernateOneToOneMain {
public static void main(String[] args) {

Txn txn = buildDemoTransaction();

SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try {
// Get Session
sessionFactory = HibernateUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
System.out.println("Session created");
// start transaction
tx = session.beginTransaction();
// Save the Model object
session.save(txn);
// Commit transaction
tx.commit();
System.out.println("Transaction ID=" + txn.getId());

// Get Saved Trasaction Data
printTransactionData(txn.getId(), sessionFactory);

} catch (Exception e) {
System.out.println("Exception occured. " + e.getMessage());
e.printStackTrace();
} finally {
if (!sessionFactory.isClosed()) {
System.out.println("Closing SessionFactory");
sessionFactory.close();
}
```

```
}
}

private static void printTransactionData(long id, SessionFactory sessionFactory) {
Session session = null;
Transaction tx = null;
try {
// Get Session
sessionFactory = HibernateUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
// start transaction
tx = session.beginTransaction();
// Save the Model object
Txn txn = (Txn) session.get(Txn.class, id);
// Commit transaction
tx.commit();
System.out.println("Transaction Details=\n" + txn);

} catch (Exception e) {
System.out.println("Exception occured. " + e.getMessage());
e.printStackTrace();
}
}

private static Txn buildDemoTransaction() {
Txn txn = new Txn();
txn.setDate(new Date());
txn.setTotal(100);

Customer cust = new Customer();
cust.setAddress("Bangalore, India");
cust.setEmail("pankaj@gmail.com");
cust.setName("Pankaj Kumar");

txn.setCustomer(cust);

cust.setTxn(txn);
return txn;
}
}
```

Now when we run above one to one mapping in hibernate test program, we get following output.

Hibernate Configuration loaded
Hibernate serviceRegistry created
Session created
Hibernate: insert into TRANSACTION (txn_date, txn_total) values (?, ?)
Hibernate: insert into CUSTOMER (cust_name, cust_email, cust_address, txn_id) values (?, ?, ?, ?)
Transaction ID=19
Hibernate: select txn0_.txn_id as txn_id1_1_0_, txn0_.txn_date as txn_date2_1_0_, txn0_.txn_total as txn_tota3_1_0_,
customer1_.txn_id as txn_id1_0_1_, customer1_.cust_name as cust_nam2_0_1_, customer1_.cust_email as cust_ema3_0_1_,
customer1_.cust_address as cust_add4_0_1_ from TRANSACTION txn0_ left outer join CUSTOMER customer1_ on
txn0_.txn_id=customer1_.txn_id where txn0_.txn_id=?

Transaction Details=
19, 100.0, Pankaj Kumar, pankaj@gmail.com, Bangalore, India
Closing SessionFactory

As you can see that it's working fine and we are able to retrieve data from both the tables using transaction id. Check the SQL used by Hibernate internally to get the data, its using joins to get the data from both the tables.

**Hibernate One to One Mapping Annotation**
In the above section, we saw how to use XML based configuration for hibernate one to one mapping, now let's see how we can use JPA and Hibernate annotation to achieve the same thing.

**Hibernate Configuration File**
**hibernate-annotation.cfg.xml**

```
<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
<property name="connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
<property name="connection.username">system</property>
<property name="connection.password">Admin@123</property>

<!-- JDBC connection pool settings ... using built-in test pool -->
<property name="connection.pool_size">5</property>

<!-- Select our SQL dialect -->
<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<!-- <property name="show_sql">true</property> -->
<property name="hbm2ddl.auto">create</property>

 <!-- Set the current session context -->
        <property name="current_session_context_class">thread</property>

        <mapping class="com.java.hibernate.model.Txn1"/>
        <mapping class="com.java.hibernate.model.Customer1"/>
</session-factory>

</hibernate-configuration>
```

Hibernate configuration is simple, as you can see that I have two model classes that we will use with annotations – Txn1 and Customer1.

**Hibernate One to One Mapping Annotation Example Model Classes**
For hibernate one to one mapping annotation configuration, model classes are the most important part. Let's see how our model classes look.

```
package com.java.hibernate.model;

import java.util.Date;
```

```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

import org.hibernate.annotations.Cascade;

@Entity
@Table(name="TXN_1")
public class Txn1 {

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="txn_id")
private long id;

@Column(name="txn_date")
private Date date;

@Column(name="txn_total")
private double total;

@OneToOne(mappedBy="txn")
@Cascade(value=org.hibernate.annotations.CascadeType.SAVE_UPDATE)
private Customer1 customer;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}

public Date getDate() {
return date;
}

public void setDate(Date date) {
this.date = date;
}

public double getTotal() {
return total;
}

public void setTotal(double total) {
this.total = total;
}

public Customer1 getCustomer() {
return customer;
}

public void setCustomer(Customer1 customer) {
this.customer = customer;
```

```
}

@Override
public String toString(){
return id+", "+total+", "+customer.getName()+", "+customer.getEmail()+", "+custo-
mer.getAddress();
}
}
```

Notice that most of the annotations are from Java Persistence API because Hibernate provide it's implementation. However for cascading, we would need to use Hibernate annotation org.hibernate.annotations.Cascade and enum org.hibernate.annotations.CascadeType.

Customer1

```
package com.java.hibernate.model;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.PrimaryKeyJoinColumn;
import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;
import org.hibernate.annotations.Parameter;

