### Hibernate

The hibernate-core.jar file is the only required Hibernate dependency.

**Hibernate**

Hibernate is one of the popular implementations of JPA.

Hibernate understands the mapping that we add between objects and tables. It ensures that data is stored/retrieved from the database based on the mapping.

Hibernate also provides additional feature on top of the JPA. But depending on them, would mean a lock to hibernate.

Hibernate is an Object Relational Mapping solution for java environments.

The term Object/Relational Mapping refers to the technique of mapping data from an object model representation to relational data model representation(and vice versa).

Hibernate not only takes care of the mapping from java classes to database tables (and from java data types to SQL data types), but also provides data query and retrieval facilities.

It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.

Hibernate can certainly help you to remove or encapsulate vendor specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects.

Hibernate, as an ORM solution, effectively sits between the Java application data access layer and the Relational Database.

The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data.

As a JPA provider, hibernate implements the Java Persistence API specifications and the association between JPA interfaces and Hibernate specific implementations.

**Object Relational Mapping (ORM)**

Databases are designed with Tables/Relations.

Java Objects are designed using OOPs.

We would want to store the data from objects into tables and vice versa.

Mapping java objects directly to database table and vice versa. This mapping is called ORM – Object Relational Mapping.

Hibernate is called an ORM framework.

**Java Persistence API (JPA)**

Java Persistence API (JPA) is the java standard for mapping java objects to a relational database.

JPA allows to map the application classes to tables in database.

JPA defines only specification. It does not provide an implementation. It is an API

* How do you define entities?
* How do you map attributes?
* How do you map relationship between entities?
* Who manages the entities?

**The Hibernate configuration file**

Hibernate uses this file to establish connection to the database server.

hibernate.cfg.xml is used to define all the hibernate related properties like dialect name, url username, password etc

The connection.driver\_class, connection.url, connection.username and connection.password <property/> elements define JDBC connection information.

**The dialect property** specifies the particular SQL variant with which Hibernate will converse.

To connect to any database with hibernate, we need to specify the SQL dialect class in hibernate.cfg.xml

hibernate.dialect=org.hibernate.dialect.Oracle10gDialect

Dialect class is java class, which contains code to map between java language data type database data type.

Dialect is used to convert HQL statements to data base specific statements.

**The hbm2ddl.auto property** enables automatic generation of database schemas directly into the database.

create: Creates schema, destroys previous data.

create-drop: Drops the schema at the end of a session.

update: Updates the schema.

validate: Validates the schema. It makes no changes to database.

Finally, add the mapping file(s) for persistent classes to the configuration. The resource attribute of the <mapping/> element causes Hibernate to attempt to locate that mapping as a classpath resource using a java.lang.ClassLoader lookup.

<mapping class="com.concretepage.Person"/>

**Hibernate mapping file is used to map java object to the database table.**

We can have any number of mapping files in a project.

Class names and property names are case sensitive in mapping file.

Table names and column names are not case sensitive in mapping file.

If we don’t give <column> tag, then property name will be column name.

If we don’t give table name, then class name will be the table name.

**you can define your mappings and queries via annotations or XML.**

This mapping is achieved through @Entity annotation in Java Bean class. Hence you need to define the mapping in the Hibernation configuration file. Here we implemented through Annotation based mapping in the java bean class.

In real time projects, developers will not define database connection details in hibernate.cfg.file.Instead they use JNDI name of data source. For example, if we want to use JNDI data source using Tomcat, the configuration will be as below in ther server.xml or you can configure in server data source configuration.

<Resource name="jdbc/MyTestDB" global="jdbc/MyTestDB" auth="Container"

type="javax.sql.DataSource" driverClassName="oracle.jdbc.driver.OracleDriver"

url="jdbc:oracle:thin:@localhost:1521/TEST" username="system" password="system"

maxActive="100" maxIdle="20" minIdle="5" maxWait="10000"/>

Hibernate.cfg.xml

<property name="**hibernate.connection.datasource**">java:comp/env/jdbc/MyTestDB</property>

If Hibernate.cfg.xml file would present on the classpath, then we don’t have to pass parameter to configure() method.

**Session Factory - org.hibernate.SessionFactory**

The SessionFactory is a thread safe object and used by all the threads of an application

A thread-safe(and immutable) representation of the mapping of the application domain model to a database.

Acts as a factory for org.hibernate.Session instances.

SessionFactory is a hibernate object which maintains data source, mappings, hibernate configuration information’s etc.,

The EntityManagerFactory is the JPA equivalent of a session factory, those two converge into the same SessionFactory implementation.

A SessionFactory is very expensive to create, so, for any given database, the application should have only one associated SessionFactory.

The SessionFactory is a heavyweight object; it is usually created during application start up and kept for later use.

The SessionFactory maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations etc.

We can create one SessionFactory implementation per database in any application. If your application is referring to multiple databases, then you need to create one SessionFactory per database.

Each database has separate hibernate configuration file.

ServiceRegistry standardRegistry = new StandardServiceRegistryBuilder().configure("hibernate1.cfg.xml").build();

SessionFactory hibernateConfig1 = configuration.buildSessionFactory(standardRegistry);

ServiceRegistry standardRegistry2 = new StandardServiceRegistryBuilder().configure("hibernate2.cfg.xml").build();

SessionFactory hibernateConfig2 = configuration.buildSessionFactory(standardRegistry2);

ServiceRegistryBuilder classes are used to build session factory through your java code.

