Versioning or ObjectVersioning

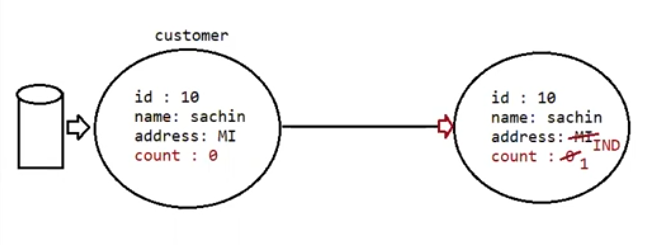
=========================

It keeps track of how many times object/record is loaded and modified using hibernate.

It generates a special column of type numeric, based on the special number the property of Entity class is used to keep track of the modification.

This special property/col initial value is 0 and it is incremented by 1 for every modification.

To configure this special property we need to use one annotation called "@Version".



Eg: HibernateVersioning

Eg: HibernateVersioningLoad

TimeStamping

===========

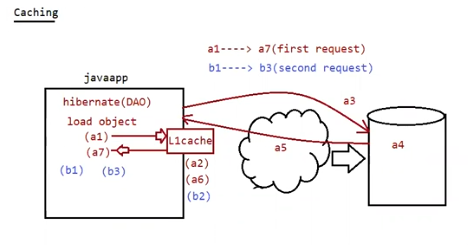
It allows us to keep track of Object is saved(record inserted) and object is lastly updated.

eg: keeping track of when the bank account opened and lastly modified

To do this we use annotations like @CreationTimeStamp, @UpdateTimeStamp

Eg: HibernateTimeStamping

Eg: HIbernateTimeStampingLastUpdated



Caching

=======

=>It is a temporoary memory that holds the data for temporary period of time.

=> Cache at client side will hold server data and uses it across the mulitple same requests to reduce the network trip

b/w client and server.

=> Hibernate supports 2 levels of Cache

a. First Level Cache(L-1 cache/session cache/default cache)

b. Second Level Cache(L2- cache/configurable cache)

eg: Stockmarket trading, live game score, weather report,

Note: session.save(obj), session.saveOrUpdate(obj), session.delete(obj) methods keep the object in L1cache unitll tx.commit() is called.

session.get() will get the object and keep it in L-1 cache and same object will be used across multiple session.get() method calls with same entity object id.

Eg: HibernateCaching

Eg: HibernateCachingGet

Caching

======

a. evict(Object obj) => it will remove particular object from L1-cache

b. clear() -> it will remove all object present in fromL1-cache

Eg: HibernateCachingEvictClear

c. in L1 cache duplicates are not allowed

Eg: HibernateCachingHashCode

2nd level cache

===============

This caching is associated with "SessionFactory", so we call it as "Global Cache".

Application will start to search for entity object in the following order

a. L1 cache of current session(if not there)

b. L2 cache of SessionFactory object(if not there)

c. Collect from db and keep in L2 cache and L1 cache then give it to application. I

It is a configurable cache and we can enbale or disable it.

hibernate supports L2 cache through "EHCache"

Eg: HibernateL2Cache

Eg: HibernateL2CacheVersion5

(don’t use the l2 cache on latest orm version 6 as the ehcache dependency is not upgraded to available orm versions)

Steps to configure ehcache in our hibernate project

1. Install 2 dependencies
2. Ehcache ( domain is net.sf.ehcache)

And the other one is

1. hibernate-ehcache integration dependency with hibernate orm

(make sure to install hibernate orm and ehcache integration with same version)

1. include this properties in the hibernate.cfg.xml file

this property is used to enable l2 cache in hibernate

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

Theis Hibernate configuration property used to enable second-level caching with **EhCache** as the caching provider:

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

**Here's what it means:**

hibernate.cache.region.factory\_class is the configuration key to define which cache region factory Hibernate should use.

org.hibernate.cache.ehcache.EhCacheRegionFactory tells Hibernate to use **EhCache** for its second-level cache.

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

This property tells **Hibernate + EHCache** where to find your **Ehcache configuration file** (ehcache.xml) on the classpath.

3. In the model class inform hiberante to use Caching startegy for Read purpose.

@Entity

@Cacheable

@Cache(usage = CacheConcurrencyStrategy.READ\_ONLY)//It specifies caching Strategy

public class InsurancePolicy implements Serializable{}

Working with LOB's

=========================

To work with LOB in hibernate we use

@Lob

private byte[] photo;

@Lob

private char[] resume;

Eg: HibernateLobOperation

Eg: HibernateLobRetrival

Customgenerator

Hibernate and JPA had supplied predefined genearator to create primary key value for almost all databases.

eg: identity,increment,auto,sequence, .....

if we want a primary key value to be generated for our columns as per our application needs then we need to go

customgenerators.

