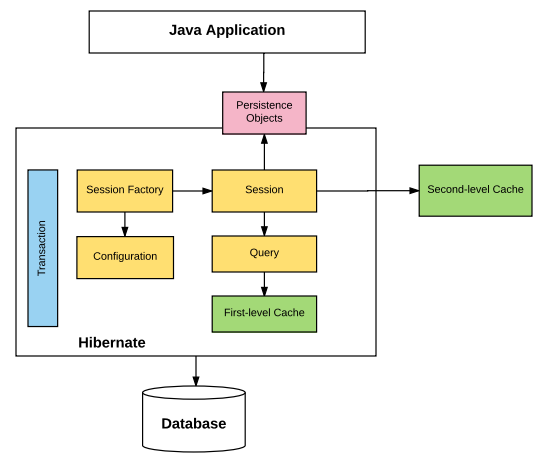
**Hibernate**

* Hibernate is an open source Java persistence framework.
* It performs powerful object relational mapping and query databases using HQL and SQL.
* Hibernate is a great tool for ORM mappings in java. It can cut down a lot of complexity and thus defects as well from your application, which may otherwise find a way to exist.
* This is specially boon for developers with limited knowledge of SQL.

**Hibernate Architecture**

1. **Configuration** :

* Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration.
* It is used by Session Factory to work with Java Application and the Database.
* It represents an entire set of mappings of an application Java Types to an SQL database.

1. **Session Factory :**

* Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.

1. **Session :**

* This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class.
* The instance of a session can be retrieved from the SessionFactory bean.
* Multiple sessions can be created from single session factory

1. **Query :**

* It allows applications to query the database for one or more stored objects.
* Hibernate provides different techniques to query database, including NamedQuery and Criteria API.

1. **First-level cache :**

* It represents the default cache used by Hibernate Session object while interacting with the database.
* It is also called as session cache and caches objects within the current session.
* All requests from the Session object to the database must pass through the first-level cache or session cache.
* One must note that the first-level cache is available with the session object until the Session object is live.

1. **Transaction :**

* Enables you to achieve data consistency, and rollback incase something goes unexpected.

1. **Persistent objects :**

* These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate.
* They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.

1. **Second-level cache :**

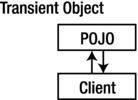
* It is used to store objects across sessions.
* This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is EhCache.

# **Hibernate Entity / Persistence LifeCycle States**

* [Hibernate](https://howtodoinjava.com/hibernate-tutorials/) works with normal Java objects that your application creates with the new operator. In raw form (without annotations), hibernate will not be able to identify your java classes; but when they are properly annotated with required annotations then hibernate will be able to identify them and then work with them
* Given an instance of an object that is mapped to Hibernate, it can be in any one of four different states: **transient, persistent, detached, or removed**.

# **Transient Object**

* Transient objects exist in heap memory. Hibernate does not manage transient objects or persist changes to transient objects.
* Transient Objects are not associated with the session.

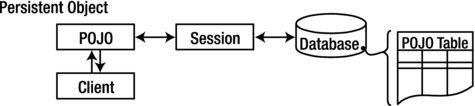


Transient objects are independent of Hibernate

* To persist the changes to a transient object, you would have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

# **Persistent Object**

* Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects.

Persistent objects are maintained by Hibernate

* If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

# **Detached Object**

* Detached objects have a representation in the database, but changes to the object will not be reflected in the database, and vice-versa. This temporary separation of the object and the database is shown in image below.



Detached objects exist in the database but are not maintained by Hibernate

* A detached object can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session’s evict() method.
* In order to persist changes made to a detached object, the application must reattach it to a valid Hibernate session. A detached instance can be associated with a new Hibernate session when your application calls one of the load, refresh, merge, update(), or save() methods on the new session with a reference to the detached object. After the call, the detached object would be a persistent object managed by the new Hibernate session.

# **How to generate primary keys**

* Hibernate supports 4 different primary key generation strategies which generate the primary key values programmatically or use database features, like auto-incremented columns or sequences.
* The only thing you have to do is to add the *@GeneratedValue* annotation to your primary key attribute and choose a generation strategy.

|  | @Id |
| --- | --- |
|  | @GeneratedValue |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

# **GenerationType.AUTO**

* The GenerationType.AUTO is the default generation type and lets the persistence provider choose the generation strategy.

|  | @Id |
| --- | --- |
|  | @GeneratedValue(strategy = GenerationType.AUTO) |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

* If you use Hibernate as your persistence provider, it selects a generation strategy based on the database specific dialect. For most popular databases, it selects GenerationType.IDENTITY

# **GenerationType.IDENTITY**

* It relies on an auto-incremented database column and lets the database generate a new value with each insert operation. From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn’t require any additional statements.

|  | @Id |
| --- | --- |
|  | @GeneratedValue(strategy = GenerationType.IDENTITY) |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

# **GenerationType.SEQUENCE**

* The *GenerationType.SEQUENCE* is my preferred way to generate primary key values and uses a database sequence to generate unique values.
* It requires additional select statements to get the next value from a database sequence.

|  | @Id |
| --- | --- |
|  | @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book\_generator") |
|  | @SequenceGenerator(name="book\_generator", sequenceName = "book\_seq", allocationSize=50) |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

# **GenerationType.TABLE**

* The *GenerationType.TABLE* gets only rarely used nowadays. It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order. This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do.

|  | @Id |
| --- | --- |
|  | @GeneratedValue(strategy = GenerationType.TABLE, generator = "book\_generator") |
|  | @TableGenerator(name="book\_generator", table="id\_generator", schema="bookstore") |
|  | @Column(name = "id", updatable = false, nullable = false) |
|  | private Long id; |

# **Composite Primary Keys In Hibernate**

## The *EmbeddedId* Annotation

* If the table has a primary key then in Entity class we configure that column using [@Id annotation](http://www.simplecodestuffs.com/auto-generate-primary-key-in-hibernate/). Even when the table doesn’t need a primary key, we must configure one column as id (one primary key is must).
* Now If the database table has more than one column as primary key then we call it as composite primary key, so if the table has multiple primary key columns , then in order to configure these primary key columns we need to create a new @Embeddable class containing the PK fields:

@Embeddable  
**public** **class** **RegistrationId** **implements** **Serializable**{  
  
        @Column(name = "STUDENT\_ID")  
        **private** **int** studentId;  
  
        @Column(name = "DEPARTMENT")  
        **private** **String** department;  
  
        // Create getters and Setters   
}

* And we should use it in the @Entity as a @EmbeddedId:

@Entity  
@Table(name = "STUDENT")  
**public** **class** **StudentEntity** {  
  
        @EmbeddedId  
        **private** **RegistrationId** regid;  
        ...  
        // Create Getters and Setters  
}

* To persist the entity:

**RegistrationId** regId = **new** **RegistrationId**();  
   regId.setStudentId(1);  
   regId.setDepartment("ECE");  
  
   **StudentEntity** student = **new** **StudentEntity**();   
   student.setRegid(regId);  
          
   session.save(student);

## 

## 

## 

## The *IdClass* Annotation

Let's say we have a table called Account and it has two columns, accountNumber and accountType, that form the composite key.

let's create an AccountId class with these primary key fields:

public class AccountId implements Serializable {

private String accountNumber;

private String accountType;

// default constructor

public AccountId(String accountNumber, String accountType) {

this.accountNumber = accountNumber;

this.accountType = accountType;

}

// equals() and hashCode()

}

Next let's associate the AccountId class with the entity Account.

In order to do that, we need to annotate the entity with the [@IdClass](https://www.baeldung.com/hibernate-identifiers) annotation. We must also declare the fields from the AccountId class in the entity Account and annotate them with @Id:

@Entity

@IdClass(AccountId.class)

public class Account {

@Id

private String accountNumber;

@Id

private String accountType;

// other fields, getters and setters

}

## *@IdClass* vs *@EmbeddedId*

with @IdClass we had to specify the columns twice, once in AccountId and again in Account; however, with @EmbeddedId we didn't.