ServiceRegistry is basically a additional layer for abstraction. And you can create the SessionFactory using ServiceRegistryBuilder

**We can choose to create the instance through a singleton design pattern.**

**Hibernate 3.0**

public static SessionFactory getSessionFactory() {

SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();

return sessionFactory;

}

**Hibernate 4.0,4.1 or 4.2**

public static SessionFactory getSessionFactory() {

ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().configure(). buildServiceRegistry();

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

return sessionFactory;

}

**Hibernate 4.3 serviceRegistry is depreciated**

public static SessionFactory getSessionFactory() {

ServiceRegistry standardRegistry = new StandardServiceRegistryBuilder().configure().build();

SessionFactory sessionFactory = configuration.buildSessionFactory(standardRegistry);

return sessionFactory;

}

**Session - org.hibernate.session**

A single-threaded, short-lived object conceptually modelling a "Unit of work".

In JPA, the session is represented by an EntityManager.

Behind the scenes, the Hibernate session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances.

It maintains a generally "repeatable read" persistence context(first level cache) of the application domain model.

* Unlike SessionFactory, the Session object will be created on demand. Session is a lightweight object.
* Session provides a physical connectivity between your application and database. The Session will be established each time your application wants do something with database.
* Session object will be provided by SessionFactory object. All the persistent objects will be saved and retrieved through Session object. The session object must be destroyed after using it.

The lifecycle of a Session is bounded by the beginning and end of a logical transaction. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of three states:

transient: never persistent, not associated with any Session.

persistent: associated with a unique Session.

detached: previously persistent, not associated with any Session.

**Persistence Object has three life cycle states**

Transient state: Object is neither associated with Session nor present in Database.

Persistent state: Object is associated with Session and also present in Database.

Detached state: Object is not associated with Session, but present in Database.

Employee emp = new Employee();

emp.setEmpId(1003);

emp.setEmpName(“Sreenath”);

emp.setEmpSalaray(“30000”);

Here emp object is not associated with session and there is no matching record in the Employee table. **So emp object is in TRANSIENT**.

Employee emp = (Employee) session.get(Employee.class,1001);

**Here emp object is associated with session and there is matching record in EMPLOYEE. So emp object is in PERSISTENET state**.

Employee emp = new Employee();

emp.setEmpId(1003);

emp.setEmpName(“Sreenath”);

emp.setEmpSalaray(“30000”);

//Here emp object is in Transient state.

session.save(emp);

//Here emp object is in Persistent state.

session.close();

//Here emp object is in Detached state.

Here **emp object becomes Detached state** **from persistent state after calling session.close().**

Changes made to object will not be stored into database, since session is closed (garbage collected).

**Difference between session.get() vs session.load() in hibernate ?**

Both are used to retrieve the objects from the database.

session.load(Author.class,2);

session.get(Author.class,2);

Author is the hibernate mapping class and Integer value 2 will be passed to the primary key field of the Author class and respective objects will be retrieved from the database.

**Differences**:

1. session.load() will **return the hibernate proxy object**, but session.get() **will return real object**.

2. session.load() **will not hit the database**, but session.get() **will hit the database all the time**.

3. session.load() **will throw object not found exception when no row found**, but session.get() **will return null when no row found in db**.

|  |  |
| --- | --- |
| **Load () method** | **Get () method** |
| Load () method throws hibernate.ObjectNotFoundException (which is Un-checked exception) if object not found in cache as well as in database. | Get () method returns NULL, if object is not found in cache as well as in database. |
| Load () method is lazy loading i.e. when you call session.load(Class,identifier) method, it will return Proxy object but not entity | Get () method is early loading e. when you call session.get(Class,identifier) method, it will hit the database immediately ,loads the entity object and returns the entity object. |
| Use load () method when you are sure that the object exists in database. | Use get () method if you are not sure that the object exists in database |
| Load () method is equivalent to getReference() method of JPA. | It is equivalent to EntityManager.find() method of JPA. |
| Employee emp = session.load(Employee.class,1);  //returns the employee proxy object  S.o.p(emp.getName());  //employee is initialized after the method called. | Employee emp = session.get(Employee.class,1);  You will be seeing select statement and employee object is initialized with database data. |

**SaveOrUpdate**

* If the record is not present in the database, it will call save() method and inserts the record in the database.
* If the record is present in the database, it will call update() method and updates the record in the database.

public void saveOrUpdate(Object object) throws HibernateException

Employee saveEmp = new Employee(1,"Kalyan","Developement");

session.saveOrUpdate(saveEmp);

**Persist () and Save () method**

**Persist** () is equivalent to Save () but same method returns the identifier and persist method returns nothing.

public void persist(Object object) throws HibernateException

**Use Save () method**, if you are using generator class to generate the identifier and you want to know the identifier value.

**Use persist () method**, if you don’t want to know the generated identifier.

**Replicate () method**

Replicate () method used to move the object from detached state to persistence state.

public void replicate(Object obj, ReplicationMode replicationMode) throws HibernateException

Session.replicate(emp, **ReplicationMode.LATEST\_VERSION**);

<https://www.connect2java.com/tutorials/hibernate/flush-method-in-hibernate/>

**Flush ()**

* Forces the session to flush. It is used to synchronize session data with database.

public void flush() throws HibernateException

* When you call session.flush(), the statements are executed in database, but it will not be committed.

If you don’t call session.flush () and if you call session.commit() , internally commit() method executes the statement and commits.

So commit()= flush+commit.

