Hibernate session factory :

Old legacy (semi deprecated approach) as hibernate 5.2: instead MetadataSources will be used

If XML mapping files are in the classpath, use addResource(). For example:

Configuration cfg = new Configuration()

// addResource does a classpath resource lookup

.addResource("Item.hbm.xml")

.addResource("Bid.hbm.xml")

// calls addResource using "/org/hibernate/auction/User.hbm.xml"

.addClass(`org.hibernate.auction.User.class`)

// parses Address class for mapping annotations

.addAnnotatedClass( Address.class )

// reads package-level (package-info.class) annotations in the named package

.addPackage( "org.hibernate.auction" )

Configuration also allows you to specify configuration properties. For example:

Configuration cfg = new Configuration()

.setProperty("hibernate.dialect", "org.hibernate.dialect.H2Dialect")

.setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")

.setProperty("hibernate.order\_updates", "true");

**As per Hibernate 5.2 :**

The bootstrapping API is quite flexible, but in most cases it makes the most sense to think of it as a 3 step process:

1. Build the StandardServiceRegistry
2. Build the Metadata
3. Use those 2 things to build the SessionFactory

*Example 9. Building SessionFactory via SessionFactoryBuilder*

StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()

.configure( "org/hibernate/example/MyCfg.xml" )

.build();

Metadata metadata = new MetadataSources( standardRegistry )

.addAnnotatedClass( MyEntity.class )

.addAnnotatedClassName( "org.hibernate.example.Customer" )

.addResource( "org/hibernate/example/Order.hbm.xml" )

.addResource( "org/hibernate/example/Product.orm.xml" )

.getMetadataBuilder()

.applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )

.build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder() .build();

**Alternative way:**

****

Hibernate Associations :

orphanRemoval =true :

orphanRemoval has nothing to do with ON DELETE CASCADE.

orphanRemoval is an entirely **ORM-specific thing**. It marks "child" entity to be removed when it's no longer referenced from the "parent" entity, e.g. when you remove the child entity from the corresponding collection of the parent entity.

ON DELETE CASCADE is a **database-specific thing**, it deletes the "child" row in the database when the "parent" row is delete

Ex:

When an Employee entity object is removed, the remove operation is cascaded to the referenced Address entity object. In this regard, orphanRemoval=true and cascade=CascadeType.REMOVE are identical, and if orphanRemoval=true is specified, CascadeType.REMOVE is redundant.

The difference between the two settings is in the response to disconnecting a relationship. For example, such as when setting the address field to null or to another Address object.

* If orphanRemoval=true is specified the disconnected Address instance is automatically removed. This is useful for cleaning up dependent objects (e.g. Address) that should not exist without a reference from an owner object (e.g. Employee).
* If only cascade=CascadeType.REMOVE is specified, no automatic action is taken since disconnecting a relationship is not a remove operation.

To avoid dangling references as a result of orphan removal, this feature should only be enabled for fields that hold private non shared dependent objects.

**@naturalId:**

In the real world, most objects have a natural identifier. Typical examples are the ISBN number of a book, a company’s tax identifier or a person’s social security number. You could, of course, use these identifiers as primary keys. But most often, it’s a better idea to [generate numeric, surrogate keys](https://www.thoughts-on-java.org/jpa-generate-primary-keys/)

## **Define an attribute as a natural id**

@NaturalId

private String registrationNumber;

The only thing you have to do to model an attribute is a natural id, is to add the [@NaturalId](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/annotations/NaturalId.html)annotation. You can see an example in the following code snippet. The isbn number of a Book is a typical natural id. It identifies the record but is more complex than the primary key id. The idattribute is a surrogate key and [gets generated by Hibernate](https://www.thoughts-on-java.org/jpa-generate-primary-keys/).

Hibernate’s Session interface provides the methods [byNaturalId](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html" \l "byNaturalId-java.lang.Class-" \t "_blank) and [bySimpleNaturalId](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html" \l "bySimpleNaturalId-java.lang.Class-" \t "_blank) to read an entity by its natural identifier from the database. Let’s have a look at the byNaturalId method first.

The call of the [using](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/NaturalIdLoadAccess.html#using-java.lang.String-java.lang.Object-) method provides the name of the natural ID attribute and its value. If the natural ID consists of multiple attributes, you have to call this method multiple times to define each part of the ID.

Book b = session.byNaturalId(Book.class).using(Book\_.isbn.getName(), “978-0321356680”).load();

Hibernate performs 2 queries, as you can see in log messages below. The first query selects the primary for the given natural Id and the second one uses it to get the entity.