With @IdClass, the query is a bit simpler:

SELECT account.accountNumber FROM Account account

With @EmbeddedId, we have to do one extra traversal:

SELECT book.bookId.title FROM Book book

# [**Hibernate Annotations**](http://www.codejava.net/frameworks/hibernate/getting-started-with-hibernate-annotations)

| **Annotation** | **Modifier** | **Description** |
| --- | --- | --- |
| @Entity |  | Marks a class as a Hibernate Entity (Mapped class) |
| @Table | Name | Maps this class with a database table specified by name modifier. If name is not supplied it maps the class with a table having same name as the class |
| @Id |  | Marks this class field as a primary key column |
| @GeneratedValue |  | Instructs database to generate a value for this field automatically |
| @Column | Name | Maps this field with table column specified by name and uses the field name if name modifier is absent |
| @ManyToMany | Cascade | Marks this field as the owning side of the many-to-many relationship and cascade modifier specifies which operations should cascade to the inverse side of relationship |
| mappedBy | This modifier holds the field which specifies the inverse side of the relationship |
| @JoinTable | Name | For holding this many-to-many relationship, maps this field with an intermediary database join table specified by name modifier |
| joinColumns | Identifies the owning side of columns which are necessary to identify a unique owning object |
| inverseJoinColumns | Identifies the inverse (target) side of columns which are necessary to identify a unique target object |
| @JoinColumn | Name | Maps a join column specified by the name identifier to the relationship table specified by @JoinTable |

# 

# **Hibernate get entity example – get vs load methods**

* To get hibernate entity by id using either session.load() or session.get() method

## session.load()

* Hibernate’s Session interface provides several overloaded load() methods for loading entities from the database. Each load() method requires the object’s primary key as an identifier, and it is mandatory to provide it.
* In addition to the ID, hibernate also needs to know which class or entity name to use to find the object with that ID. After the load() method returns, we need to cast the returned object to suitable type of class to further use it. It’s all what load() method need from us to work it correctly.

public Object load(Class theClass, Serializable id) throws HibernateException

public Object load(String entityName, Serializable id) throws HibernateException

public void load(Object object, Serializable id) throws HibernateException

public class TestHibernate

{

public static void main(String[] args)

{

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

// Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

//store the employee id generated for future use

Integer empId = emp.getEmployeeId();

sessionOne.getTransaction().commit();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Let's open a new session to test load() methods

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();

sessionTwo.beginTransaction();

//first load() method example

EmployeeEntity emp1 = (EmployeeEntity) sessionTwo.load(EmployeeEntity.class, empId);

System.out.println(emp1.getFirstName() + " - " +emp1.getLastName());

//Let's verify the entity name

System.out.println(sessionTwo.getEntityName(emp1));

sessionTwo.getTransaction().commit();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Session sessionThree = HibernateUtil.getSessionFactory().openSession();

sessionThree.beginTransaction();

//second load() method example

EmployeeEntity emp2 = (EmployeeEntity) sessionThree.load("com.emexo.demo.entity.EmployeeEntity", empId);

System.out.println(emp2.getFirstName() + " - " +emp2.getLastName());

sessionThree.getTransaction().commit();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Session sessionFour = HibernateUtil.getSessionFactory().openSession();

sessionFour.beginTransaction();

//third load() method example

EmployeeEntity emp3 = new EmployeeEntity();

sessionFour.load(emp3 , empId);

System.out.println(emp3.getFirstName() + " - " +emp3.getLastName());

sessionFour.getTransaction().commit();

HibernateUtil.shutdown();

}

}

## session.get()

The get() method is very much similar to load() method. The get() methods take an identifier and either an entity name or a class.

public Object get(Class clazz, Serializable id) throws HibernateException

public Object get(String entityName, Serializable id) throws HibernateException

## Difference between load() and get() methods

The difference between get and load methods lies in return value when the identifier does not exist in database.

* In case of get() method, we will get return value as NULL if identifier is absent.
* But in case of load() method, we will get a runtime exception.

# **Hibernate save() and saveOrUpdate() methods**

[hibernate](https://howtodoinjava.com/hibernate-tutorials/) works only with [persistent entities](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/) and persistent entities are classes which are attached to any hibernate session.

## save() method

Two versions of save() method

public Serializable save(Object object) throws HibernateException

public Serializable save(String entityName,Object object) throws HibernateException

* Both save() methods take a transient object reference (which must not be null) as an argument.

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

You may need to update the employee entity and then save again in another session.

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

sessionOne.getTransaction().commit();

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();

sessionTwo.beginTransaction();

emp.setLastName("temp");

//Save employee again second time

sessionTwo.save(emp);

sessionTwo.getTransaction().commit();

HibernateUtil.shutdown();

Here hibernate tried to insert the entity again. Though it was failed due to primary key check, but check may not be there for other entities and you may end up with duplicate rows.

BUT in the same session they will work correct.

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

emp.setLastName("Kundan");

//Save employee again second time

sessionOne.save(emp);

sessionOne.getTransaction().commit();

HibernateUtil.shutdown();

Any change to persistent entity is saved automatically.

## saveOrUpdate() method

saveOrUpdate() can be used with persistent as well as non-persistent entities both. Persistent entities will get updated, and transient entities will be inserted into database.

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

sessionOne.getTransaction().commit();

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();

sessionTwo.beginTransaction();

emp.setLastName("temp");

//Save employee again second time

sessionTwo.saveOrUpdate(emp);

sessionTwo.getTransaction().commit();

HibernateUtil.shutdown();

# **Hibernate Merging and Refreshing Entities**

## refresh() Method

Sometimes we face situations where our application database is modified with some external application/agent and thus corresponding hibernate entity in your application actually becomes out of sync with it’s database representation i.e. having old data. In this case, you can use session.refresh() method to re-populate the entity with the latest data available in the database.

public void refresh(Object object) throws HibernateException

public void refresh(Object object, LockMode lockMode) throws HibernateException

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Ajay");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

sessionOne.getTransaction().commit();

sessionOne.close();

//Verify employee's firstname

System.out.println(verifyEmployeeFirstName(1, "Ajay"));

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();

sessionTwo.beginTransaction();

//This

emp.setFirstName("Vikas");

sessionTwo.refresh(emp);

sessionTwo.getTransaction().commit();

sessionTwo.close();

System.out.println(emp.getFirstName().equals("Ajay"));

HibernateUtil.shutdown();

}

private static boolean verifyEmployeeFirstName(Integer employeeId, String firstName){

Session session = HibernateUtil.getSessionFactory().openSession();

EmployeeEntity employee = (EmployeeEntity) session.load(EmployeeEntity.class, employeeId);

//Verify first name

boolean result = firstName.equals(employee.getFirstName());

session.close();

//Return verification result

return result;

}

## merge() Method

* Method merge() does exactly opposite to what refresh() does i.e. It updates the database with values from a detached entity.
* Merging is performed when you desire to have a detached entity changed to persistent state again, with the detached entity’s changes migrated to (or overriding) the database

Object merge(Object object)

Object merge(String entityName, Object object)

Session sessionOne = HibernateUtil.getSessionFactory().openSession();

sessionOne.beginTransaction();

//Create new Employee object

EmployeeEntity emp = new EmployeeEntity();

emp.setEmployeeId(1);

emp.setFirstName("Lokesh");

emp.setLastName("Gupta");

//Save employee

sessionOne.save(emp);

sessionOne.getTransaction().commit();

sessionOne.close();

//Verify employee's firstname

System.out.println(verifyEmployeeFirstName(1, "Lokesh"));

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();

sessionTwo.beginTransaction();

//Set new first name

emp.setFirstName("Vikas");

//Merge the emp object using merge() method

EmployeeEntity mergedPersistentEmpEntity = (EmployeeEntity) sessionTwo.merge(emp);

sessionTwo.getTransaction().commit();

sessionTwo.close();

//Verify employee's firstname again in database

System.out.println(verifyEmployeeFirstName(1, "Vikas"));

HibernateUtil.shutdown();

}

private static boolean verifyEmployeeFirstName(Integer employeeId, String firstName){

Session session = HibernateUtil.getSessionFactory().openSession();

EmployeeEntity employee = (EmployeeEntity) session.load(EmployeeEntity.class, employeeId);

//Verify first name

boolean result = firstName.equals(employee.getFirstName());

session.close();

//Return verification result

return result;

}

# [**Hibernate**](http://www.codejava.net/frameworks/hibernate/hibernate-query-language-hql-example) **Mappings**

# **One to One Mapping**

* We have two entities here: Employee and Account.
* One Employee can have only one Account. Similarly, an account will be associated with one employee only. It’s one to one relationship for this example.
* we have to use [@OneToOne](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation
* Various supported techniques for one to one mapping

1. Using foreign key association

2. Using join table

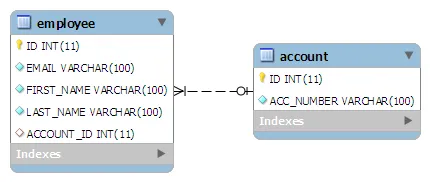
3. Using shared primary key

4. @MapsId

* First technique is most widely used and uses a foreign key column in one of the tables.
* Second technique uses a rather known solution of having a third table to store mapping between first two tables.
* Third technique is something new which uses a common primary key value in both the tables

## 1. Hibernate one to one mapping with foreign key association

In this kind of association, a foreign key column is created in owner entity. For example, if we make EmployeeEntity owner, then a extra column "ACCOUNT\_ID" will be created in Employee table. This column will store the foreign key for Account table.



