# HIBERNATE

***is an ORM tool)***

- Mapping the objects in Java directly to the relational entities in a database.
- Hibernate Query Language(HQL) which makes it database-independent.
- It supports auto DDL operations.
- This Java framework also has an Auto Primary Key Generation support.
- Supports cache memory.
- Validator provides the reference implementation of bean validation specs.
- Exception handling is not mandatory in the case of Hibernate.
- It is open source and therefore free.

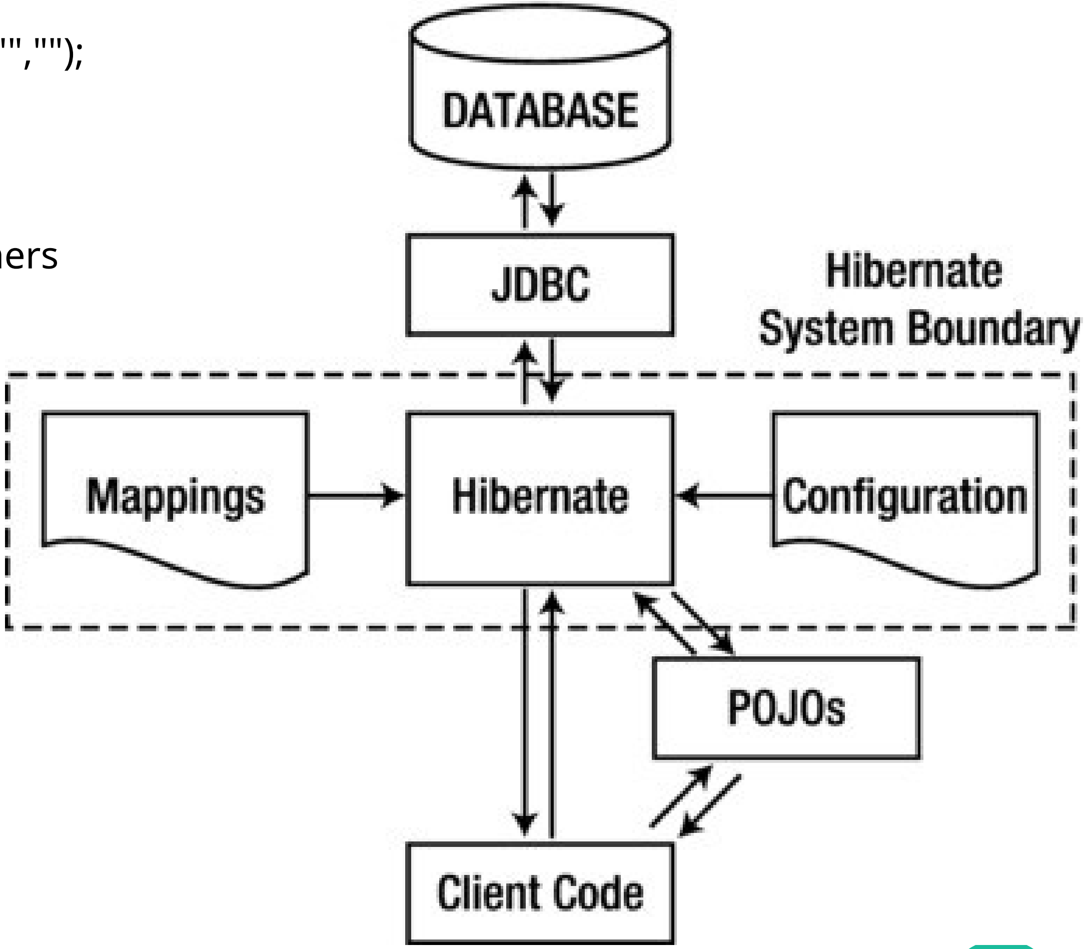
---

Hibernate is able to deal with object-relational impedance mismatch
problems by replacing direct, persistent database accesses with high-level object handling functions.