The bySimpleNaturalId method provides a convenient option to select entities with simple natural IDs that consist of only one attribute. As you can see in the following code snippet, you can provide the natural ID value directly to the load method and don’t need to call the usingmethod.

|  |
| --- |
| Book b = session.bySimpleNaturalId(Book.class).load(“978-0321356680”); |

**Merge() in Hibernate :**

Employee empmanaged=(Employee) session.merge(emp);

emp.setName(emp.getName()+"AM-AF"); // it will not update the DB

empmanaged.setName(emp.getName()+"AM-AF"); // will update the DB

transaction.commit();

First of all, using annotations as our configure method is just a convenient method instead of coping the endless XML configuration file.

The @Idannotation is inherited from javax.persistence.Id， indicating the member field below is the primary key of current entity. Hence your Hibernate and spring framework as well as you can do some reflect works based on this annotation. for details please check [javadoc for Id](http://docs.oracle.com/javaee/6/api/javax/persistence/Id.html)

The @GeneratedValue annotation is to configure the way of increment of the specified column(field). For example when using Mysql, you may specify auto\_increment in the definition of table to make it self-incremental, and then use

@GeneratedValue(strategy = GenerationType.IDENTITY)

in the Java code to denote that you also acknowledged to use this database server side strategy. Also, you may change the value in this annotation to fit different requirements.

### **1. Define Sequence in database**

For instance, Oracle has to use sequence as increment method, say we create a sequence in Oracle:

create sequence oracle\_seq;

### **2. Refer the database sequence**

Now that we have the sequence in database, but we need to establish the relation between Java and DB, by using [@SequenceGenerator](http://docs.oracle.com/javaee/6/api/javax/persistence/SequenceGenerator.html):

@SequenceGenerator(name="seq",sequenceName="oracle\_seq")

sequenceName is the real name of a sequence in Oracle, name is what you want to call it in Java. You need to specify sequenceName if it is different from name, otherwise just use name. I usually ignore sequenceName to save my time.

### **3. Use sequence in Java**

Finally, it is time to make use this sequence in Java. Just add [@GeneratedValue](http://docs.oracle.com/javaee/6/api/javax/persistence/SequenceGenerator.html):

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")

The generator field refers to which sequence generator you want to use. Notice it is not the real sequence name in DB, but the name you specified in name field of SequenceGenerator.

### **4. Complete**

So the complete version should be like this:

public class MyTable

{

@Id

@SequenceGenerator(name="seq",sequenceName="oracle\_seq")

@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="seq")

private Integer pid;

}

Unidirectional associations :

@ManyToOne

@ManyToOne is the most common association, having a direct equivalent in the relational database as well (e.g. foreign key), and so it establishes a relationship between a child entity and a parent.

*Example 119. @ManyToOne association*

@Entity(name = "Person")

public static class Person {

@Id

@GeneratedValue

private Long id;

public Person() {

}

}

@Entity(name = "Phone")

public static class Phone {

@Id

@GeneratedValue

private Long id;

@Column(name = "`number`")

private String number;

@ManyToOne

@JoinColumn(name = "person\_id",

foreignKey = @ForeignKey(name = "PERSON\_ID\_FK")

)

private Person person;

public Phone() {

}

public Phone(String number) {

this.number = number;

}

public Long getId() {

return id;

}

public String getNumber() {

return number;

}

public Person getPerson() {

return person;

}

public void setPerson(Person person) {

this.person = person;

}

}

CREATE TABLE Person (

id BIGINT NOT NULL ,

PRIMARY KEY ( id )

)

CREATE TABLE Phone (

id BIGINT NOT NULL ,

number VARCHAR(255) ,

person\_id BIGINT ,

PRIMARY KEY ( id )

)

ALTER TABLE Phone

ADD CONSTRAINT PERSON\_ID\_FK

FOREIGN KEY (person\_id) REFERENCES Person

Each entity has a lifecycle of its own. Once the @ManyToOne association is set, Hibernate will set the associated database foreign key column.

*Example 120. @ManyToOne association lifecycle*

Person person = new Person();

entityManager.persist( person );

Phone phone = new Phone( "123-456-7890" );

phone.setPerson( person );

entityManager.persist( phone );

entityManager.flush();

phone.setPerson( null );

INSERT INTO Person ( id )

VALUES ( 1 )

INSERT INTO Phone ( number, person\_id, id )

VALUES ( '123-456-7890', 1, 2 )

UPDATE Phone

SET number = '123-456-7890',

person\_id = NULL

WHERE id = 2

Bidrectional :

@ManyToOne

@JoinColumn(name = "EMP\_DEPT\_ID")

**private** Department department;

@OneToMany :

The @OneToMany association is by definition a parent association, even if it’s a unidirectional or a bidirectional one. Only the parent side of an association makes sense to cascade its entity state transitions to children.