**EmployeeEntity.java**

**@OneToOne**

**@JoinColumn(name="ACCOUNT\_ID")**

**private AccountEntity account;**

* The join column is declared with the [@JoinColumn](https://docs.oracle.com/javaee/5/api/javax/persistence/JoinColumn.html) annotation which looks like the [@Column](https://docs.oracle.com/javaee/5/api/javax/persistence/Column.html) annotation. It has one more parameters named referencedColumnName
* In a bidirectional relationship, one of the sides (and only one) has to be the owner. The owner is responsible for the association column(s) update. To declare a side as not responsible for the relationship, the attribute [mappedBy](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html#mappedBy%28%29) is used. ‘mappedBy’ refers to the property name of the association on the owner side.
* @JoinColumn should specify in the Owner entity and mapped by in the mapped entity

**AccountEntity.java**

| **@OneToOne(mappedBy="account")**  **private EmployeeEntity employee;** |
| --- |

## 

## 

## 

## 2. Hibernate one to one mapping with common join table

****

* In this technique, main annotation to be used is [@JoinTable](https://docs.oracle.com/javaee/5/api/javax/persistence/JoinTable.html). This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables.

**EmployeeEntity.java**

**@OneToOne(cascade = CascadeType.ALL)**

**@JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),**

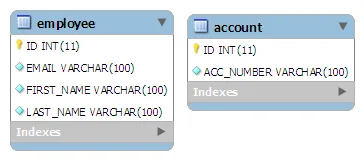
**inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))**

**private AccountEntity account;**

* @JoinTable annotation is used in EmployeeEntity class. It declares that a new table EMPLOYEE\_ACCOUNT will be created with two columns EMPLOYEE\_ID (primary key of EMPLOYEE table) and ACCOUNT\_ID (primary key of ACCOUNT table).

## 3. Hibernate one to one mapping with shared primary key

In this technique, hibernate will ensure that it will use a common primary key value in both the tables. This way primary key of EmployeeEntity can safely be assumed the primary key of AccountEntity also.



In this approach, [@PrimaryKeyJoinColumn](https://docs.oracle.com/javaee/5/api/javax/persistence/PrimaryKeyJoinColumn.html) is the main annotation to be used.

EmployeeEntity.java

**@OneToOne(cascade = CascadeType.ALL)**

**@PrimaryKeyJoinColumn**

**private AccountEntity account;**

**AccountEntity.java**

**@OneToOne(mappedBy="account", cascade=CascadeType.ALL)**

**private EmployeeEntity employee;**

## 4. Hibernate one to one mapping with @MapsId

* In this technique, hibernate assumes both the source and target share the same primary key values.
* In this approach, [@MapsId](https://javaee.github.io/javaee-spec/javadocs/javax/persistence/MapsId.html) is the main annotation to be used.

EmployeeEntity.java

**@Id**

**private Integer employeeId;**

**@OneToOne**

**@MapsId**

**private AccountEntity account;**

AccountEntity.java

**@Id**

**private Integer accountId;**





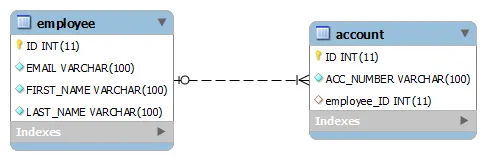


# **One to Many Mapping**

* Hibernate one to many mapping is made between two entities where first entity can have relation with multiple second entity instances but second can be associated with only one instance of first entity. Its 1 to N relationship.
* For example, in any company an employee can register multiple bank accounts but one bank account will be associated with one and only one employee.
* Hibernate one to many mapping solutions
  + Hibernate one to many mapping with foreign key association
  + Hibernate one to many mapping with join table

## Hibernate one to many mapping with foreign key association

In this approach, both entity will be responsible for making the relationship and maintaining it. EmployeeEntity should declare that relationship is one to many, and AccountEntity should declare that relationship from its end is many to one.



EmployeeEntity.java

**@OneToMany(mappedBy=”employee” cascade=CascadeType.ALL)**

**private Set<AccountEntity> accounts;**

**AccountEntity.java**

**@ManyToOne**

**@JoinColumn(name = "employee\_id")**

**private EmployeeEntity employee**

## Hibernate one to many mapping with join table

This approach uses a join table to store the associations between account and employee entities. @JoinTable annotation has been used to make this association.



EmployeeEntity.java

**@OneToMany(cascade=CascadeType.ALL)**

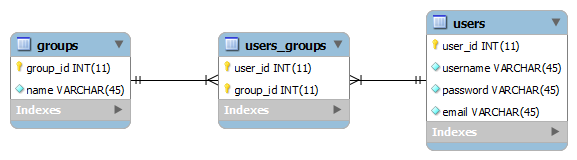
**@JoinTable(name="EMPLOYEE\_ACCOUNT", joinColumns={@JoinColumn(name="EMPLOYEE\_ID", referencedColumnName="ID")}**

**, inverseJoinColumns={@JoinColumn(name="ACCOUNT\_ID", referencedColumnName="ID")})**

**private Set<AccountEntity> accounts;**

# **Hibernate many to many mapping annotation example**

* we are going to illustrate how to use Hibernate annotations to map a many-to-many association (both bidirectional and unidirectional)

****

* Here, the multiplicity between the groups and users table is many-to-many, meaning that a group can have many users and vice-versa, a user can belong to many groups. A join table (users\_groups) is required to connect the both sides.
* We can see that both the Group and User classes have a collection (Set) of elements of each other, thus this association is bidirectional. Here, the Group is the owner side and the User is the other side.
* To map this many-to-many association, these JPA annotations are used: @ManyToMany, @JoinTable and @JoinColumn, besides the basic annotations (@Entity, @Column, @Id, etc) .
* On the owner side (Group)

| 1  2  3  4  5  6  7  8  9  10  11 | private Set<User> users = new HashSet<User>();    @ManyToMany(cascade = CascadeType.ALL)  @JoinTable(  name = "USERS\_GROUPS",  joinColumns = @JoinColumn(name = "GROUP\_ID"),  inverseJoinColumns = @JoinColumn(name = "USER\_ID")  )  public Set<User> getUsers() {  return users;  }   * Here, the @JoinTable annotation is used to specify the details of the join table (table name and two join columns - using the @JoinColumn annotation). The cascade attribute of the @ManyToMany annotation is required, so that Hibernate will update the associated users when the group is updated. * On the other side (User)   private Set<Group> groups = new HashSet<Group>();    @ManyToMany(mappedBy = "users")  public Set<Group> getGroups() {  return groups;  }   * This side is much simpler than the owner side, as we only need to specify the mappedBy attribute of the @ManyToMany annotation. That means this groups collection is mapped by the users collection on the owner side. * To make direction of the association becomes unidirectional, simply remove the groups collection from the User class: |
| --- | --- |

# **Hibernate JPA Cascade Types**

* We wanted to save the mapped entity whenever the relationship owner entity got saved. To enable this we had to use the “CascadeType” attribute.

## How JPA Cascade Types Work?

* Take a scenario where an Employee can have multiple Accounts; but one account must be associated with only one employee.

**EmployeeEntity.java**

| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {  private static final long serialVersionUID = -1798070786993154676L;  @Id  @Column(name = "ID", unique = true, nullable = false)  private Integer employeeId;  @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)  private String firstName;  @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)  private String lastName;    @OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts;    //Getters and Setters  }  **AccountEntity.java**   | @Entity  @Table(name = "Account")  public class AccountEntity implements Serializable  {  private static final long serialVersionUID = 1L;  @Id  @Column(name = "ID", unique = true, nullable = false)  @GeneratedValue(strategy = GenerationType.SEQUENCE)  private Integer accountId;  @Column(name = "ACC\_NO", unique = false, nullable = false, length = 100)  private String accountNumber;    @OneToOne (mappedBy="accounts", fetch = FetchType.LAZY)  private EmployeeEntity employee;    }   * Look at the above source code for EmployeeEntity.java. It defines “cascade=CascadeType.ALL” and it essentially means that any change happened on EmployeeEntity must cascade to AccountEntity as well. * If you save an employee, then all associated accounts will also be saved into database. * If you delete an Employee then all accounts associated with that Employee also be deleted. Simple enough. * But what if we only want to cascade only save operations but not delete operation. Then we need to clearly specify it using below code.   @OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts;   * Now only when save() or persist() methods are called using employee instance then only accounts will be persisted. If any other method is called on session, it’s effect will not affect/cascade to accounts. | | --- | |
| --- | --- |
| JPA Cascade Types  * **CascadeType.PERSIST :** cascade type presist means that save() or persist() operations cascade to related entities. * **CascadeType.MERGE :** cascade type merge means that related entities are merged when the owning entity is merged. * **CascadeType.REFRESH :** cascade type refresh does the same thing for the refresh() operation. * **CascadeType.REMOVE :** cascade type remove removes all related entities association with this setting when the owning entity is deleted. * **CascadeType.DETACH :** cascade type detach detaches all related entities if a “manual detach” occurs. * **CascadeType.ALL :** cascade type all is shorthand for all of the above cascade operations. * There is no default cascade type in JPA. By default no operations are cascaded. * The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:   @OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts; **Hibernate Lazy Loading**  * In any application, hibernate fetches data from database either in eager or lazy mode. Hibernate lazy loading refer to strategy when data is loaded lazily, on demand. * The default behavior is to load ‘property values eagerly’ and to load ‘collections lazily’. * [@OneToMany](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping-using-annotations/) and @ManyToMany associations are defaulted to LAZY loading * [@OneToOne](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping-using-annotations/) and @ManyToOne are defaulted to EAGER loading. * To enable lazy loading explicitly you must use “fetch = FetchType.LAZY” on a association which you want to lazy load when you are using hibernate annotations.   @OneToMany( mappedBy = "category", fetch = FetchType.LAZY )  private Set<ProductEntity> products;   * The simplest way that Hibernate can apply lazy load behavior upon the entities and associations is by providing a proxy implementation of them. |

# **Hibernate Query Language (HQL) Example**

* The Hibernate ORM framework provides its own query language called Hibernate Query Language or HQL for short. It is very powerful and flexible and has the following characteristics:
* SQL similarity:
  + HQL’s syntax is very similar to standard SQL. If you are familiar with SQL then writing HQL would be pretty easy: from SELECT, FROM, ORDER BY to arithmetic expressions and aggregate functions, etc.
* Fully object-oriented:
  + HQL doesn’t use real names of table and columns. It uses class and property names instead. HQL can understand inheritance, polymorphism and association.
* Case-insensitive for keywords:
  + Like SQL, keywords in HQL are case-insensitive. That means SELECT, select or Select are the same.
* Case-sensitive for Java classes and properties:
  + HQL considers case-sensitive names for Java classes and their properties, meaning Person and person are two different objects.