*It was started in 2001 by Gavin King as an alternative to EJB2 style entity bean.*

## Hibernate in a Java application

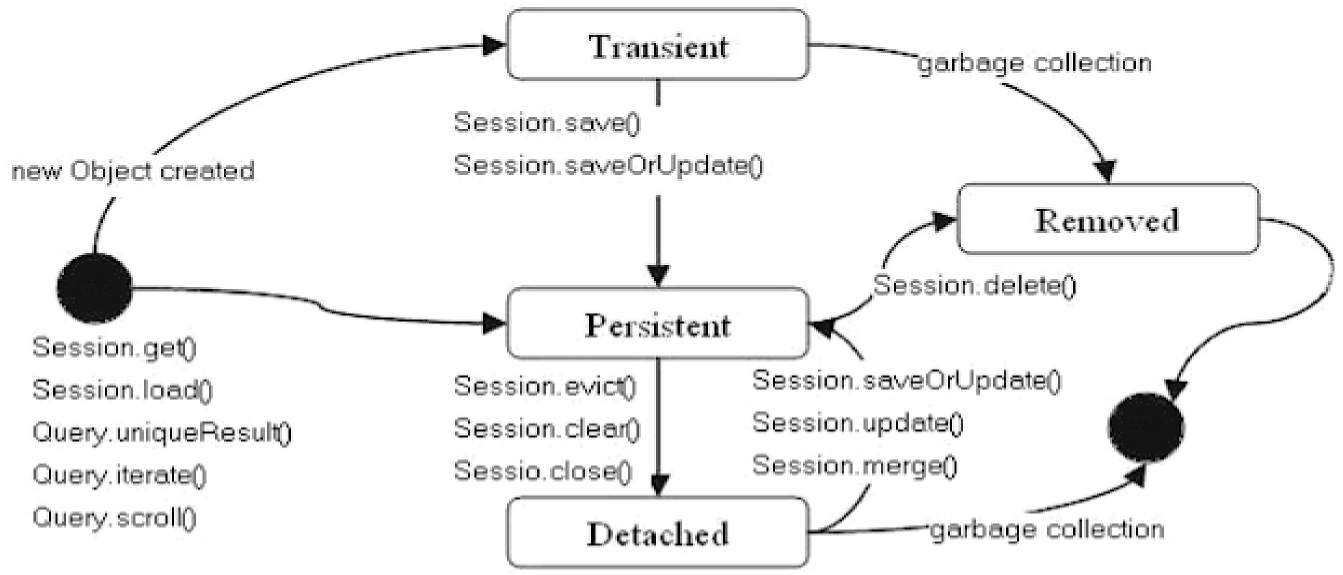
```
try {
   String url = "jdbc:msql://200.210.220.1:1114/Demo";
   Connection conn = DriverManager.getConnection(url,"","");
   Statement stmt = conn.createStatement();
   ResultSet rs;

   rs = stmt.executeQuery("SELECT Lname FROM Customers
                                          WHERE Snum = 2001");
   while ( rs.next() ) {
      String lastName = rs.getString("Lname");
      System.out.println(lastName);
   }
   conn.close();
} catch (Exception e) {
   System.err.println("Got an exception! ");
   System.err.println(e.getMessage());
}
```



Hibernate uses JDBC connections in order to interact with a database.

## The Persistence Life Cycle

### Object States



Transient  A New instance of  a persistent class which is not associated with a Session, has no representation in the database and no identifier value is considered transient by Hibernate:

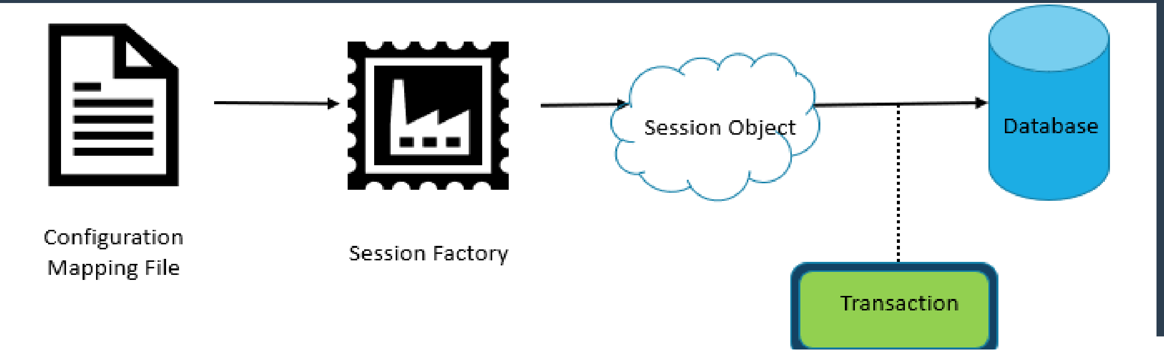
Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects.

Detached Hibernate Session, the persistent instance will become a detached instance.

use update() if you are sure that the Hibernate session does not contain an already persistent instance with the same id.
use merge() if you want to merge your modifications at any time without considering the state of the session.

load(): can return proxy without hitting the database unless required. (throw an exception)
get(): It always goes to the database. (Will return null)

*Create org.hibernate.cfg configuration object*
*Load Meta information (Mapping files)*
*Create org.hibernate.SessionFactory object*
*Make Hibernate API call on Session object*
*Close the Session*
*Close the SessionFactory object*



## Key components

***SessionFactory*** is immutable and shared by all Session. It also lives until the Hibernate is running.
Configures hibernate for the application using the provided configuration file and instantiates the session object.

***Session*** is used to get a physical network with a database. is a single-threaded, short-lived object. It provides the first-level cache.
Is created within the database layer in every DTO method.

***Transaction*** represents the unit of work with a DB. Is associated with session
(init by session.beginTransaction()

***Query*** uses SQL and HQL string to retrieve the data from the database and create objects.

***Criteria*** the primary use of criteria is to create and execute object-oriented queries and retrieve the objects.

***Configuration*** It represents the properties of files required by Hibernate

## Configuration

```
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-core</artifactId>
<version>5.3.6.Final</version> </dependency>
```

- hibernate.dialect  This property makes Hibernate generate the appropriate SQL for the chosen database.
- hibernate.connection.driver_class The JDBC driver class.
- hibernate.connection.url  The JDBC URL to the database instance.
- hibernate.connection.username The database username.
- hibernate.connection.password The database password.
- hibernate.connection.pool_size Limits the number of connections waiting in the Hibernate database connection pool.
- hibernate.connection.autocommit Allows autocommit mode to be used for the JDBC connection.
- hibernate.jdbc.batch_size  Controls the maximum number of statements Hibernate will batch together before asking the driver

- Configuration cfg=new Configuration();    //creating configuration
- cfg.configure("hibernate.cfg.xml");        //configure( .cfg.xml)
- SessionFactory factory=cfg.buildSessionFactory();  //creating seession factory object
- Session session=factory.openSession();      //creating session object
- Transaction t=session.beginTransaction();          //creating transaction object
- Employee e1=new Employee(111,"XXXX",40000);    session.persist(e1);  //persisting the object
- t.commit();                    //transaction is commited
- session.close();

# hibernate.cfg.xml

Specific Details

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

<hibernate-configuration>
 <session-factory>
 <!-- Database connection settings -->
 <property name="connection.driver_class">org.h2.Driver</property>
 <property name="connection.url">jdbc:h2:./db2</property>
 <property name="connection.username">sa</property>
 <property name="connection.password"/>
 <property name="dialect">org.hibernate.dialect.H2Dialect</property>
 <!-- Echo all executed SQL to stdout -->
 <property name="show_sql">true</property>
 <!-- Drop and re-create the database schema on startup -->
 <property name="hbm2ddl.auto">create-drop</property>
 <mapping class="chapter02.hibernate.Message"/>
<mapping resource="emp.hbm.xml"/>  // mapping file
```

Note: .hbm.xml should be used to save the file with the mapping document.

## Hibernate Query Language

HQL is Hibernate Query Language, it based on SQL and behind the scenes it is changed into SQL but the syntax is different. (Independent of the DB)

hql = "From EntityName"; Selecting a whole table

Avoid SQLQuery (1L caching)

hql = "Select id, name From Employee"; Select specific columns

hql = "From Employee where id = 22"; Include a Where clause

hql = "From Author a, Book b Where a.id = book.author"; Join