Every bidirectional association must have one owning side only (the child side), the other one being referred to as the inverse (or the mappedBy) side.

@Entity(name = "Person")

public static class Person {

@Id

@GeneratedValue

private Long id;

@OneToMany(mappedBy = "person", cascade = CascadeType.ALL, orphanRemoval = true)

private List<Phone> phones = new ArrayList<>();

@Entity(name = "Phone")

public static class Phone {

@Id

@GeneratedValue

private Long id;

@NaturalId

@Column(name = "`number`", unique = true)

private String number;

@ManyToOne

private Person person;

class Employee{

@ManyToOne

@JoinColumn(name = "EMP\_DEPT\_ID")

**private** Department department;

#### @OneToOne

The @OneToOne association can either be unidirectional or bidirectional. A unidirectional association follows the relational database foreign key semantics, the client-side owning the relationship. A bidirectional association features a mappedBy @OneToOne parent side too.

##### **Unidirectional @OneToOne**

*Example 125. Unidirectional @OneToOne*

@Entity(name = "Phone")

public static class Phone {

@Id

@GeneratedValue

private Long id;

@Column(name = "`number`")

private String number;

@OneToOne

@JoinColumn(name = "details\_id")

private PhoneDetails details;

CREATE TABLE Phone (

id BIGINT NOT NULL ,

number VARCHAR(255) ,

details\_id BIGINT ,

PRIMARY KEY ( id )

)

CREATE TABLE PhoneDetails (

id BIGINT NOT NULL ,

provider VARCHAR(255) ,

technology VARCHAR(255) ,

PRIMARY KEY ( id )

)

ALTER TABLE Phone

ADD CONSTRAINT FKnoj7cj83ppfqbnvqqa5kolub7

FOREIGN KEY (details\_id) REFERENCES PhoneDetails

From a relational database point of view, the underlying schema is identical to the unidirectional @ManyToOneassociation, as the client-side controls the relationship based on the foreign key column.

##### **Bidirectional @OneToOne**

*Example 126. Bidirectional @OneToOne*

@Entity(name = "Phone")

public static class Phone {

@Id

@GeneratedValue

private Long id;

@Column(name = "`number`")

private String number;

@OneToOne(mappedBy = "phone", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

private PhoneDetails details;

@Entity(name = "PhoneDetails")

public static class PhoneDetails {

@Id

@GeneratedValue

private Long id;

private String provider;

private String technology;

@OneToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "phone\_id")

private Phone phone;

}

CREATE TABLE Phone (

id BIGINT NOT NULL ,

number VARCHAR(255) ,

PRIMARY KEY ( id )

)

CREATE TABLE PhoneDetails (

id BIGINT NOT NULL ,

provider VARCHAR(255) ,

technology VARCHAR(255) ,

phone\_id BIGINT ,

PRIMARY KEY ( id )

)

ALTER TABLE PhoneDetails

ADD CONSTRAINT FKeotuev8ja8v0sdh29dynqj05p

FOREIGN KEY (phone\_id) REFERENCES Pho

When using a bidirectional @OneToOne association, Hibernate enforces the unique constraint upon fetching the child-side. If there are more than one children associated with the same parent, Hibernate will throw a constraint violation exception. Continuing the previous example, when adding another PhoneDetails, Hibernate validates the uniqueness constraint when reloading the Phone object.

*Example 128. Bidirectional @OneToOne unique constraint*

PhoneDetails otherDetails = new PhoneDetails( "T-Mobile", "CDMA" );

otherDetails.setPhone( phone );

entityManager.persist( otherDetails );

entityManager.flush();

entityManager.clear();

//throws javax.persistence.PersistenceException: org.hibernate.HibernateException: More than one row with the given identifier was found: 1

phone = entityManager.find( Phone.class, phone.getId() );

Miscellaneous :

@ColumnTransformer(

read = "decrypt( 'AES', '00', DEPT\_DESC )",

write = "encrypt('AES', '00', ?)"

)

Here decrypt should be supported by DB

# [**Default fetch type for one-to-one, many-to-one and one-to-many in Hibernate**](https://stackoverflow.com/questions/26601032/default-fetch-type-for-one-to-one-many-to-one-and-one-to-many-in-hibernate)

It depends on whether you are using JPA or Hibernate.