# How to execute HQL in Hibernate

* Basically, it’s fairly simple to execute HQL in Hibernate. Here are the steps:
* Write your HQL:

| 1 | String hql = "Your Query Goes Here"; |
| --- | --- |

* Create a Query from the Session:

| 1 | Query query = session.createQuery(hql); |
| --- | --- |

* Execute the query: depending on the type of the query (listing or update), an appropriate method is used:
  + For a listing query (SELECT):

| 1 | List listResult = query.list(); |
| --- | --- |

* + For an update query (INSERT, UPDATE, DELETE):

| 1 | int rowsAffected = query.executeUpdate(); |
| --- | --- |

* Extract result returned from the query: depending of the type of the query, Hibernate returns different type of result set. For example:
  + Select query on a mapped object returns a list of those objects.
  + Join query returns a list of arrays of Objects which are aggregate of columns of the joined tables. This also applies for queries using aggregate functions (count, sum, avg, etc).

# List Query Example

* The following code snippet executes a query that returns all Category objects:

| 1  2  3  4  5  6  7 | String hql = "from Category";  Query query = session.createQuery(hql);  List<Category> listCategories = query.list();    for (Category aCategory : listCategories) {      System.out.println(aCategory.getName());  } |
| --- | --- |

* Note that in HQL, we can omit the SELECT keyword and just use the FROM instead.

# Search Query Example

* The following statements execute a query that searches for all products in a category whose name is ‘Computer’:

| 1  2  3  4  5  6  7 | String hql = "from Product where category.name = 'Computer'";  Query query = session.createQuery(hql);  List<Product> listProducts = query.list();    for (Product aProduct : listProducts) {      System.out.println(aProduct.getName());  } |
| --- | --- |

* The cool thing here is Hibernate automatically generates JOIN query between the Product and Category tables behind the scene. Thus we don’t have to use explicit JOIN keyword:

| 1 | from Product where category.name = 'Computer' |
| --- | --- |

# Using Named Parameters Example

* You can parameterize your query using a colon before parameter name, for example **:id** indicates a placeholder for a parameter named **id**. The following example demonstrates how to write and execute a query using named parameters:

| 1  2  3  4  5  6  7  8  9  10  11 | String hql = "from Product where description like :keyword";    String keyword = "New";  Query query = session.createQuery(hql);  query.setParameter("keyword", "%" + keyword + "%");    List<Product> listProducts = query.list();    for (Product aProduct : listProducts) {      System.out.println(aProduct.getName());  } |
| --- | --- |

* The above HQL searches for all products whose description contains the specified keyword:

| 1 | from Product where description like :keyword |
| --- | --- |

* Then use the **setParameter(name, value)** method to set actual value for the named parameter:

| 1 | query.setParameter("keyword", "%" + keyword + "%"); |
| --- | --- |

* Note that we want to perform a LIKE search so the percent signs must be used outside the query string, unlike traditional SQL.

# Insert - Select Query Example

* HQL doesn’t support regular INSERT statement (you know why - because the session.save(Object) method does it perfectly). So we can only write INSERT … SELECT query in HQL. The following code snippet executes a query that inserts all rows from Category table to OldCategory table:

| 1  2  3  4  5  6  7  8  9 | String hql = "insert into Category (id, name)"          + " select id, name from OldCategory";    Query query = session.createQuery(hql);    int rowsAffected = query.executeUpdate();  if (rowsAffected > 0) {      System.out.println(rowsAffected + "(s) were inserted");  } |
| --- | --- |

* Note that HQL is object-oriented, so Category and OldCategory must be mapped class names (not real table names)

# Update Query Example

* The UPDATE query is similar to SQL. The following example runs a query that updates price for a specific product:

| 1  2  3  4  5  6  7  8  9  10 | String hql = "update Product set price = :price where id = :id";    Query query = session.createQuery(hql);  query.setParameter("price", 488.0f);  query.setParameter("id", 43l);    int rowsAffected = query.executeUpdate();  if (rowsAffected > 0) {      System.out.println("Updated " + rowsAffected + " rows.");  } |
| --- | --- |

# Delete Query Example

* Using DELETE query in HQL is also straightforward. For example:

| 1  2  3  4  5  6  7  8  9 | String hql = "delete from OldCategory where id = :catId";    Query query = session.createQuery(hql);  query.setParameter("catId", new Long(1));    int rowsAffected = query.executeUpdate();  if (rowsAffected > 0) {      System.out.println("Deleted " + rowsAffected + " rows.");  } |
| --- | --- |

# 8. Join Query Example

* HQL supports the following join types (similar to SQL):
  + **inner** **join** (can be abbreviated as **join**).
  + **left** **outer** **join** (can be abbreviated as **left** **join**).
  + **right** **outer** **join** (can be abbreviated as **right** **join**).
  + **full** **join**
* For example, the following code snippet executes a query that retrieves results which is a join between two tables Product and Category:

| 1  2  3  4  5  6  7  8  9  10 | String hql = "from Product p inner join p.category";    Query query = session.createQuery(hql);  List<Object[]> listResult = query.list();    for (Object[] aRow : listResult) {      Product product = (Product) aRow[0];      Category category = (Category) aRow[1];      System.out.println(product.getName() + " - " + category.getName());  } |
| --- | --- |

* Using the **join** keyword in HQL is called **explicit join**. Note that a JOIN query returns a list of Object arrays, so we need to deal with the result set differently:

| 1 | List<Object[]> listResult = query.list(); |
| --- | --- |

* HQL provides **with** keyword which can be used in case you want to supply extra join conditions. For example:

| 1 | from Product p inner join p.category with p.price > 500 |
| --- | --- |

* That joins the Product and Category together with a condition specifies that product’s price must be higher than 500.
* As stated earlier, we can write **implicit join** query which uses dot-notation. For example:

| 1 | from Product where category.name = 'Computer' |
| --- | --- |

* That result in **inner** **join** in the resulting SQL statement.

# 9. Sort Query Example

* Sorting in HQL is very similar to SQL using ORDER BY clause follows by a sort direction ASC (ascending) or DESC(descending). For example:

| 1  2  3  4  5  6  7  8 | String hql = "from Product order by price ASC";    Query query = session.createQuery(hql);  List<Product> listProducts = query.list();    for (Product aProduct : listProducts) {      System.out.println(aProduct.getName() + "\t - " + aProduct.getPrice());  } |
| --- | --- |

* That lists all products by the ascending order of price.

# 10. Group By Query Example

* Using GROUP BY clause in HQL is similar to SQL. The following query summarizes price of all products grouped by each category:

| 1 | select sum(p.price), p.category.name from Product p group by category |
| --- | --- |

* And here is the code snippet:

| 1  2  3  4  5  6  7  8  9  10 | String hql = "select sum(p.price), p.category.name from Product p group by category";    Query query = session.createQuery(hql);  List<Object[]> listResult = query.list();    for (Object[] aRow : listResult) {      Double sum = (Double) aRow[0];      String category = (String) aRow[1];      System.out.println(category + " - " + sum);  } |
| --- | --- |

# 11. Pagination Query Example

* To return a subset of a result set, the **Query**interface has two methods for limiting the result set:
  + **setFirstResult(int** **firstResult)**: sets the first row to retrieve.
  + **setMaxResults(int** **maxResults)**: sets the maximum number of rows to retrieve.
* For example, the following code snippet lists first 10 products:

| 1  2  3  4  5  6  7  8  9  10  11 | String hql = "from Product";    Query query = session.createQuery(hql);  query.setFirstResult(0);  query.setMaxResults(10);    List<Product> listProducts = query.list();    for (Product aProduct : listProducts) {      System.out.println(aProduct.getName() + "\t - " + aProduct.getPrice());  } |
| --- | --- |

# 12. Date Range Query Example

* A nice feature of Hibernate is that it is able to defer parameter type to generate the resulting SQL statement accordingly. So using date time parameters in HQL is quick and easy, for example:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | String hql = "from Order where purchaseDate >= :beginDate and purchaseDate <= :endDate";    Query query = session.createQuery(hql);    SimpleDateFormat dateFormatter = new SimpleDateFormat("yyyy-MM-dd");  Date beginDate = dateFormatter.parse("2014-11-01");    query.setParameter("beginDate", beginDate);    Date endDate = dateFormatter.parse("2014-11-22");  query.setParameter("endDate", endDate);    List<Order> listOrders = query.list();    for (Order anOrder : listOrders) {      System.out.println(anOrder.getProduct().getName() + " - "              +  anOrder.getAmount() + " - "              + anOrder.getPurchaseDate());  } |
| --- | --- |

* The above query lists only orders whose purchase date is in a specified range.