```
List result = session.createNativeQuery("SELECT * FROM some_table").list();

for (Object[] row : result) {

for (Object col : row) { System.out.print(col); }}
```

### Native SQL Queries

only when it can't be done using HQL

```
Object pollAnswered = getCurrentSession().createSQLQuery( "select * from ANSWERED where pol_id =
"+pollId+" and prf_log = '"+logid+"'").uniqueResult();
```

Example to get a unique result

### Criteria

```
List reviews = session.createCriteria(TravelReview.class)

.add(Restrictions.eq("author", "John Jones"))

.add(Restrictions.between("date",fromDate,toDate))

.add(Restrictions.ne("title","New York")).list();
```

Dinamic (for read)

Object-oriented Queries to retrieve the objects.

If you forget to JOIN FETCH an EAGER association in a JPQL or Criteria API query, you'll end up with an N+1 query issue.

Using Projections

# @Anotations

**@Entity** Specifies an object that maps to a database table. (to declare a class as an entity)

**@Table** Specifies which database table this object maps too. (default value->NamingStrategies)

**@JoinColumn** Specifies which column a foregin key is stored in.

**@JoinTable** Specifies an intermediate table that stores foreign keys.

**@Id** and **@GeneratedValue** (AUTO dont use , SEQUENCE , TABLE , IDENTITY)

**@ElementCollection** is used to specify a collection of a basic or **@Embeddable** types.

**@Column** (name,length,nullable,unique)

**@Cascade** = { CascadeType.ALL } (DELETE,DETACH,LOCK,MERGE,REPLICATE,SAVE_UPDATE)

**@Enumerated** for enums.

**@Access**(value=AccessType.PROPERTY) if you want Hibernate to use setters.

**@Inheritance**(strategy = InheritanceType.TABLE_PER_CLASS)

**@NamedQuery** is used to specify a JPQL query that can be retrieved later by its name.

**@Transient** is used to specify that a given entity attribute should not be persisted.

**@Fetch**(FetchMode.JOIN) mark as FetchType.LAZY, the FetchMode.JOIN will load the association eagerly.

Hibernate will always load the @ManyToOne children by default (FetchType.EAGER)

@ManyToOne and the @OneToOne associations are now EAGER by default.

## Hibernate Validation (Jakarta Bean Validation)

@Valid , @NotNull , @AssertTrue , @Past, @Size(min=6, max=20)

```
<dependency>
<groupId>org.hibernate.validator<groupId>
<artifactId>hibernate-validator</artifactId>
<version>6.1.2.Final</version>
</dependency>
```

```
@Table(name = “")
@Entity
public class Customer {
 @Id @GeneratedValue
private Integer id;

  @OneToMany(mappedBy="customer")
  @OrderBy("number")
  private List<Order> orders;}
```

```
 @Table(name = "”)
@Entity
   public class Order {
    @Id @GeneratedValue
    private Integer id;

       private String number;
    }
```

```
@Entity
@Table(name = "purchaseOrder")
@NamedQuery(name = Order.QUERY_BY_CUSTOMER,
query = "SELECT o FROM Order o WHERE o.customer = :"+Order.PARAM_CUSTOMER)
public class Order {

   public static final String QUERY_BY_CUSTOMER = "query.Order.byCustomer";
   public static final String PARAM_CUSTOMER = "customer";}
```

**use String constants for query, parameter and attribute names**

use the **Tuple** to process query results that return more than 1 object

## Logging

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

```
logging.file.name=hibernate_log_file_name.log
logging.pattern.console=
```

# *Type of Association Options*

## *Relationships (has)*

One-to-one Either end can be made the owner, but one (and only one) of them should be.

One-to-many The many end must be made the owner of the association.

Many-to-one One-to-many relationship viewed from the opposite the many end must be made the owner of the association.

Many-to-many Either end of the association can be made the owner.

Bi-Directional Many to Many using user managed join table object.