* So session.flush() just executes the statements in database (but not commits) and statements are NOT IN MEMORY anymore. It just forces the session to flush.
* After session.flush(), hibernate compares employee object data and corresponding record in database. If there is a difference it will execute update query to update object data in the database, but it will not commit.
* After transaction.commit(),  Here also , hibernate compares employee object data and corresponding record in database. If there is a difference it will execute update query to update object data in the database and commits transaction.
* session.flush() must be called before committing the transaction and closing the session.
* session.flush() : this function is used to serialize the data to the database. If you have something in the session that is not submitted to DB, you can submit them using session.flush().

<https://www.connect2java.com/tutorials/hibernate/clear-evict-and-close-methods-in-hibernate/>

**Clear ()**

* Completely clear the session and is used to dissociate/disconnect all the objects from the session.

public void clear();

session.clear();

**evict ()**

* Removes the object from the session. This method is used to dissociate/disconnect the specified object from the session.

public void evict(Object object) throws HibernateException

session.evict (emp1);

**close ()**

* close the session by calling session.close () after transaction is completed. All the associated objects will be dissociated after calling session.close().

public Connection close() throws HibernateException

It is not strictly necessary to close the session, but you must at least using disconnect it using session.disconnect ()

**contains(Object object)**

* Used to check whether the instance is associated with the session or not.

public boolean contains(Object object)

if(session.contains(emp))

                System.out.println("emp instance is associated with session");

            else

                System.out.println("emp instance is NOT associated with session");

**isConnected()**

* Used to check whether session is currently connected or not.

Public boolean isConnected()

if(session.isConnected())

            System.out.println(“Session is conncted”);

        else

            System.out.println(“Session is not conncted”);

**getIdentifier()**

* Use to find the object identifier value at run-time.

public Serializable getIdentifier(Object object) throws HibernateException

Employee emp = (Employee)session.get(Employee.class, 1);

Serializable id = session.getIdentifier(emp);

System.out.println("EMP object identifer is "+ id);

**Refresh ()**

* It is used to synchronize database data with session data.

public void refresh(Object object) throws HibernateException

session.refresh (employee);

    System.out.println("After updating database");

    System.out.println("EMP NAME :" + employee.getEmpName());

    System.out.println("EMP DEPT :" + employee.getEmpDept());

<https://www.connect2java.com/tutorials/hibernate/merge-method-in-hibernate/>

**Merge ()**

It merge the detached object data into persistent data.

merge() method called, it will check whether there is any object associated with session with identifier.

Session.merge(employee);

If any object is already associated with session, so merge() method copy the new state(emp2) into employee1 object.

Employee emp1 = (Employee)session.get (Employee.class, 1);

Employee emp2 = new Employee(1,"SAINATH","R&D");

session.merge (emp2);

update () - Use update (), when a session does not contain the persistent instance with the same identifier.  
merge () - **Use merge (),** **Irrespective of the state of a session, if you need to save the modifications at any given time**.

**Hibernate Query object**

<https://www.codejava.net/frameworks/hibernate/hibernate-query-language-hql-example>

* The Hibernate Query object is used to retrieve data from database.
* You can use either SQL or Hibernate Query Language (HQL). A Query instance is obtained by calling Session.createQuery().

The Query object is used to bind query parameters, limit query results and execute the query.

list()

Return the query results as a List. If the query contains multiple results pre row, the results are returned in an instance of Object[].

Session session = getSessionFactory().openSession();

List employees = session.createQuery("FROM Employee").list();

**Transaction - org.hibernate.Transaction**

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries.

Transaction object can be created by 2 ways

1) Transaction transaction = session.beginTransaction();

2) Transaction transaction = session.getTransaction();

Transaction tx = session.beginTransaction();

tx.commit();

**getTransaction():**

Returns the Transaction object **associated with current session**. It **returns the same Transaction object** for every request.

public Transaction getTransaction() throws HibernateException

**We need to call begin() method on Transaction object to start the transactio**n.

Transaction transaction = session.getTransaction();

transaction.begin();

**beginTransaction()**

Returns different transaction objects. For each and every request transaction object creates.

Transaction transaction = session.beginTransaction();

**Important points regarding Transaction Object**

1. It is not thread -safe that means any thread can access this object. So, it’s better to close session after finishing the operation.

2. We can have single commit for multiple operation within a single session. So, no need to create multiple Transaction object for each operation within a session.

3. Must commit the database operation otherwise the operation doesn't reflect the changes in the database table and hence hibernate unable to perform the operation.

**Domain class annotations**

@Entity denotes the class is an entity i.e. persistence POJO class. Entity is a lightweight persistent domain object.

@Table denotes the database table to which this entity is mapping. If no @Table is defined the default values are used.

@Id denotes the primary key of the entity.

@GeneratedValue denotes the strategy of generating the primary key. The default strategy is the AUTO strategy.

@Column denotes the column mapping of entity attribute.

The column(s) used for a property mapping can be defined using the @Column annotation. You can use this annotation at the property level.

@JoinColum denotes the foreign key column.

@OnDelete denotes the cascade delete action. In this example, When the customer entity gets deleted, all its accounts will be removed at the same time.

The @JoinColumn annotation combined with a @OneToOne mapping indicates that a given column in the owner entity refers to a primary key in the reference entity.

@Entity

public class Office {

@OneToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "addressId")

private Address address;

}

When using a @OneToMany mapping we can use the mappedBy parameter to indicate that the given column is owned by another entity.