# 13. Using Expressions in Query

* For expressions used in the WHERE clause, HQL supports all basic arithmetic expressions similar to SQL include the following:
  + mathematical operators: **+, -, \*, /**
  + binary comparison operators: **=, >=, <=, <>, !=, like**
  + logical operators: **and, or, not**
  + etc
* For a complete list of expressions supported by Hibernate, [click here](https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/queryhql.html#queryhql-expressions).
* For example, the following query returns only products with price is ranging from 500 to 1000 dollars:

| 1 | from Product where price >= 500 and price <= 1000 |
| --- | --- |

# 14. Using Aggregate Functions in Query

* HQL supports the following aggregate functions:
  + **avg(…)**, **sum(…)**, **min(…)**, **max(…)**
  + **count(\*)**
  + **count(…)**, **count**(**distinct…)**, **count(all…)**
* For example, the following query counts all products:

| 1 | select count(name) from Product |
| --- | --- |

* And here’s the code snippet that shows how to extract the result:

| 1  2  3  4  5 | String hql = "select count(name) from Product";    Query query = session.createQuery(hql);  List listResult = query.list();  Number number = (Number) listResult.get(0);  System.out.println(number.intValue()); |
| --- | --- |

## Native SQL Queries

* We should probably use HQL whenever possible
* One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL.
* Another reason is that you may want to call stored procedures from your Hibernate application.
* Underlying Hibernate’s native SQL support is the org.hibernate.SQLQuery interface, which extends the org.hibernate.Query interface.
* Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

public SQLQuery createSQLQuery(String queryString) throws HibernateException

* The SQLQuery interface has addEntity(), addJoin(), and addScalar() methods

SQLQuery query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee");

List<Object[]> rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

## Notice that list() method returns the List of Object array, we need to explicitly parse them to double, long etc

### **Hibernate SQL Query addScalar**

Hibernate uses ResultSetMetadata to deduce the type of the columns returned by the query, from performance point of view we can use addScalar() method to define the data type of the column. However we would still get the data in form of Object array.

//Get All Employees - addScalar example

query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee")

.addScalar("emp\_id", new LongType())

.addScalar("emp\_name", new StringType())

.addScalar("emp\_salary", new DoubleType());

rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

**Hibernate Criteria Queries**

* Hibernate provides three different ways to retrieve data from database. We have already discussed **HQL and native SQL queries**. Now we will discuss our third option i.e. Criteria. The **Criteria Query API** lets you build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL.
* The Criteria API also includes **query by example (QBE)** functionality. This lets you supply example objects that contain the properties you would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count().

# Basic Usage Example

* The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains several createCriteria() methods. Pass the persistent object’s class or its entity name to the createCriteria() method, and Hibernate will create a Criteria object that returns instances of the persistence object’s class when your application executes a criteria query.
* The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

| Criteria crit = session.createCriteria(Product.class);  List<Product> results = crit.list(); |
| --- |

* Moving on from this simple example, we will add constraints to our criteria queries so we can whittle down the result set.

**Using Restrictions with Criteria**

* The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

# Restrictions.eq() Example

* To retrieve objects that have a property value that “equals” your restriction, use the eq() method on Restrictions, as follows:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.eq("description","Mouse"));  List<Product> results = crit.list() |
| --- |

* Above query will search all products having description as “Mouse”.

# ii) Restrictions.ne() Example

* To retrieve objects that have a property value “not equal to” your restriction, use the ne() method on Restrictions, as follows:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.ne("description","Mouse"));  List<Product> results = crit.list() |
| --- |

* Above query will search all products having description anything but not “Mouse”.

# Restrictions.like() and Restrictions.ilike() Example

* Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. The ilike() method is case-insensitive.

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));  List<Product> results = crit.list(); |
| --- |

* Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:
* ANYWHERE: Anyplace in the string  
  END: The end of the string  
  EXACT: An exact match  
  START: The beginning of the string

# iv) Restrictions.isNull() and Restrictions.isNotNull() Example

* The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.isNull("name"));  List<Product> results = crit.list(); |
| --- |

# v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples

* Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison is le(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java’s type promotions to handle the conversion to Double:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price", 25.0));  List<Product> results = crit.list(); |
| --- |

# vi) Combining Two or More Criteria Examples

* Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.lt("price",10.0));  crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));  List<Product> results = crit.list(); |
| --- |

* If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);  crit.add(orExp);  List results=crit.list(); |
| --- |

* The orExp logical expression that we have created here will be treated like any other criterion. We can therefore add another restriction to the criteria:

| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  Criterion name = Restrictions.like("name","Mou%");  LogicalExpression orExp = Restrictions.or(price,name);  crit.add(orExp);  crit.add(Restrictions.ilike("description","blocks%"));  List results = crit.list(); |
| --- |

# vii) Using Disjunction Objects with Criteria

* If we wanted to create an OR expression with more than two different criteria (for example, “price > 25.0 OR name like Mou% OR description not like blocks%”), we would use an org.hibernate.criterion.Disjunction object to represent a disjunction.
* You can obtain this object from the disjunction() factory method on the Restrictions class. The disjunction is more convenient than building a tree of OR expressions in code. To represent an AND expression with more than two criteria, you can use the conjunction() method, although you can easily just add those to the Criteria object. The conjunction can be more convenient than building a tree of AND expressions in code. Here is an example that uses the disjunction:

| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  Criterion browser = Restrictions.ilike("description", "browser", MatchMode.ANYWHERE);  Disjunction disjunction = Restrictions.disjunction();  disjunction.add(priceLessThan);  disjunction.add(mouse);  disjunction.add(browser);  crit.add(disjunction);  List results = crit.list(); |
| --- |

# viii) Restrictions.sqlRestriction() Example

* sqlRestriction() restriction allows you to directly specify SQL in the Criteria API. It’s useful if you need to use SQL clauses that Hibernate does not support through the Criteria API.
* Your application’s code does not need to know the name of the table your class uses. Use {alias} to signify the class’s table, as follows:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.sqlRestriction("{alias}.description like 'Mou%'"));  List<Product> results = crit.list(); |
| --- |

# Paging Through the ResultSet

* One common application pattern that criteria can address is pagination through the result set of a database query. There are two methods on the Criteria interface for paging, just as there are for Query: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to retrieve a fixed number of objects with the setMaxResults() method. Using both of these together, we can construct a paging component in our web or Swing application.

| Criteria crit = session.createCriteria(Product.class);  crit.setFirstResult(1);  crit.setMaxResults(20);  List<Product> results = crit.list(); |
| --- |

* As you can see, this makes paging through the result set easy. You can increase the first result you return (for example, from 1, to 21, to 41, etc.) to page through the result set.

**Obtaining a Unique Result**

* Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result. If you want obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.
* The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setMaxResults(1);  Product product = (Product) crit.uniqueResult(); |
| --- |

* Again, please note that you need to make sure that your query returns only one or zero results if you use the uniqueResult() method. Otherwise, Hibernate will throw a NonUniqueResultException exception.

**Obtaining Distinct Results**

* If you would like to work with distinct results from a criteria query, Hibernate provides a result transformer for distinct entities, org.hibernate.transform.DistinctRootEntityResultTransformer, which ensures that no duplicates will be in your query’s result set. Rather than using SELECT DISTINCT with SQL, the distinct result transformer compares each of your results using their default hashCode() methods, and only adds those results with unique hash codes to your result set. This may or may not be the result you would expect from an otherwise equivalent SQL DISTINCT query, so be careful with this.

| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setResultTransformer( DistinctRootEntityResultTransformer.INSTANCE )  List<Product> results = crit.list(); |
| --- |

* An additional performance note: the comparison is done in Hibernate’s Java code, not at the database, so non-unique results will still be transported across the network.

**Sorting the Query’s Results**

* Sorting the query’s results works much the same way with criteria as it would with HQL or SQL. The Criteria API provides the org.hibernate.criterion.Order class to sort your result set in either ascending or descending order, according to one of your object’s properties.
* This example demonstrates how you would use the Order class:

| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price",10.0));  crit.addOrder(Order.desc("price"));  List<Product> results = crit.list(); |
| --- |

* You may add more than one Order object to the Criteria object. Hibernate will pass them through to the underlying SQL query. Your results will be sorted by the first order, then any identical matches within the first sort will be sorted by the second order, and so on. Beneath the covers, Hibernate passes this on to an SQL ORDER BY clause after substituting the proper database column name for the property.