```
@Entity
@Table(name="FOO_BAR")
public class FooBar {

 @ManyToOne
 @JoinColumn(name = "fooId")
 private Foo foo;

 @ManyToOne
 @JoinColumn(name = "barId")
```

Bi-Directional Many to Many using Hibernate managed join table

```
@OneToMany
 @JoinTable(name="FOO_BAR",
 joinColumns = @JoinColumn(name="fooId"),
 inverseJoinColumns = @JoinColumn(name="barId"))
 private List<Bar> bars;
```

## Bi-directional One to Many Relationship

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

 private UUID fooid;

 @OneToMany(mappedBy = "bar")
 private List<Bar> bars;
}


@Entity
@Table(name="BAR")
public class Bar {

 private UUID barId;

 @ManyToOne
 @JoinColumn(name = "fooId")
 private Foo foo;
}
```

## Uni-Directional One to Many Relationship

```
@Table(name="BAR")
public class Bar {


@Table(name="FOO")
public class Foo {

 @OneToMany
 @JoinTable(name="FOO_BAR",
 joinColumns = @JoinColumn(name="fooId"),
 inverseJoinColumns = @JoinColumn(name="barId"))
 private List<Bar> bars;

@Table(name="FOO_BAR")
public class FooBar {

 @ManyToOne
 @JoinColumn(name = "fooId")
 private Foo foo;

 @ManyToOne
 @JoinColumn(name = "barId", unique = true)
 private Bar bar;
```

## Uni-directional One to One Relationship

```
@Entity

@Table(name="FOO")

public class Foo {


@OneToOne

private Bar bar; }


@Entity

@Table(name="BAR")

public class Bar {

private UUID barId;

}
```

## Bi-directional One to One

```
@Table(name = "countries")

public class Country {

@Id @GeneratedValue(strategy = GenerationType.IDENTITY)

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


@OneToOne(mappedBy = "country")

private Capital capital;


@Table(name = "capitals")

public class Capital {


 @OneToOne(cascade = CascadeType.ALL)

 @JoinColumn(name = "country_id")

 private Country country;
```

## Inheritance Mapping Strategies

Is **Table** a type ?)

Table per concrete class (InheritanceType.TABLE_PER_CLASS)

when there is a need to store each concrete class objects

of inheritance in separate tables. Should be avoided since it does not

render efficient SQL statements.

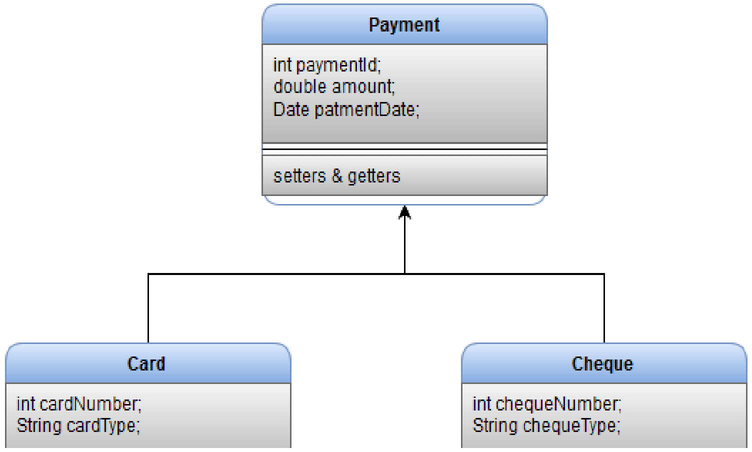


Table per hierarchy (InheritanceType.SINGLE_TABLE) cannot use NOT NULL

A single table is created for each class hierarchy.

To save the data of all classes hierarchy in to a single table of database.