@Entity

public class Employee {

@Id

private Long id;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")

private List<Email> emails;

}

@Entity

public class Email {

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "employee\_id")

private Employee employee;

}

@JoinColumns

In situations when we want to create multiple join columns we can use the @JoinColumns annotation:

@Entity

public class Office {

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumns({

@JoinColumn(name="ADDR\_ID", referencedColumnName="ID"),

@JoinColumn(name="ADDR\_ZIP", referencedColumnName="ZIP")

})

private Address address;

}

**@GeneratedValue**

* @GeneratedValue denotes the strategy of generating the primary key. The default strategy is the AUTO strategy.
* If the values of the primary column are auto-increment, we need to use this annotation to tell Hibernate knows, along with one of the following strategy types: AUTO, IDENTITY, SEQUENCE, and TABLE.

**@GeneratedValue with the AUTO strategy:**

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

int id;

**AUTO is the default strategy, so the following definition is equivalent**:

@Id

@GeneratedValue

int id;

**we use the strategy IDENTITY** which specifies that the generated values are unique at table level,

whereas **the strategy AUTO** implies that the generated values are unique at database level.

**During a commit the AUTO strategy** uses the global number generator to generate a primary key for every new entity object. These generated values are unique at the database level.

**The IDENTITY strategy** also generates an automatic value during commit for every new entity object. The difference is that a separate identity generator is managed per type hierarchy, so generated values are unique only per type hierarchy.

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

int id;

Suppose you have two tables namely store and coupon, store id is a primary key in store table and foreign key in coupon table.

Store table: store\_id | store\_name | store\_url

Coupon table: coupon\_id | coupon\_name | store\_id

@Entity

public class Coupon{

public Coupon(){

}

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "coupon\_id")

private Long couponId;

}

Unlike AUTO and IDENTITY, the **SEQUENCE strategy** generates an automatic value as soon as a new entity object is persisted (i.e. **before commit**). This may be useful when the primary key value is needed earlier.

@SequenceGenerator(name=”seq”,sequenceName = "coupon\_seq" initialValue=1, allocationSize=100)

public class UserDetails {

// Use the sequence that is defined above:

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

@Id int id;

}

**The TABLE strategy** is very similar to the SEQUENCE strategy:

@Entity

@TableGenerator(name=”tab”, initialValue=0, allocationSize=50)

public class UserDetails {

@GeneratedValue(strategy=**GenerationType.TABLE**, generator=”tab”)

@Id int id;

**@Embeddable and @Embedded**

The way to tell hibernate not to create a new table for this class is by using @Embeddable annotation on the Address class.

so, add the attribute Address in your UserDetails class and then annotate it with @Embedded

@Entity

class UserDetails {

@Column (name="USER\_NAME")

private String name;

@Embedded

private Address address;

}

@Embeddable

public class Address {

private String street;

private String city;

private String state;

private String pincode;

}

**@Lob**

* @Lob stands for Large Object and it can be either BLOB(Binary Large Object) or CLOB(Character Large Object).
* If the data is of the char[] or String type then the persistent provider maps the data to a CLOB column, otherwise the data is mapped to BLOB column.

**@Temporal**

* The DATE, TIME and TIMESTAMP can be used to map a temporal type. If no values is specified for the TemporalType parameter of the @Temporal annotation, the TIMESTAMP is selected as default value.

**@MapsId**

* @MapsId defines the embedded primary key.

**FETCHING STRATEGIES**

So, these are the fetching strategies in hibernate :

**join fetching(**fetch="join") : In this case Hibernate will retrieves all the information in a single select statement.

**select fetching**(fetch="select") : Here the Hibernate will pass the second select statement for fetching the associated collection unless you disable lazy fetching by specifying lazy="false".

**subselect fetching**(fetch="subselect") : Here in this case the collection will be fetched using sub select SQL statement unless you explicitly disable lazy fetching using lazy="false";

**batch fetching**(batch-size="") : As the name suggest if you want to customize the number of select statement you can use batch fetching. It can be configured on both class and collection label.

**immediate fetching**(lazy="false") ; Hibernate will load all the associated child collection with the parent.

**lazy collection fetching**(lazy="true") : here the associated collection will be loaded only when you use it, or you can perform any operation on it(one of the best practice to tune the performance of your application).

**extra lazy** : specific element of the collection is accessed as needed. Here in this case hibernate will not load the whole collection unless you tried. It is suitable for large collection.

proxy fetching, No-proxy, Lazy attribute fetching these 3 are also the fetching strategies that is rarely used. Mostly these three are used in lazy applications.

**FetchType.LAZY**: It fetches the child entities lazily, that is, at the time of fetching parent entity it just fetches proxy (created by cglib or any other utility) of the child entities and when you access any property of child entity then it is actually fetched by hibernate.

**FetchType.EAGER**: it fetches the child entities along with parent.

**Lazy initialization** improves performance by avoiding unnecessary computation and reduce memory requirements.

**Eager initialization** takes more memory consumption and processing speed is slow.

**One-to-one Directional relationship**

A [row](https://database.guide/what-is-a-row/) in [table](https://database.guide/what-is-a-table/) A can have only one matching row in table B, and vice versa.

1. Uni directional
2. Bidirectional
3. Shared primary keys

In bi-directional relationship, we specify @OneToOne annotation on the both the entities but only one entity is the owner of the relationship.

Most often child entity is the owner of the relationship and the parent entity is the inverse side of the relationship.