**Performing Associations (Joins)**

* The association works when going from either one-to-many or from many-to-one. First, we will demonstrate how to use one-to-many associations to obtain suppliers who sell products with a price over $25. Notice that we create a new Criteria object for the products property, add restrictions to the products’ criteria we just created, and then obtain the results from the supplier Criteria object:

| Criteria crit = session.createCriteria(Supplier.class);  Criteria prdCrit = crit.createCriteria("products");  prdCrit.add(Restrictions.gt("price",25.0));  List results = crit.list(); |
| --- |

* Going the other way, we obtain all the products from the supplier MegaInc using many-to-one associations:

| Criteria crit = session.createCriteria(Product.class);  Criteria suppCrit = crit.createCriteria("supplier");  suppCrit.add(Restrictions.eq("name","Hardware Are We"));  List results = crit.list(); |
| --- |

**Adding Projections and Aggregates**

* Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.
* To use projections, start by getting the org.hibernate.criterion.Projection object you need from the org.hibernate.criterion.Projections factory class. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method. When the Criteria object executes, the list contains object references that you can cast to the appropriate type.
* Example 1 : Single Aggregate ( Getting Row Count )

| Criteria crit = session.createCriteria(Product.class);  crit.setProjection(Projections.rowCount());  List<Long> results = crit.list(); |
| --- |

* Other aggregate functions available through the Projections factory class include the following:
  + avg(String propertyName): Gives the average of a property’s value
  + count(String propertyName): Counts the number of times a property occurs
  + countDistinct(String propertyName): Counts the number of unique values the property contains
  + max(String propertyName): Calculates the maximum value of the property values
  + min(String propertyName): Calculates the minimum value of the property values
  + sum(String propertyName): Calculates the sum total of the property values

**Example 2 : Multiple Aggregates**

* We can apply more than one projection to a given Criteria object. To add multiple projections, get a projection list from the projectionList() method on the Projections class. The org.hibernate.criterion.ProjectionList object has an add() method that takes a Projection object. You can pass the projections list to the setProjection() method on the Criteria object because ProjectionList implements the Projection interface.

| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.max("price"));  projList.add(Projections.min("price"));  projList.add(Projections.avg("price"));  projList.add(Projections.countDistinct("description"));  crit.setProjection(projList);  List<object[]> results = crit.list(); |
| --- |

**Example 3 : Getting Selected Columns**

* Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.property("name"));  projList.add(Projections.property("description"));  crit.setProjection(projList);  crit.addOrder(Order.asc("price"));  List<object[]> results = crit.list(); |
| --- |

**Query By Example (QBE)**

* In QBE, instead of programmatically building a Criteria object with Criterion objects and logical expressions, you can partially populate an instance of the object. You use this instance as a template and have Hibernate build the criteria for you based upon its values. This keeps your code clean and makes your project easier to test.
* For instance, if we have a user database, we can construct an instance of a user object, set the property values for type and creation date, and then use the Criteria API to run a QBE query. Hibernate will return a result set containing all user objects that match the property values that were set. Behind the scenes, Hibernate inspects the Example object and constructs an SQL fragment that corresponds to the properties on the Example object.
* The following basic example searches for suppliers that match the name on the example Supplier object:

| Criteria crit = session.createCriteria(Supplier.class);  Supplier supplier = new Supplier();  supplier.setName("MegaInc");  crit.add(Example.create(supplier));  List results = crit.list(); |
| --- |

# Hibernate @NamedQuery

* Named queries in hibernate is a**technique to group the HQL statements in single location**, and lately refer them by some name whenever need to use them. It **helps largely in code cleanup**because these HQL statements are no longer scattered in whole code.
* Apart from above, below are some minor **advantages** of named queries:
  + **Fail fast**: Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
  + **Reusable**: They can be accessed and used from several places which increase re-usability.
* But, you must know that named query really **make code less readable and sometimes debugging becomes more hard**, as you have to locate the actual query definition being executed and understand that as well.
* **Performance wise named queries does not make much difference**, nor put any excessive cost.
  + The *cost of transforming a HQL query to SQL is negligible* compared to the cost of actually executing the query.
  + The*memory cost of caching the query is really small*. Remember that Hibernate needs to have all the entities meta-data in memory anyway.
* In this tutorial, I am giving an example of named queries using annotation configuration.
* I have a DepartmentEntity.java class which is mapped to Department table in database. I have written two named queries i.e. one for updating a department name by it’s id, and second for selecting a department by it’s id.
* Named query definition has two important attributes:
  + **name**: The name of name query by which it will be located using hibernate session.
  + **query**: Here you give the HQL statement to get executed in database.

#### DepartmentEntity.java

| @Entity  @Table(name = "DEPARTMENT", uniqueConstraints = {                      @UniqueConstraint(columnNames = "ID"),                      @UniqueConstraint(columnNames = "NAME") })  @NamedQueries  (      {          @NamedQuery(name=DepartmentEntity.GET\_DEPARTMENT\_BY\_ID, query=DepartmentEntity.GET\_DEPARTMENT\_BY\_ID\_QUERY),          @NamedQuery(name=DepartmentEntity.UPDATE\_DEPARTMENT\_BY\_ID, query=DepartmentEntity.UPDATE\_DEPARTMENT\_BY\_ID\_QUERY)      }  )  public class DepartmentEntity implements Serializable {        static final String GET\_DEPARTMENT\_BY\_ID\_QUERY = "from DepartmentEntity d where d.id = :id";      public static final String GET\_DEPARTMENT\_BY\_ID = "GET\_DEPARTMENT\_BY\_ID";        static final String UPDATE\_DEPARTMENT\_BY\_ID\_QUERY = "UPDATE DepartmentEntity d SET d.name=:name where d.id = :id";      public static final String UPDATE\_DEPARTMENT\_BY\_ID = "UPDATE\_DEPARTMENT\_BY\_ID";        private static final long serialVersionUID = 1L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer id;        @Column(name = "NAME", unique = true, nullable = false, length = 100)      private String name;        public Integer getId() {          return id;      }        public void setId(Integer id) {          this.id = id;      }        public String getName() {          return name;      }        public void setName(String name) {          this.name = name;      }  } |
| --- |

To test, above named query I have written following code and executed both named queries.

#### TestHibernateNamedQuery.java

| public class TestHibernateNamedQuery  {      public static void main(String[] args)      {          //Open the hibernate session          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();          try          {              //Update record using named query              Query query = session.getNamedQuery(DepartmentEntity.UPDATE\_DEPARTMENT\_BY\_ID)                                          .setInteger("id", 1)                                          .setString("name", "Finance");              query.executeUpdate();                //Get named query instance              query = session.getNamedQuery(DepartmentEntity.GET\_DEPARTMENT\_BY\_ID)                                          .setInteger("id", 1);              //Get all departments (instances of DepartmentEntity)              DepartmentEntity department = (DepartmentEntity) query.uniqueResult();              System.out.println(department.getName());          }          finally          {              session.getTransaction().commit();              HibernateUtil.shutdown();          }      }  }    Output in console:    Hibernate: update DEPARTMENT set NAME=? where ID=?  Hibernate: select department0\_.ID as ID0\_, department0\_.NAME as NAME0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Finance |
| --- |

# Hibernate EhCache Configuration

<dependency>

    <groupId>net.sf.ehcache</groupId>

    <artifactId>ehcache</artifactId>

    <version>[2.0.0]</version>

    <type>pom</type>

</dependency>

<property key="hibernate.cache.use\_second\_level\_cache">true</property>

<propertyname="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

@Entity

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY,

region="department")

public class DepartmentEntity implements Serializable

{

    //code

}

### **Java Persistence API (JPA)**

* JPA is a Java API **specification** for relational data management in applications using Java SE and Java EE.
* JPA defines a **standard way** for simplifying database programming for developers, **reduce** development time and **increase** productivity.
* When using JPA, you have to import interfaces and classes from the package javax.persistence.
* JPA defines **Java Persistence Query Languag**e (JPQL) which is an object-oriented query language. The syntax of JPQL is similar to SQL but it operates against Java objects rather than directly with database tables.
* Remember JPA is a specification, and Hibernate is one of its **implementations**, among others such as EclipseLink and OpenJPA.
* Hibernate is a popular Object Relational Mapping (ORM) framework that aims at simplifying database programming for developers.
* Hibernate was developed before JPA. And after JPA becomes a standard, Hibernate restructures itself to become an implementation of JPA.