From the [JPA 2.0 spec](http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/), the defaults are:

OneToMany: LAZY

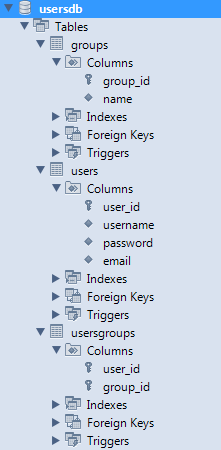
ManyToOne: EAGER

ManyToMany: LAZY

OneToOne: EAGER

And in hibernate, all is Lazy

Many to Many Relationships:



|  |
| --- |
| package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class Group {      private long id;      private String name;        private Set<User> users = new HashSet<User>();        public Group(String name) {          this.name = name;      }        public void addUser(User user) {          this.users.add(user);      }        // setters and getters  } |

File **net\codejava\hibernate\User.java**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class User {      private long id;      private String username;      private String password;      private String email;        private Set<Group> groups = new HashSet<Group>();        public User(String username, String password, String email) {          this.username = username;          this.password = password;          this.email = email;      }        public void addGroup(Group group) {          this.groups.add(group);      }        // setters and getters  } |

**NOTES:**For a bidirectional many-to-many association, we should specify a collection of another side from each side (using java.util.Set implementation in this tutorial). For example:

* + A Group has a set of Users.
  + A User has a set of Groups.

And note that the methods addUser(User) and addGroup(Group) were added to make adding items to the collection easily.

|  |
| --- |
| package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class Group {      private long id;      private String name;        private Set<User> users = new HashSet<User>();        public Group(String name) {          this.name = name;      }        public void addUser(User user) {          this.users.add(user);      }        // setters and getters  } |

File **net\codejava\hibernate\User.java**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class User {      private long id;      private String username;      private String password;      private String email;        private Set<Group> groups = new HashSet<Group>();        public User(String username, String password, String email) {          this.username = username;          this.password = password;          this.email = email;      }        public void addGroup(Group group) {          this.groups.add(group);      }        // setters and getters  } |

**NOTES:**For a bidirectional many-to-many association, we should specify a collection of another side from each side (using java.util.Set implementation in this tutorial). For example:

* + A Group has a set of Users.
  + A User has a set of Groups.

And note that the methods addUser(User) and addGroup(Group) were added to make adding items to the collection easily.

|  |
| --- |
| package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class Group {      private long id;      private String name;        private Set<User> users = new HashSet<User>();        public Group(String name) {          this.name = name;      }        public void addUser(User user) {          this.users.add(user);      }        // setters and getters  } |

File **net\codejava\hibernate\User.java**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.codejava.hibernate;    import java.util.HashSet;  import java.util.Set;    public class User {      private long id;      private String username;      private String password;      private String email;        private Set<Group> groups = new HashSet<Group>();        public User(String username, String password, String email) {          this.username = username;          this.password = password;          this.email = email;      }        public void addGroup(Group group) {          this.groups.add(group);      }        // setters and getters  } |

**NOTES:**For a bidirectional many-to-many association, we should specify a collection of another side from each side (using java.util.Set implementation in this tutorial). For example:

* + A Group has a set of Users.
  + A User has a set of Groups.

And note that the methods addUser(User) and addGroup(Group) were added to make adding items to the collection easily.

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

<!DOCTYPE hibernate-mapping PUBLIC

        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="net.codejava.hibernate">

    <class name="Group" table="GROUPS">

        <id name="id" column="GROUP\_ID">

            <generator class="native"/>

        </id>

        <property name="name" column="NAME" />

        <set name="users" table="UsersGroups" cascade="save-update">

            <key column="GROUP\_ID"/>

            <many-to-many column="USER\_ID" class="User" />

        </set>

    </class>

</hibernate-mapping>

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

<!DOCTYPE hibernate-mapping PUBLIC

        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="net.codejava.hibernate">

    <class name="User" table="USERS">

        <id name="id" column="USER\_ID">

            <generator class="native"/>

        </id>

        <property name="username" column="USERNAME" />

        <property name="password" column="PASSWORD" />

        <property name="email" column="EMAIL" />

        <set name="groups" table="UsersGroups" inverse="true">

            <key column="USER\_ID"/>

            <many-to-many column="GROUP\_ID" class="Group" />

        </set>

    </class>

</hibernate-mapping>

* We have to specify name of the join table (**UsersGroups**) in the **<set>** element.
* The attribute **inverse=”true”** can be placed on either side to specify which is owner of the relationship. In the mapping above, the relationship owner is the **Group** (because **inverse=”true”** is set on the **User** side).
* The attribute **cascade=”save-update”** on the **Group** side tells Hibernate to update the children (**users**) when the parent (**group**) is saved or updated.
* We don’t have to create model class/mapping file for the join table.