OneToOne Mapping

|  |
| --- |
| User |
| User\_id  Firstname  Lastname  email |

|  |
| --- |
| UserProfile |
| User\_profile\_id  User\_id  Address1  Address2  City  Country |

|  |  |
| --- | --- |
| CREATE TABLE user (  user\_id long(11) NOT NULL AUTO\_INCREMENT,  firstname varchar(128) NOT NULL,  lastname varchar(512) NOT NULL,  email varchar(128) NOT NULL,  PRIMARY KEY (user\_id)  ); | CREATE TABLE user\_profile (  user\_profile\_id long(11) NOT NULL AUTO\_INCREMENT,  user\_id long(11) NOT NULL,  address1 varchar(128) NOT NULL,  address2 varchar(512) NOT NULL,  city varchar(128) NOT NULL,  country varchar(128) NOT NULL,  PRIMARY KEY (user\_profile\_id),  KEY up\_fk (user\_id),  CONSTRAINT up\_fk FOREIGN KEY (user\_id) REFERENCES user (user\_id)  ); |
| Bi-Directional  public class User{  @Id  @Column(name="user\_id")  @GeneratedValue  long id;  String firstname;  String lastname;  String email;  @OneToOne(mappedBy="user", fetchType=FetchType.LAZY, cascade=CascadeType.ALL)  private UserProfile userProfile;  } | public class UserProfile {  @Id  @column(name="user\_profile\_id")  @GeneratedValue  private long id;  private String address1;  private String address2;  private String city;  private String country;  @OneToOne(cascade=CascadeType.ALL)  @JoinColumn(name="user\_id")  private User user;  } |
| Uni-Directional  public class User{  @Id  @Column(name="user\_id")  @GeneratedValue  long id;  String firstname;  String lastname;  String email;  } | public class UserProfile {  @Id  @column(name="user\_profile\_id")  @GeneratedValue  private long id;  private String address1;  private String address2;  private String city;  private String country;  @OneToOne(cascade=CascadeType.ALL)  @JoinColumn(name="user\_id")  private User user;  } |

**Unidirectional – OneToOne – One more example**

|  |
| --- |
| User |
| User\_id  Firstname  Lastname  Email  **User\_profile\_id** |

|  |
| --- |
| UserProfile |
| User\_profile\_id  Address1  Address2  City  Country |

|  |  |
| --- | --- |
| CREATE TABLE user (  user\_id long(11) NOT NULL AUTO\_INCREMENT,  firstname varchar(128) NOT NULL,  lastname varchar(512) NOT NULL,  email varchar(128) NOT NULL,  user\_profile\_id long(11) NOT NULL  PRIMARY KEY (user\_id),  KEY up\_fk (user\_id),  CONSTRAINT up\_fk FOREIGN KEY (user\_profile\_id) REFERENCES user\_profile(user\_profile\_id)  ); | CREATE TABLE user\_profile (  user\_profile\_id long(11) NOT NULL AUTO\_INCREMENT,  address1 varchar(128) NOT NULL,  address2 varchar(512) NOT NULL,  city varchar(128) NOT NULL,  country varchar(128) NOT NULL,  PRIMARY KEY (user\_profile\_id),    ); |
| public class User{  @Id  @Column(name="user\_id")  @GeneratedValue  long id;  String firstname;  String lastname;  String email;  @Column(name="user\_profile\_id")  private Long userProfileId;  } | public class UserProfile {  @Id  @column(name="user\_profile\_id")  @GeneratedValue  private long id;  private String address1;  private String address2;  private String city;  private String country;  @OneToOne(cascade=CascadeType.ALL)  @JoinColumn(name="user\_id")  private User user;  } |

@OneToOne and @JoinColumn: are used together to specify a one-to-one association and the join column.

you can see @JoinColumn annotation that adds metadata about foreign key column.

**@OneToOne** defines a one-to-one relationship between 2 entities.

**@JoinColumn** defines foreign key column and indicates the owner of the relationship.

**mappedBy** indicates the inverse of the relationship.

**SharedPrimaryKeys – OneToOne**

|  |
| --- |
| User |
| User\_id  Firstname  Lastname  Email |

|  |
| --- |
| UserProfile |
| User\_id  Address1  Address2  City  Country |

|  |  |
| --- | --- |
| CREATE TABLE user (  user\_id long(11) NOT NULL AUTO\_INCREMENT,  firstname varchar(128) NOT NULL,  lastname varchar(512) NOT NULL,  email varchar(128) NOT NULL,  PRIMARY KEY (user\_id)  ); | CREATE TABLE user\_profile (  user\_id long(11) NOT NULL,  address1 varchar(128) NOT NULL,  address2 varchar(512) NOT NULL,  city varchar(128) NOT NULL,  country varchar(128) NOT NULL,  PRIMARY KEY (user\_id)  ); |
| public class User{  @Id  @Column(name="user\_id")  @GeneratedValue  long id;  String firstname;  String lastname;  String email;  } | public class UserProfile {  @OneToOne(cascade=CascadeType.ALL)  @JoinColumn(name="user\_id")  @MapsId  private long id;  private String address1;  private String address2;  private String city;  private String country;  } |

**MapsId defines embedded primary key**, user\_profile.user\_id is embedded from user.id.

**One-to-many Directional relationship**

A row in table A can have many matching rows in table B, but a row in table B can have only one matching row in table A.