## Create MySQL Database

create a table name **users** with 4 columns: user\_id, fullname, email and password, using the following script:

| 1  2  3  4 | CREATE TABLE `users` (  `user\_id` int(11) NOT NULL AUTO\_INCREMENT,  `fullname` varchar(45) NOT NULL,  `email` varchar(45) NOT NULL,  `password` varchar(45) NOT NULL,  PRIMARY KEY (`user\_id`)  ) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
| --- | --- |

### 

### **Configure Maven Dependencies:**

Next, we need to add dependencies in Maven’s Project Object Model (pom.xml) for Hibernate, JPA and MySQL Connector Java. Open the pom.xml file in XML mode and insert the following XML just before the </project> tag:

| 1  2  3  4  5  6  7 | <dependencies>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>5.2.12.Final</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>8.0.8-dmr</version>  </dependency>  </dependencies> |
| --- | --- |

## 

## Code Entity Class

@Entity

@Table(name = "USERS")

public class User {

private Integer id;

private String fullname;

private String email;

private String password;

@Column(name = "USER\_ID")

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

public Integer getId() {

return id;

}

// other setters and getters are not shown for brevity

}

## 

## Create JPA Configuration File (persistence.xml)

Next, we need to create an XML configuration file for JPA called persistence.xml, in order to tell Hibernate how to connect to the database. This file must be present in the classpath, under the META-INF folder.

Under the src/main/resources folder, create a new folder named META-INF (Right-click, select **New > Other… > Folder**).

Right click on the newly created folder META-INF, select **New > Other… > XML > XML File**. Enter the file name as persistence.xml. And paste the following XML code:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <?xml version="1.0" encoding="UTF-8"?>  <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence  http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">  <persistence-unit name="UsersDB">  <properties>  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/usersdb" />  <property name="javax.persistence.jdbc.user" value="root" />  <property name="javax.persistence.jdbc.password" value="P@ssw0rd" />  <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />  <property name="hibernate.show\_sql" value="true" />  <property name="hibernate.format\_sql" value="true" />  </properties>  </persistence-unit>    </persistence> |
| --- | --- |

The root element <persistence> specifies the version of JPA to be used, and as you can see, we use JPA version 2.1.

The element <persistence-unit> specifies a unit of persistence with a name. The name (UsersDB) will be looked up by Java code.

Then we specify several properties for database connection information:

- javax.persistence.jdbc.url: specifies the [JDBC URL](https://www.codejava.net/java-se/jdbc/jdbc-database-connection-url-for-common-databases) points to the database.

- javax.persistence.jdbc.user: specifies the username of the account having privilege to access to the database.

- javax.persistence.jdbc.password: specifies the password of the user.

- javax.persistence.jdbc.driver: specifies the class name of the [JDBC driver](https://www.codejava.net/java-se/jdbc/jdbc-driver-library-download) to be used. Here we use MySQL Connector Java so the name is com.mysql.jdbc.Driver.

- hibernate.show\_sql: tells Hibernate to show SQL statements in standard output.

- hibernate.format\_sql: tells Hibernate to format the SQL statements.

So you may need to change the values for url, user, and password accordingly.

## 

## 

## 

## 

## 

## Understand EntityManager and EntityManagerFactory

Finally, we need to code a test program to check if everything we’ve done so far is correct or not. But let’s understand a couple of key interfaces in JPA first.

## EntityManager:

An EntityManager instance is associated with a persistence context, and it is used to interact with the database.

A persistence context is a set of entity instances, which are actually the objects or instances of the model classes.

So we use the EntityManager to manage entity instances and their life cycle, such as create entities, update entities, remove entities, find and query entities.

## 

## 

## 

## EntityManagerFactory:

An EntityManagerFactory is used to create an EntityManager. And EntityManagerFactory is associated with a persistence unit. In Java SE environments, an EntityManagerFactory can be obtained from the Persistence class.

And here are the typical steps to manage entity instances via JPA:

- Create an EntityManagerFactory from a persistence unit

- Create an EntityManager from the EntityManagerFactory

- Begin a transaction

- Manage entity instances (create, update, remove, find, query, etc)

- Commit the transaction

- Close the EntityManager and EntityManagerFactory

Let’s see the code details below.

## Code a Test Program

Now, let’s write some code to create, update, find, query and remove User entity instances using JPA. Create a new Java class under src/main/java folder called UserDAOTest.java, with the main() method.

## 

### **Persist an entity instance:**

In the main() method, add the following code to create an EntityManager and begin the transaction:

| 1  2  3  4 | EntityManagerFactory factory = Persistence.createEntityManagerFactory("UsersDB");  EntityManager entityManager = factory.createEntityManager();    entityManager.getTransaction().begin(); |
| --- | --- |

And write the following code to saves a new User object to the database:

| 1  2  3  4  5  6 | User newUser = new User();  newUser.setEmail("billjoy@gmail.com");  newUser.setFullname("bill Joy");  newUser.setPassword("billi");    entityManager.persist(newUser); |
| --- | --- |

As you can see, we call the **persist(Object)** method of the EntityManager class to save the User object to the underlying database.

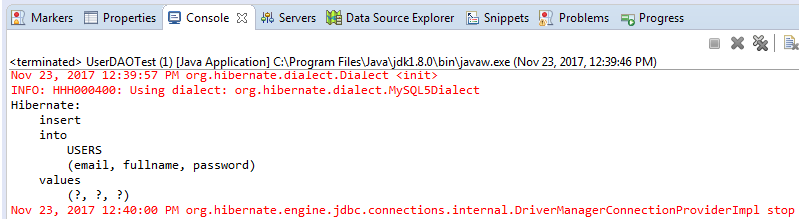
And finally commit the transaction and close the EntityManager and EntityManagerFactory:

| 1  2  3 | entityManager.getTransaction().commit();  entityManager.close();  factory.close(); |
| --- | --- |

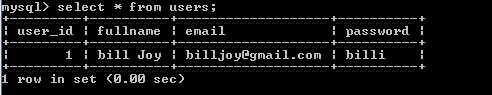
So the complete program should look like this:

| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import javax.persistence.EntityManager;  import javax.persistence.EntityManagerFactory;  import javax.persistence.Persistence;    public class UserDAOTest {    public static void main(String[] args) {  EntityManagerFactory factory = Persistence.createEntityManagerFactory("UsersDB");  EntityManager entityManager = factory.createEntityManager();    entityManager.getTransaction().begin();    User newUser = new User();  newUser.setEmail("billjoy@gmail.com");  newUser.setFullname("bill Joy");  newUser.setPassword("billi");    entityManager.persist(newUser);    entityManager.getTransaction().commit();    entityManager.close();  factory.close();  }  } |
| --- | --- |

Run this program, and you should see the following output in the Console view:



You see, Hibernate prints the SQL statement which is nicely formatted. That means the program has been executed successfully. You can check the result by typing the command “select \* from users” in MySQL Command Line Client tool:



You see, a new row created and you don’t have to write any SQL statement, right? That’s the power of using Hibernate/JPA for database programming.

Let’s see how to perform other operations.

### 

### 

### 

### 

### **Update an entity instance:**

The following code snippet updates an entity instance which is already persisted in the database:

| 1  2  3  4  5  6  7 | User existingUser = new User();  existingUser.setId(1);  existingUser.setEmail("bill.joy@gmail.com");  existingUser.setFullname("Bill Joy");  existingUser.setPassword("billionaire");    entityManager.merge(existingUser); |
| --- | --- |

As you can see, we need to specify the ID of the object and use the **merge(Object)** method of the EntityManager class.

### **Find an entity instance:**

To find an entity from the database, we use the method **find(Class<T> entityClass, Object primaryKey)** of the EntityManager class. For example:

| 1  2  3  4  5  6 | Integer primaryKey = 1;  User user = entityManager.find(User.class, primaryKey);    System.out.println(user.getEmail());  System.out.println(user.getFullname());  System.out.println(user.getPassword()); |
| --- | --- |

This code finds the user with ID = 1 in the database.

### **Execute a query:**

The following code snippet shows you how to execute a query (JPQL):

| 1  2  3  4  5  6  7 | String sql = "SELECT u from User u where u.email = 'bill.joy@gmail.com'";  Query query = entityManager.createQuery(sql);  User user = (User) query.getSingleResult();    System.out.println(user.getEmail());  System.out.println(user.getFullname());  System.out.println(user.getPassword()); |
| --- | --- |

Note that the query looks similar to traditional SQL syntax but it is not. The difference is JPQL operates against entity instances (Java objects) rather than tables in database.

### **Remove an entity instance:**

And the following code demonstrates how to delete an entity instance:

| 1  2  3 | Integer primaryKey = 1;  User reference = entityManager.getReference(User.class, primaryKey);  entityManager.remove(reference); |
| --- | --- |

As you can see, this code removes the User object with ID = 1 from the database, first by looking up a reference based on the class type (User.class) and primary key value (1), then remove the reference.