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Plane")
public class Plane { ... }                    @DiscriminatorValue("A320")
```

Table per subclass (InheritanceType.JOINED)

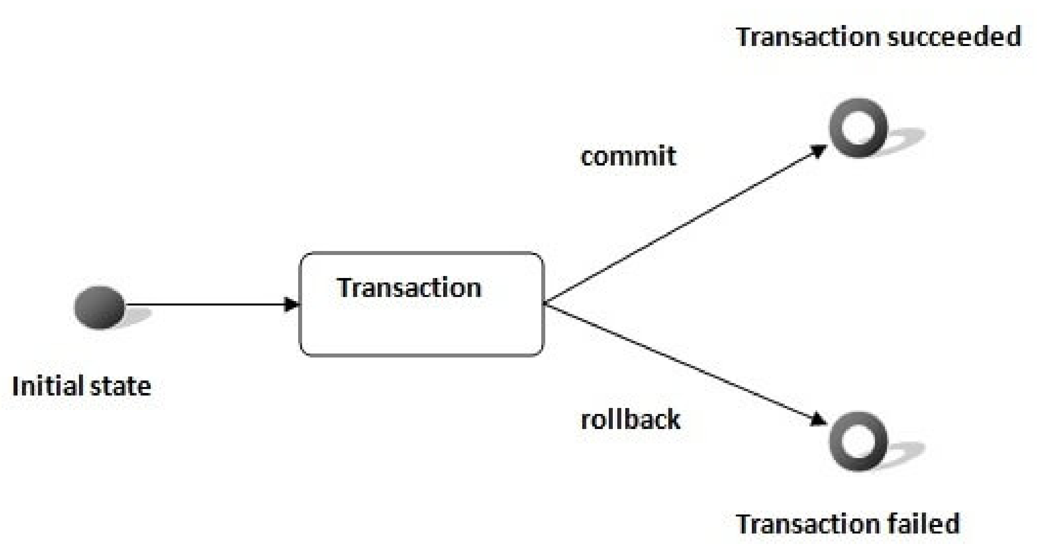
Hibernate creates separate tables for each class. To map a super class and its sub classes to its own tables of database

Each sub class table has a foreign key column, and we need to represent the foreign key in hibernate with <key> tag.