To create our own generator we need to implement an interface called "IdentifierGenerator"

It is a functional interface which contains only one method

public Serializable generate(SharedSessionContractImplementor session,Object object) throws HBE

Connection pooling

=> SessionFactory object holds jdb connection pool having set of ready made jdbc connection objects and uses them in the creation of HB session objects.

=> By default hibernate uses built in jdbc connection pool which is not suitable for production environment because of performance issue.

To control hibernate build in jdbc connection pool we write the following property in hibernate.cfg.xml

<property name="hibernate.connection.pool\_size">25</property>

Eg: HibernateBuiltInConnectionPool

Which jdbcconnection pool is best with hibernate integration?

standalone mode -> Don't use hibernate built in jdbc connection pool use Third party supplied jdbcconnection pool like hikaricp(best in market), proxool,viboor,agroal,c3po .....

webapplication mode-> Don't use 3rd party supplied only, use underlying server provided I

connection pool from servers like weblogic,tomcat, wildfly, ....

Eg: HibernateHickaricpConnectionPooling

Configuration of hibernate.cfg.xml for hickaricp

======================================

<!-- Use HikariCP as Connection Pool -->

<**property** name=*"hibernate.connection.provider\_class"*>org.hibernate.hikaricp.internal.HikariCPConnectionProvider</**property**>

<!-- HikariCP Settings -->

<**property** name=*"hibernate.hikari.maximumPoolSize"*>10</**property**>

<**property** name=*"hibernate.hikari.minimumIdle"*>5</**property**>

<**property** name=*"hibernate.hikari.idleTimeout"*>30000</**property**>

<**property** name=*"hibernate.hikari.connectionTimeout"*>20000</**property**>

<**property** name=*"hibernate.hikari.maxLifetime"*>1800000</**property**>

And include some dependencies as mentioned in the above pom.xml file.

Note: DataSource(I) --------- > jars provider should implement and give the class.

In **JDBC** (Java Database Connectivity), a **DataSource** is basically a **factory** for connections to a physical data source, like a database.

Instead of manually creating a Connection using DriverManager, a DataSource provides a **better, more flexible, and managed way** to get database connections.

**Simple way to think of it:**

* DriverManager = old-school, manual connection management.
* DataSource = modern, flexible, often managed by app servers, and supports features like connection pooling, distributed transactions, etc.

**Technically speaking:**

* DataSource is an **interface** (javax.sql.DataSource) in JDBC API.
* It provides a method like getConnection() to get a database connection.
* It can be configured with properties like **URL**, **username**, **password**, and sometimes advanced settings like **connection pooling**.

**Example: Without DataSource (using DriverManager):**

java

CopyEdit

Connection conn = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/mydb", "user", "password");

**Example: With DataSource:**

java

CopyEdit

DataSource ds = new MysqlDataSource();

ds.setURL("jdbc:mysql://localhost:3306/mydb");

ds.setUser("user");

ds.setPassword("password");

Connection conn = ds.getConnection();

**Why use DataSource?**

* **Easier to manage** (configured externally sometimes, no hardcoded passwords in code)
* **Supports connection pooling** (better performance)
* **Works better with application servers** (like Tomcat, WildFly, etc.)
* **More scalable** for real-world apps

 **DataSource** is a standard interface from **javax.sql.DataSource** (part of JDBC API).

 Hibernate uses its own **ConnectionProvider** (org.hibernate.engine.jdbc.connections.spi.ConnectionProvider).

 **Why not** just use DataSource everywhere inside Hibernate?

Hibernate **cannot** use DataSource directly **because DataSource is limited** — it only represents **one way** of getting a connection (through a managed source, like a connection pool or an app server).

Hibernate needs **more flexibility** than what DataSource alone provides.

| **Hibernate Needs** | **Can DataSource Alone Handle It?** | **Why ConnectionProvider is Needed** |
| --- | --- | --- |
| Support **DriverManager** based connections (no datasource) | ❌ No | DataSource can't create direct connections; ConnectionProvider can |
| Support **user-managed JDBC connections** | ❌ No | Sometimes users want to give Hibernate an already-opened Connection |
| Support **programmatic custom connection creation** | ❌ No | DataSource is just a standard API; ConnectionProvider can be customized easily |
| Integrate with **JNDI** lookup of DataSource | ✅ Yes | ConnectionProvider can wrap a JNDI lookup DataSource |
| Support **multiple types of connection pools** | ❌ No | ConnectionProvider can manage HikariCP, C3P0, DBCP differently |
| Manage **connection lifecycle (open/close settings, aggressive release mode)** | ❌ No | ConnectionProvider controls how and when connections are released |

**Key Difference**:

* DataSource = Just **"here’s a way to get a connection"** (simple).
* ConnectionProvider = **"I will handle how to create, manage, and close connections in many different ways depending on your environment."**
* **DataSource** is like a **tap** that gives you water.
* **ConnectionProvider** is like a **plumbing manager** — it decides whether the tap gives you city water, well water, filtered water, etc.

Hibernate needs the **plumbing manager** — not just a tap — because **different apps need different water sources**.