1. Uni directional
2. Bidirectional

**One publisher to many books**

|  |  |
| --- | --- |
| Publisher  Id;  Name; | Book  Id;  Name;  Publisher\_id |
| CREATE TABLE publisher (  Publisher\_id long(11) NOT NULL AUTO\_INCREMENT,  Name varchar(128) NOT NULL,  PRIMARY KEY (Publisher\_id)  ); | CREATE TABLE book (  Book\_id long(11) NOT NULL AUTO\_INCREMENT,  name varchar(128) NOT NULL,    PRIMARY KEY (book\_id),  KEY pub\_fk (publisher\_id),  CONSTRAINT pub\_fk FOREIGN KEY (publisher\_id) REFERENCES publisher (publisher\_id)  ); |
| **Bi-Directional**  @Entity  public class Publisher {    @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column (name="PUBLISHER\_ID",unique=true)  private int publisherId;    private String publisherName;    @OneToMany(mappedBy = "publisher")  private Set<Book> books = new HashSet<Book>();  } | @Entity  public class Book {  @Id  @Column(name="book\_id")  @GeneratedValue  private int bookId;  private String title;    @ManyToOne  @JoinColumn(name = "PUBLISHER\_ID")  private Publisher publisher;  } |
| **Uni-Directional**  @Entity  public class Publisher {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column (name="PUBLISHER\_ID",unique=true)  private int publisherId;    private String publisherName;    @OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)  @JoinColumn(name="PUBLISHER\_ID")  private Set<Book> books = new HashSet<Book>();  } | @Entity  public class Book {  @Id  @Column(name="book\_id")  @GeneratedValue  private int bookId;  private String title;    @Column(name="PUBLISHER\_ID")  private int publisherId;  } |

**One to Many – with 3 tables**

|  |  |  |
| --- | --- | --- |
| Publisher  Id;  Name; | Book  Id;  Name; | Publisher\_Book  Publisher\_id  Book\_id |
| CREATE TABLE publisher (  Publisher\_id long(11) NOT NULL AUTO\_INCREMENT,  Name varchar(128) NOT NULL,  PRIMARY KEY (Publisher\_id)  ); | CREATE TABLE book (  Book\_id long(11) NOT NULL AUTO\_INCREMENT,  name varchar(128) NOT NULL,  PRIMARY KEY (book\_id)    ); | CREATE TABLE publisher\_book (  publisher\_id long(11) NOT NULL,  Book\_id long(11) NOT NULL ,  PRIMARY KEY (publisher\_id,book\_id),  KEY pub\_fk (publisher\_id),  KEY book\_fk (book\_id),  CONSTRAINT pub\_fk FOREIGN KEY (publisher\_id) REFERENCES publisher (publisher\_id),  CONSTRAINT book\_fk FOREIGN KEY (book\_id) REFERENCES book (book\_id)  ); |
| @Entity  public class Publisher {    @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column (name="PUBLISHER\_ID",unique=true)  private int publisherId;    private String publisherName;    @OneToMany(cascade = CascadeType.ALL)  @JoinTable(  name = "publisher\_book",  joinColumns = @JoinColumn(name = "publisher\_id"),  inverseJoinColumns = @JoinColumn(name = "book\_id")  )  private Set<Book> books = new HashSet<Book>();      } | @Entity  public class Book {  @Id  @Column(name="book\_id")  @GeneratedValue  private int bookId;  private String title;    } |  |

Here, we use the **@JoinTable** annotation to specify the details of the join table (table name and two join columns - using the **@JoinColumn**annotation); and we set the **cascade** attribute of the **@OneToMany** annotation so that Hibernate will update the associated articles when the category is updated.

The @JoinTable annotation is used to create the publisher\_book link table and @JoinColumn annotation is used to refer the linking columns in both the tables.

**Many-to-Many Directional relationship**

In a many-to-many relationship, a row in table A can have many matching rows in table B, and vice versa.

A many-to-many relationship could be thought of as two one-to-many relationships, linked by an intermediary table.

The intermediary table is typically referred to as a “junction table” (also as a “cross-reference table”). This table is used to link the other two tables together. It does this by having two fields that reference the [primary key](https://database.guide/what-is-a-primary-key/) of each of the other two tables.

**Many publisher to many books**

|  |  |  |
| --- | --- | --- |
| **Publisher**  Id;  Name; | **Book**  Id;  Name; | **Publisher\_Book**  Publisher\_id  Book\_id |
| CREATE TABLE publisher (  Publisher\_id long(11) NOT NULL AUTO\_INCREMENT,  Name varchar(128) NOT NULL,  PRIMARY KEY (Publisher\_id)  ); | CREATE TABLE book (  Book\_id long(11) NOT NULL AUTO\_INCREMENT,  name varchar(128) NOT NULL,    PRIMARY KEY (book\_id)    ); | CREATE TABLE publisher\_book (  publisher\_id long(11) NOT NULL,  Book\_id long(11) NOT NULL ,  PRIMARY KEY (publisher\_id,book\_id),  KEY pub\_fk (publisher\_id),  KEY book\_fk (book\_id),  CONSTRAINT pub\_fk FOREIGN KEY (publisher\_id) REFERENCES publisher (publisher\_id),  CONSTRAINT book\_fk FOREIGN KEY (book\_id) REFERENCES book (book\_id)  ); |
| **Bi-Directional**  @Entity  public class Publisher {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column (name="PUBLISHER\_ID",unique=true)  private int publisherId;  private String publisherName;  @ManyToMany(cascade = CascadeType.ALL)  @JoinTable(  name = "publisher\_book",  joinColumns = @JoinColumn(name = "publisher\_id"),  inverseJoinColumns = @JoinColumn(name = "book\_id")  )  private Set<Book> books = new HashSet<Book>();  } | @Entity  public class Book {  @Id  @Column(name="book\_id")  @GeneratedValue  private int bookId;  private String title;    @ManyToMany(mappedBy = "books")  private Set<Publisher> publishers = new HashSet<publlisher>();  } |  |
| **Uni-Directional**  Exactly same as Bi-Directional source code | @Entity  public class Book {  @Id  @Column(name="book\_id")  @GeneratedValue  private int bookId;  private String title;  } |  |

 this association is bidirectional. Here, the Publisher is the owner side and the book is the other side.