```
@Inheritance(strategy=InheritanceType.JOINED)    →    @PrimaryKeyJoinColumn(name="BOAT_ID")
```

# Hibernate Transaction Management

**A transaction simply represents a unit of work. In such case, if one step fails, the whole transaction fails. A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).**



In hibernate framework, we have Transaction interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

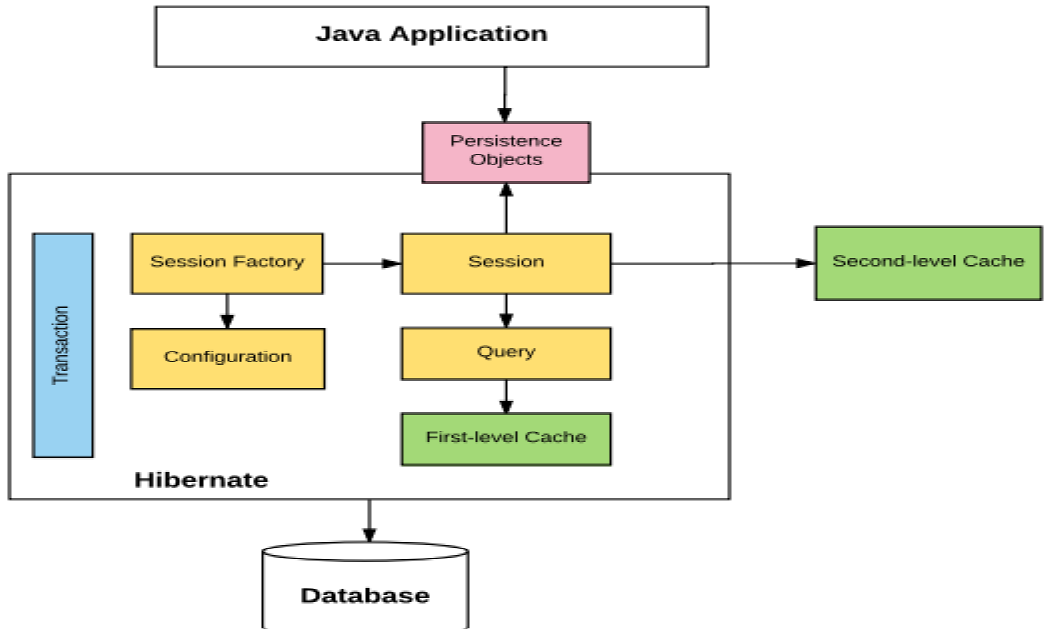
A transaction is associated with Session and instantiated by calling session.beginTransaction().

void begin() starts a new transaction.
void commit() ends the unit of work unless we are in FlushMode.NEVER.
void rollback() forces this transaction to rollback.
void setTimeout(int seconds) it sets a transaction timeout for any transaction
boolean isAlive() checks if the transaction is still alive.
void registerSynchronization(Synchronization s) registers a user synchronization callback for this transaction.
boolean wasCommited() checks if the transaction is commited successfully.
boolean wasRolledBack()

Note: you can leave it to the Spring declarative transaction management using @Transactional annotation.

## Hibernate Caching

improves the performance of the application by pooling the object in the cache. It is useful when we have to fetch the same data multiple times.



### First Level Cache

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. An application can use many session object.

### Second Level Cache

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 explicitely. (shared by all sessions)

*Is key in cache?* *Y →* ***N→***

*Is immutable nature key query?* *N→* *Y→*

*Have tables been updated ?* *Y→* ***N→***

*Execute query* *→*

*Put Result in cache →*

***→ return***

| | |
|---|---|
| hibernate.cache.provider_class | It represents the classname of a custom CacheProvider. |
| hibernate.cache.use_minimal_puts | optimizes the second-level cache. It minimizes writes, at the cost of more frequent reads. |
| hibernate.cache.use_query_cache | It is used to enable the query cache. |
| hibernate.cache.use_second_level_cache | is enabled by default for classes which specify a mapping. |
| hibernate.cache.query_cache_factory | It represents the classname of a custom QueryCache interface. |
| hibernate.cache.region_prefix | specifies the prefix which is used for second-level cache region names. |
| hibernate.cache.use_structured_entries | to store data in the second-level cache in a more human-friendly format. |

**Vendors (EH Cache, OS Cache, Swarm Cache, JBoss Cache)**

**Read-only: Nonstrict-read-write: Read-write: Transactional**

SessionFactory holds the second level cache data.

## Best Practices

- Always check the primary key field access, if it’s generated at the database layer then you should not have a setter for this.
- By default hibernate set the field values directly, without using setters. So if you want Hibernate to use setters, then make sure proper access is defined as @Access(value=AccessType.PROPERTY).
- If access type is property, make sure annotations are used with getter methods and not setter methods. Avoid mixing of using annotations on both filed and getter methods.
- Use native sql query only when it can’t be done using HQL, such as using the database-specific feature.
- If you have to sort the collection, use ordered list rather than sorting it using Collection API.
- Use named queries wisely, keep it at a single place for easy debugging. Use them for commonly used queries only. For entity-specific query, you can keep them in the entity bean itself.
- For web applications, always try to use JNDI DataSource rather than configuring to create a connection in hibernate.
- Do not treat exceptions as recoverable, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that the in-memory state accurately represents the persistent state.
- Avoid cascade remove for huge relationships.
- Prefer DAO pattern for exposing the different methods that can be used with entity bean
- Prefer lazy fetching for associations.
- Avoid Many-to-Many relationships, it can be easily implemented using bidirectional One-to-Many and Many-to-One relationships.
- For collections, try to use Lists, maps and sets. Avoid array because you don’t get benefit of lazy loading.
- Use @Immutable when possible.

**Thanks...**