Thus, Hibernate defines its own **ConnectionProvider interface**, which **can** internally use a DataSource, **but is much more powerful and flexible**.

Why are we not configuring the Datasource class directly in hibernate? why are configuring connection provider class name?

Answer. hibernate f/w is designed to pickup the datasource class based on the connection provider that we have configured. by configuring in this style, we can restrict datasource and jdbc connection pool associated with hibernate. hibernate will give support only for few thirdparty vendors like

* 1. hikaricp(best) b. c3po c.proxool d. viboor e. agroal

BulkOperation

=> To select or manipulate one or more record/object having our choice criterial value we need to go for "Bulk operation".

a. HQL.

b. Native SQL.

c. Criterian API.

HQL

=====

1. HQL stands for Hibernate Query Language.

2. It uses Objects based Query Language(these queries will be written based on the entity class names and properties name)

3. Hibernate dialect internally converts HQL queries to DB specific SQL Queries.

4. HQL queries are DBIndependent and they supports portability.

5. HQL supports both select and non-select operation

6. HQL can also be used to perform SingleRowOperation(SRO) and also for bulk operation having our choice conditions/criteria.

7. HQL supports positional params(?) (supported only in older versions) and also it supports named params(:name)

8. HQL keywords are not case sensitive, but entity class names and properties names are case sensitive.

9. HQL supports relational operators, conditional statements, joins,aggregate functions, projections, ....

eg:

SQL> SELECT \* FROM EMP WHERE EMPNO> =? AND EMPNO <=?

HQL> FROM in.ineuron.entity.Employee WHERE eno> =? AND eno <=? (positional param)

HQL> FROM in.ineuron.entity.Employee WHERE eno =: firstNum AND enq =: secondNum(named param)

SQL> DELETE FROM EMP WHERE JOB =?

HQL> DELETE FROM in.ineuron.entity.Employee WHERE job =?

HQL> DELETE FROM in.ineuron.entity.Employee WHERE job =: desg

Note: if we are selecting all columns/properties in the HQL Select query then placing select keyword is optional.

HQL select Queries

a. Entity Queries(Getting all properties values of the record)

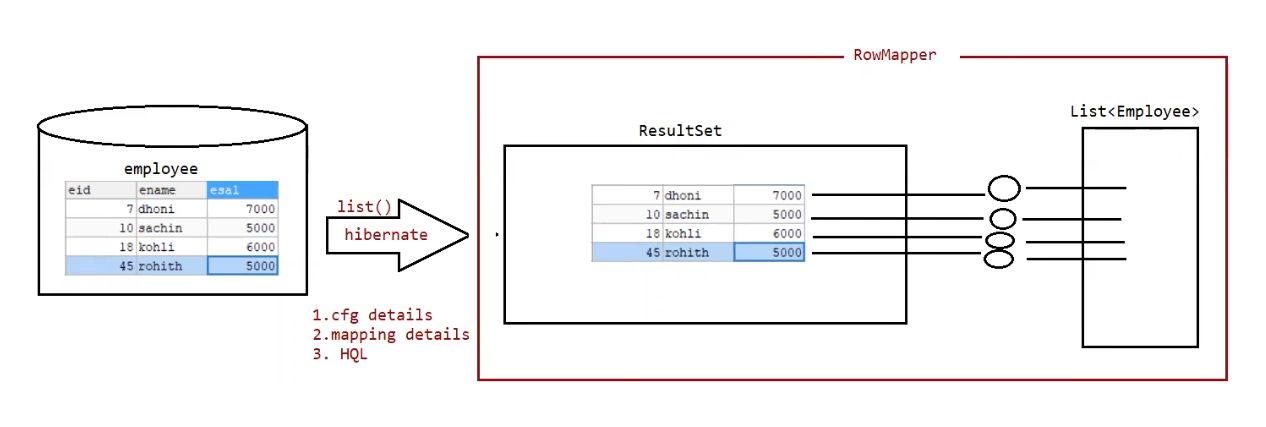
eg: FROM in.ineuron.entity.Employee(with or without condition)

b. Scalar Queries(Getting specific column or specific multiple column values)

eg: SELECT eno,ename,eaddr FROM in.ineuron.entity.Employee (with or without condition)

SELECT eno From in.ineuron.entity.Employee(with or without condition)

SELECT count(\*) From in.ineuron.entity.Employee



Eg: HibernateBulkOperation

Note:

In plain jdbc converting ResultSet object to DTO object is a manual process, where as in orm framework like hibernate, spring jdbc, spring orm and spring datajpa same happens internally using "rowmapper" concept.

Note:

If we use xml approach setter and getter methods are mandatory, but if we use Annotations for mapping setter and getter methods are not required, hibernate internally uses reflection api and it binds the value from ResultSet to private properties of the Model.

Eg: HibernateBulkOperation2WithAnnotation

Eg: HibernateBulkOperation3WithXml