# **Spring Data JPA:**

**-** The DAO layer usually consists of a lot of boilerplate code that can and should be simplified

* Spring Data takes this simplification one step further and **makes it possible to remove the DAO implementations entirely**.
* The interface of the DAO is now the only artifact that we need to explicitly define.
* In order to start leveraging the Spring Data programming model with JPA, a DAO interface needs to extend the JPA specific *Repository* interface, *JpaRepository*.
* Spring Data JPA doesn't depend on the old ORM templates (*JpaTemplate*, *HibernateTemplate*), and they have been removed since Spring 5
* To activate the Spring JPA repository support, we can use the *@EnableJpaRepositories* annotation and specify the package that contains the DAO interfaces:

@EnableJpaRepositories(basePackages = “packagename”)

* [CrudRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html) mainly provides CRUD functions.
* [PagingAndSortingRepository](http://static.springsource.org/spring-data/data-commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html) provides methods to do pagination and sorting records.
* [JpaRepository](http://static.springsource.org/spring-data/data-jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html) provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.

**Maven Dependency**

<**dependency**>

<**groupId**>org.springframework.data</**groupId**>

<**artifactId**>spring-data-jpa</**artifactId**>

<**version**>2.4.0</**version**>

</**dependency**>

**Using Spring Boot**

<**dependency**>

<**groupId**>org.springframework.boot</**groupId**>

<**artifactId**>spring-boot-starter-data-jpa</**artifactId**>

<**version**>2.4.0</**version**>

</**dependency**>

<**dependency**>

<**groupId**>com.h2database</**groupId**>

<**artifactId**>h2</**artifactId**>

<**version**>1.4.200</**version**>

</**dependency**>

**Spring Data JPA’s Pagination APIs**

———————————————————-

Iterable<T> findAll(Sort sort);

Page<T> findAll(Pageable pageable);

* JpaRepository is a subtype of PagingAndSortingRepository so if your repository interface is of type JpaRepository, you don’t have to make change to it.

**int** pageNumber = 1;

**int** pageSize = 5;

Pageable pageable = PageRequest.of(pageNumber, pageSize);

Page<Product> page = repository.findAll(pageable);

you can get the actual content as follows:

List<Product> listProducts = page.getContent();

With a Page object you can know the total rows in the database and the total pages according to the given page size:

**long** totalItems = page.getTotalElements();

**int** totalPages = page.getTotalPages();

Service

———-

**public** Page<Product> listAll(**int** pageNum) {

**int** pageSize = 5;

Pageable pageable = PageRequest.of(pageNum - 1, pageSize);

**return** repo.findAll(pageable);

}

Controller

—————

@RequestMapping("/page/{pageNum}")

**public** String viewPage(Model model,

@PathVariable(name = "pageNum") **int** pageNum) {

Page<Product> page = service.listAll(pageNum);

List<Product> listProducts = page.getContent();

model.addAttribute("currentPage", pageNum);

model.addAttribute("totalPages", page.getTotalPages());

model.addAttribute("totalItems", page.getTotalElements());

model.addAttribute("listProducts", listProducts);

**return** "index";

}

**Dynamically build queries using JPA Criteria API/Specification**

**———————————————————————————————————**

public interface EmployeeRepository extends CrudRepository<Employee, Long>, JpaSpecificationExecutor {

}

* If the designation field is specified, search employees entities matching entities and
* If the lastName field is specified, search employees whose lastName contains the given lastName - with case-insensitive search and
* If the firstName field is specified, search employees whose firstName contains the given firstName - with case-insensitive search

public List<Employee> retrieveEmployees(Employee filter) {

List<Employee> employees = employeeRepository.findAll(new Specification<Employee>() {

@Override

public Predicate toPredicate(Root<Employee> root, CriteriaQuery< ?> query, CriteriaBuilder cb) {

List<Predicate> predicates = new ArrayList<>();

// If designation is specified in filter, add equal where clause

if (filter.getDesignation() != null) {

predicates.add(cb.equal(root.get("designation"), filter.getDesignation()));

}

// If firstName is specified in filter, add contains (lile)

// filter to where clause with ignore case

if (filter.getFirstName() != null) {

pr.add(cb.like(cb.lower(root.get("firstName")),

"%" + filter.getFirstName().toLowerCase() + "%"));

}

// If lastName is specified in filter, add contains (lile)

// filter to where clause with ignore case

if (filter.getLastName() != null) {

pr.add(cb.like(cb.lower(root.get("lastName")),

"%" + filter.getLastName().toLowerCase() + "%"));

}

return cb.and(predicates.toArray(new Predicate[0]));

}

});

}

**Hibernate Second Level Cache**

———————————————

- We can use Second Level Cache in Hibernate to optimize application performance

* First Level Cache - Hibernate Session

- Second Level Cache is associated with the SessionFactory

* When the user fetches the data from the database for the first time, the data gets stored in the Second Level Cache if it is enabled for that entity
* Data is stored in the cache in the form of key value pairs of String.
* For our example, we shall be using [Ehcache](http://www.ehcache.org/) as our Cache provider.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>5.0.0.Final</version>

</dependency>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>2.6.11</version>

</dependency>

**Entity:**

@Entity

@Table(name = "account", catalog = "test")

@Cache(usage=CacheConcurrencyStrategy.READ\_WRITE, region="account")

public class Account implements java.io.Serializable

{ }

- @Cache is used to mark the entity as cache-able.

* usage parameter tells the Hibernate which Concurrency Strategy is to be used for that particular Entity/Collection.
* Concurrency Strategy refers to the act of updating the entity once it is cached once the underlying data is modified/updated.
* Different cache Concurrency strategies in ascending order of their strictness are :

READ\_ONLY

NONSTRICT\_READ\_WRITE

READ\_WRITE

TRANSACTIONAL

- EhCache does not support Transactional Concurrency Strategy.

- region argument declares the name of the cache region in which the instances of this entity will be cached.

- By default it is the fully qualified name of the entity.

**hibernate.cfg.xml**

<property name="hibernate.cache.use\_second\_level\_cache">true</property>

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<property name="hibernate.cache.use\_query\_cache">true</property>

<property name="net.sf.ehcache.configurationResourceName">ehcache.xml</property>

* To enable the Second Level Cache, we use the property hibernate.cache.use\_second\_level\_cache and set it to true
* hibernate.cache.use\_query\_cacheproperty is used to select the underlying Cache Vendor which is EhCacheRegionFactory in our case.
* To enable the Query Cache, we use the property hibernate.cache.use\_query\_cache and set it to true.
* net.sf.ehcache.configurationResourceName is used to provide the XML filename used to configure the Ehcache.
* If this file is not provided, it is picked from the ehcache-failsafe.xml present in the ehcache-core.jar.

**ehcache.xml**

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

<ehcache xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"

xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"

monitoring="autodetect" dynamicConfig="true">

<diskStore path="java.io.tmpdir/ehcache" />

<defaultCache maxEntriesLocalHeap="10000" eternal="false"

timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"

maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"

memoryStoreEvictionPolicy="LRU" statistics="true">

<persistence strategy="localTempSwap" />

</defaultCache>

<cache name="org.hibernate.cache.internal.StandardQueryCache"

maxEntriesLocalHeap="5" eternal="false" timeToLiveSeconds="120">

<persistence strategy="localTempSwap" />

</cache>

<cache name="org.hibernate.cache.spi.UpdateTimestampsCache"

maxEntriesLocalHeap="5000" eternal="true">

<persistence strategy="localTempSwap" />

</cache>

</ehcache>

**Evicts all second level cache hibernate entites**

public void evict2ndLevelCache() {

try {

Map<String, ClassMetadata> classesMetadata = sessionFactory.getAllClassMetadata();

for (String entityName : classesMetadata.keySet()) {

logger.info("Evicting Entity from 2nd level cache: " + entityName);

sessionFactory.evictEntity(entityName);

}

} catch (Exception e) {

logger.logp(Level.SEVERE, "SessionController", "evict2ndLevelCache", "Error evicting 2nd level hibernate cache entities: ", e);

}

}

**Evict from first level cache**

//Entity is fecthed very first time

DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));

System.out.println(department.getName());

//fetch the department entity again

department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));

System.out.println(department.getName());

//Evict from first level cache

session.evict(department);

department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));

System.out.println(department.getName());

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount()); //Prints 1

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount()); //Prints 1

Output: 1 1

**Spring boot caching**

**———————————**

**@EnableCaching**

**- I**t enables Spring’s annotation-driven cache management capability.

* In spring boot project, we need to add it to the boot application class annotated with @SpringBootApplication.
* Spring provides one concurrent hashmap as default cache, but we can override CacheManager to register external cache providers as well easily.

**@Cacheable**

* It is used on the method level to let spring know that the response of the method are cacheable.

@Cacheable(value="books", key="#isbn")

public Book findStoryBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)

@Cacheable(value="books", key="#isbn.rawNumber")

public Book findStoryBook (ISBN isbn, boolean checkWarehouse, boolean includeUsed)

**@CachePut**

Sometimes we need to manipulate the cacheing manually to put (update) cache before method call. This will allow us to update the cache and will also allow the method to be executed. The method will always be executed and its result placed into the cache (according to the @CachePut options).

**@CacheEvict**

It is used when we need to evict (remove) the cache previously loaded of master data. When **CacheEvict** annotated methods will be executed, it will clear the cache.