@Entity
@Table(name = "O2O_CUSTOMER_11")
public class Customer1 {
@Id
@Column(name = "txn_id", unique = true, nullable = false)
@GeneratedValue(generator = "gen")
@GenericGenerator(name = "gen", strategy = "foreign", parameters = { @Para-
meter(name = "property", value = "txn") })
private long id;

@Column(name = "cust_name")
private String name;

@Column(name = "cust_email")
private String email;

@Column(name = "cust_address")
private String address;

@OneToOne
@PrimaryKeyJoinColumn
private Txn1 txn;

public long getId() {
return id;
}

public void setId(long id) {
this.id = id;
}
```

```
public String getName() {
return name;
}

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

public String getEmail() {
return email;
}

public void setEmail(String email) {
this.email = email;
}

public String getAddress() {
return address;
}

public void setAddress(String address) {
this.address = address;
}

public Txn1 getTxn() {
return txn;
}

public void setTxn(Txn1 txn) {
this.txn = txn;
}
}
```

Note that we would need to @GenericGenerator so that id is used from the txn rather than generating it.

**Hibernate SessionFactory Utility class**

Creating SessionFactory is independent of the way we provide hibernate mapping. Our utility class for creating SessionFactory looks like below.

```
package com.java.hibernate.util;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateAnnotationUtil {
private static SessionFactory sessionFactory;

private static SessionFactory buildSessionFactory() {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
        Configuration configuration = new Configuration();
        configuration.configure("/resources/hibernate-annotation.cfg.xml");
        System.out.println("Hibernate Configuration loaded");

        SessionFactory sessionFactory = configuration.buildSessionFactory();

            return sessionFactory;
```

```
        }
        catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            ex.printStackTrace();
            throw new ExceptionInInitializerError(ex);
        }
    }

public static SessionFactory getSessionFactory() {
if(sessionFactory == null) sessionFactory = buildSessionFactory();
        return sessionFactory;
    }
}
```

Hibernate One to One Mapping Annotation Example Test Program
Here is a simple test program for our hibernate one to one mapping annotation example.

```
package com.java.hibernate;
import java.util.Date;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;

import com.java.hibernate.model.Customer1;
import com.java.hibernate.model.Txn1;
import com.java.hibernate.util.HibernateAnnotationUtil;

public class HibernateOneToOneAnnotationMain {

public static void main(String[] args) {

Txn1 txn = buildDemoTransaction();

SessionFactory sessionFactory = null;
Session session = null;
Transaction tx = null;
try{
//Get Session
sessionFactory = HibernateAnnotationUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
System.out.println("Session created using annotations configuration");
//start transaction
tx = session.beginTransaction();
//Save the Model object
session.save(txn);
//Commit transaction
tx.commit();
System.out.println("Annotation Example. Transaction ID="+txn.getId());

//Get Saved Trasaction Data
printTransactionData(txn.getId(), sessionFactory);
}catch(Exception e){
System.out.println("Exception occured. "+e.getMessage());
e.printStackTrace();
}finally{
if(sessionFactory != null && !sessionFactory.isClosed()){
System.out.println("Closing SessionFactory");
sessionFactory.close();
```

```
}
}
}

private static void printTransactionData(long id, SessionFactory sessionFactory) {
Session session = null;
Transaction tx = null;
try{
//Get Session
sessionFactory = HibernateAnnotationUtil.getSessionFactory();
session = sessionFactory.getCurrentSession();
//start transaction
tx = session.beginTransaction();
//Save the Model object
Txn1 txn = (Txn1) session.get(Txn1.class, id);
//Commit transaction
tx.commit();
System.out.println("Annotation Example. Transaction Details=\n"+txn);

}catch(Exception e){
System.out.println("Exception occured. "+e.getMessage());
e.printStackTrace();
}
}

private static Txn1 buildDemoTransaction() {
Txn1 txn = new Txn1();
txn.setDate(new Date());
txn.setTotal(100);

Customer1 cust = new Customer1();
cust.setAddress("San Jose, USA");
cust.setEmail("pankaj@yahoo.com");
cust.setName("Pankaj Kr");

txn.setCustomer(cust);

cust.setTxn(txn);
return txn;
}

}
```

Here is the output snippet when we execute above program.

```
Hibernate Annotation Configuration loaded
Hibernate Annotation serviceRegistry created
Session created using annotations configuration
Hibernate: insert into TRANSACTION (txn_date, txn_total) values (?, ?)
Hibernate: insert into CUSTOMER (cust_address, cust_email, cust_name, txn_id) va-
lues (?, ?, ?, ?)
Annotation Example. Transaction ID=20
Hibernate: select txn1x0_.txn_id as txn_id1_1_0_, txn1x0_.txn_date as
txn_date2_1_0_, txn1x0_.txn_total as txn_tota3_1_0_,
customer1x1_.txn_id as txn_id1_0_1_, customer1x1_.cust_address as cust_add2_0_1_,
customer1x1_.cust_email as cust_ema3_0_1_,
customer1x1_.cust_name as cust_nam4_0_1_ from TRANSACTION txn1x0_ left outer join
CUSTOMER customer1x1_ on
txn1x0_.txn_id=customer1x1_.txn_id where txn1x0_.txn_id=?
```

```
Annotation Example. Transaction Details=
20, 100.0, Pankaj Kr, pankaj@yahoo.com, San Jose, USA
Closing SessionFactory
```

Notice that the output is similar to hibernate one to one XML based configuration.

That’s all for Hibernate One to One mapping example, you can download the final project from below link and play around with it to learn more.