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.

**Hibernate N+1 SELECT’s Problems And Solution**

Hibernate n+1 problems only comes for one to many relationship.

Let say one example – Retailer with a one-to-many relationship with Product. One Retailer has many Products.

|  |  |
| --- | --- |
| \*\*\*\*\* Table: Retailer \*\*\*\*\*  +—–+——————-+  | ID | NAME |  +—–+——————-+  | 1 | Retailer Name 1 |  | 2 | Retailer Name 2 |  | 3 | Retailer Name 3 |  | 4 | Retailer Name 4 |  +—–+——————-+ | \*\*\*\*\* Table: Product \*\*\*\*\*  +—–+———–+——————–+——-+————+  | ID | NAME | DESCRIPTION | PRICE | RETAILERID |  +—–+———–+——————–+——-+————+  |1 | Product 1 | Name for Product 1 | 82.0 | 4 |  |2 | Product 2 | Name for Product 2 | 20.0 | 2 |  |3 | Product 3 | Name for Product 3 | 10.0 | 2 |  |4 | Product 4 | Name for Product 4 | 77.0 | 3 | |

**Factors**:

Lazy mode for Retailer set to “true” (default)

Fetch mode used for querying on Product is Select

Fetch mode (default): Retailer information is accessed

Caching does not play a role for the first time the Retailer is accessed

Fetch mode is Select Fetch (default)

// It takes Select fetch mode as a default

Query query = session.createQuery( "from Product p");

List list = query.list();

// Retailer is being accessed

displayProductsListWithRetailerName(results);

select \* from PRODUCT

select \* from RETAILER where RETAILER.id=?

select \* from RETAILER where RETAILER.id=?

select \* from RETAILER where RETAILER.id=?

**Result:**

1 select statement for Product

N select statements for RETAILER

This is N+1 select problem!

**Solution to N+1 SELECTs problem**

(i) **HQL fetch join**

“from RETAILER retailer join fetch retailer.product Product”

Corresponding SQL would be –

SELECT \* FROM RETAILER retailer LEFT OUTER JOIN PRODUCT product ON product.product\_id=retailer.product\_id

(ii) **Criteria query**

Criteria criteria = session.createCriteria(Retailer.class);

criteria.setFetchMode(“product”, FetchMode.EAGER);

In both cases, the query returns a list of Retailer objects with the Product initialized. Only one query needs to be run to return all the Product and Retailer information required.

**Hibernate Caching**

**Caching is nothing but some buffer where a record is stored when first time retrieved from the database.** When second time needed the same record, hibernate does not access the database and instead reads from the **cache**. This type of adjustment decreases the database hits. This is what Cache Hibernate is. Accessing cache is much faster than accessing the database.

To increase the database access performance, hibernate comes with **caching mechanism** to give **high performance.**

Hibernate caching is used to optimize application performance; it minimizes the DB connection cost thus reduces data fetch time. First time it will fetch data from DB and load objects then these objects will be stored in cache(like in files), on request from next time onwards it will pick from cache. So, it is very faster.

Both caches help you to reduce the number of executed SQL statements by storing entities in local memory. This can provide huge performance improvements if you have to read the same entity multiple times within the same or multiple transactions.

**First-level cache**

As the first level cache is associated with the **Session**object, its scope is limited to one session only.

It is a session scoped cache which ensures that each entity instance is loaded only once in the persistent context. Once the session is closed, first-level cache is terminated as well.

The [first-level cache](https://thoughts-on-java.org/free-sample-lecture-hibernate-1st-level-cache/), which contains all entities used within a transaction.

Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. Objects will be cached within the same session. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

As the first level cache is associated with the **Session**object, its scope is limited to one session only. If we fetch a record (better call as persistent object in Hibernate), say with [get()](http://way2java.com/hibernate/hibernate-get-load-delete-and-update-methods/) method, from the database, it is stored with the Session cache. If the same record is fetched again second time, database hit is not made as it gets from the **Session** cache.

If you fetch the same record with another **Session**object, database hit is made.

**second-level cache**

Second-level cache always associates with the **SessionFactor**y object

SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitly.

you can use any third-party cache with Hibernate. An org.hibernate.cache.CacheProvider interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

second-level cache is SessionFactory-scoped, meaning it is shared by all sessions created with the same session factory.

 The second level cache is responsible for caching objects across sessions. When this is turned on, objects will first be searched in the cache and if they are not found, a database query will be fired.

Second level cache will be used when the objects are loaded using their primary key.

Second-level cache always associates with the **SessionFactor**y object. While running the transactions, whatever records (or persistent objects) the session object fetches from the database are preserved in **SessionFactory cache (buffer)**. These records are available not only to the current Session object but all Session objects created from SessionFactory object. Any Session object requires the same record again, the database is not hit and instead reads from the **SessionFactory cache**. This we can prove later in the client program.

[**https://way2java.com/hibernate/hibernate-first-level-and-second-level-cache-examples/**](https://way2java.com/hibernate/hibernate-first-level-and-second-level-cache-examples/)

In this article we use Ehcache as a cache provider, which is a mature and widely used cache.

We add the Ehcache region factory implementation to the classpath with the following Maven dependency:

|  |  |
| --- | --- |
|  | <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-ehcache</artifactId>      <version>5.2.2.Final</version>  </dependency> |

**Enabling Second-Level Caching**

With the following two properties we tell Hibernate that L2 caching is enabled and we give it the name of the region factory class:

|  |  |
| --- | --- |
|  | hibernate.cache.use\_second\_level\_cache=true  hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory  **Making an Entity Cacheable**  In order to make an entity eligible for second-level caching, we annotate it with Hibernate specific @org.hibernate.annotations.Cache annotation and specify a cache concurrency strategy.  Some developers consider that it is a good convention to add the standard @javax.persistence.Cacheable annotation as well is:  @Entity  @Cacheable  @org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)  public class Foo { } |

**Query Cache**

* Query Cache is used to cache the results of a query.
* Results of HQL queries can also be cached. This is useful if you frequently execute a query on entities that rarely change.
* Query cache is disabled by default. You have to enable explicitly if required.
* Whenever you execute the hibernate Queries HQL, QBC etc then all records returned by select statement will be placed in Query cache.

To enable query cache, set the value of hibernate.cache.use\_query\_cache property to true:

|  |  |
| --- | --- |
|  | hibernate.cache.use\_query\_cache=true |

**Design Patterns used in the Hibernate**

|  |  |
| --- | --- |
| Singleton Pattern | SessionFactory |
| Proxy pattern | Session.load() |
| Builder Pattern | QueryBuilder |

**Inheritance Strategies**

https://thoughts-on-java.org/complete-guide-inheritance-strategies-jpa-hibernate/

JPA and Hibernate support 4 inheritance strategies which map the domain objects to different table structures.

**Mapped Superclass**

The mapped superclass strategy is the simplest approach to mapping an inheritance structure to database tables. It maps each concrete class to its own table.

That allows you to share the attribute definition between multiple entities. But it also has a huge drawback. A mapped superclass is not an entity, and there is no table for it.

|  |  |
| --- | --- |
| @MappedSuperclass  public abstract class Publication {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column(name = “id”, updatable = false, nullable = false)  protected Long id;  @Column  protected String title;  @Version  @Column(name = “version”)  private int version;  @Column  @Temporal(TemporalType.DATE)  private Date publishingDate;  …  } | @Entity(name = “Book”)  public class Book extends Publication {  @Column  private int pages;  …  }  @Entity(name = “BlogPost”)  public class BlogPost extends Publication {  @Column  private String url;  …  } |

**Table per Class**

The table per class strategy is similar to the mapped superclass strategy. The main difference is that the superclass is now also an entity.

The definition of the superclass with the table per class strategy looks similar to any other entity definition. You annotate the class with @Entity and add your mapping annotations to the attributes. The only difference is the additional @Inheritance annotation which you have to add to the class to define the inheritance strategy. In this case, it’s the InheritanceType.TABLE\_PER\_CLASS.

|  |  |
| --- | --- |
| @Entity  @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  public abstract class Publication {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  @Column(name = “id”, updatable = false, nullable = false)  protected Long id;  @Column  protected String title;  @Version  @Column(name = “version”)  private int version;  @ManyToMany  @JoinTable(name = “PublicationAuthor”, joinColumns = { @JoinColumn(name = “publicationId”, referencedColumnName = “id”) }, inverseJoinColumns = { @JoinColumn(name = “authorId”, referencedColumnName = “id”) })  private Set authors = new HashSet();  @Column  @Temporal(TemporalType.DATE)  private Date publishingDate;  …  } | @Entity(name = “Book”)  public class Book extends Publication {  @Column  private int pages;  …  }  @Entity(name = “BlogPost”)  public class BlogPost extends Publication {  @Column  private String url;  …  } |

The superclass is now also an entity and you can, therefore, use it to define a relationship between the Author and the Publication entity. This allows you to call the getPublications() method to get all Publications written by that Author. Hibernate will map each Publication to its specific subclass.

List authors= em.createQuery(“SELECT a FROM Author a”, Author.class).getResultList();

for (Author a : authors) {

for (Publication p : a.getPublications()) {

if (p instanceof Book)

log(p.getTitle(), “book”);

else

log(p.getTitle(), “blog post”);

}

}

**Single Table**

The single table strategy maps all entities of the inheritance structure to the same database table. This approach makes polymorphic queries very efficient and provides the best performance.

But it also has some drawbacks. The attributes of all entities are mapped to the same database table. Each record uses only a subset of the available columns and sets the rest of them to null. You can, therefore, not use not null constraints on any column that isn’t mapped to all entities. That can create data integrity issues, and your database administrator might not be too happy about it.

@Entity

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = “Publication\_Type”)

public abstract class Publication {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = “id”, updatable = false, nullable = false)

protected Long id;

}

**Joined**

The joined table approach maps each class of the inheritance hierarchy to its own database table. This sounds similar to the table per class strategy. But this time, also the abstract superclass Publication gets mapped to a database table. This table contains columns for all shared entity attributes. The tables of the subclasses are much smaller than in the table per class strategy. They hold only the columns specific for the mapped entity class and a primary key with the same value as the record in the table of the superclass.

@Entity

@Inheritance(strategy = InheritanceType.JOINED)

public abstract class Publication {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = “id”, updatable = false, nullable = false)

protected Long id;

}