**What is hibernate framework?**

Hibernate is java based ORM tool that provides framework for mapping application domain objects to the relational database tables and vice versa. Hibernate provides reference implementation of Java Persistence API, Hibernate framework provide option to map plain old java objects to traditional database tables with the use of JPA annotations as well as XML based configuration.

**What are the important benefit of Hibernate ?**

* Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business logic.
* Hibernate framework provides support for **XML** as well as JPA **annotations**, that makes our code implementation independent.
* Hibernate provides a powerful query language (**HQL**) that is similar to SQL.
* Hibernate provides **Caching**
* Hibernate provides **Lazy Loading**
* Hibernate is **Open source**

**What are the advantage of Hibernate over JDBC**

Hibernate is used to speed up development and enhance maintainability, not to get a perfromance edge over JDBC.

**In Jdbc** : The following steps we do.

* Load the driver
* Register the driver with driver manager
* Establish the connection
* Create Statement Object.
* Send the sql query
* Get the result
* Close the connection

**In Hibernate :**

* Hibernate removes a lot of boiler-plate code that comes with JDBC API, the code looks more cleaner and readable.
* Hibernate supports inheritance, associations and collections. These features are not present with JDBC API.
* Hibernate implicitly provides transaction management, in fact most of the queries can’t be executed outside transaction. In JDBC API, we need to write code for transaction management using commit and rollback.
* JDBC API throws SQLException that is a checked exception, so we need to write a lot of try-catch block code. Most of the times it’s redundant in every JDBC call and used for transaction management. Hibernate wraps JDBC exceptions and throw JDBCException or HibernateException un-checked exception, so we don’t need to write code to handle it.

**What are the different levels of ORM quality?**

**Pure relational :** The entire application, including the user interface, is designed around the relational model and SQL-based relational operations.

**Light object mapping :** The entities are represented as classes that are mapped manually to the relational tables. The code is hidden from the business logic using specific design patterns.

**Medium object mapping :** The application is designed around an object model. The SQL code is generated at build time. And the associations between objects are supported by the persistence mechanism, and queries are specified using an object-oriented expression language.

**Full object mapping :** Full object mapping supports sophisticated object modeling: composition, inheritance, polymorphism and persistence. The persistence layer implements transparent persistence; persistent classes do not inherit any special base class or have to implement a special interface. Efficient fetching strategies and caching strategies are implemented transparently to the application.

**What are the key component/object of hibernate ?**

**Configuration** : Represents a configuration or properties file required by the hibernate.

**SessionFactory** − Configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated.

**Session** − Used to get a physical connection with a database.

**Transaction** − Represents a unit of work with the database and most of the RDBMS supports transaction functionality.

**Query** − Uses SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects.

**Criteria** − Used to create and execute object oriented criteria queries to retrieve objects.

**Hibernate configuration** (hibernate.cfg.xml)

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

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name=*"hibernate.connection.****driver\_class****"*>org.postgresql.Driver</property>

<property name=*"hibernate.connection.****url****"*>jdbc:postgresql://localhost:5432/hibernatedb</property>

<property name=*"hibernate.connection.****username****"*>postgres</property>

<property name=*"hibernate.connection.****password****"*>postgres</property>

<property name=*"hibernate.connection.****pool\_size****"*>1</property>

<property name=*"hibernate.chache.****provider\_class****"*>org.hibernate.cache.NoCacheProvider</property>

<property name=*"****hibernate.show\_sql****"*>true</property>

<property name=*"****hibernate.dialect****"*>org.hibernate.dialect.PostgreSQLDialect</property>

<property name=*"****hbm2ddl.auto****"*>create</property>

<mapping class=*"dto.UserDetails"*></mapping>

</session-factory>

</hibernate-configuration>

**How to Persist a object?**

**public** **static** **void** main(String[] args) {

UserDetails user = **new** UserDetails();

user.setUserId(1);

user.setUsername("amit");

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

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(user);

session.getTransaction().commit();

}

**What are the options to generate primary key?**

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

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.SEQUENCE

**GenerationType.IDENTITY :** The GenerationType.IDENTITY is the easiest to use but not the best one from a performance point of view. It relies on an auto-incremented database column and lets the database generate a new value with each insert operation.

**GenerationType.SEQUENCE :** The GenerationType.SEQUENCE is my preferred way to generate primary key values and uses a database sequence to generate unique values.

**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.

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "id", updatable = false, nullable = false)

private Long id;

**Object states in hibernate?**

**Transient :** An object which is not associated with hibernate session and does not represent a row in the database is considered as transient.

**Persistent :** An object that is associated with the hibernate session is called as Persistent object. When the object is in persistent state, then it represent one row of the database and consists of an identifier value.

**Detached :** **Object which is just removed from hibernate session** is called as detached object. Any changes made to the detached objects are not saved to the database. The detached object can be reattached to the new session and save to the database using update, saveOrUpdate and merge methods.

public static void main(String[] args) {

**// Transient object state**

        Student student = new Student();

        student.setId(101);

        student.setName("Mukesh");

        student.setRoll("10");

        student.setDegree("B.E");

        student.setPhone("9999");

**// Transient object state**

        Session session = new **AnnotationConfiguration**().configure()

                .buildSessionFactory().openSession();

        Transaction t = session.beginTransaction();

**// Persistent object state**

        session.save(student);

        t.commit();

**// Persistent object state**

        session.close();

**// Detached object state**

    }

**Different between session.get() and session.load()**

**get()** : It always **hit the database** and return the real object, an object that represent the database row, not proxy.If no row found , it **return null**.

For example, in a Stock application , Stock and StockTransactions should have a “one-to-many” relationship, when you want to save a stock transaction, it’s common to declared something like below

Stock stock = (Stock)session.get(Stock.class, new Integer(2));

StockTransaction stockTransactions = new StockTransaction();

//set stockTransactions detail

stockTransactions.setStock(stock);

session.save(stockTransactions);

Hibernate:

select ... from mkyong.stock stock0\_

where stock0\_.STOCK\_ID=?

Hibernate:

insert into mkyong.stock\_transaction (...)

values (?, ?, ?, ?, ?, ?)

In session.get(), Hibernate will hit the database to retrieve the Stock object and put it as a reference to StockTransaction. However, this save process is extremely high demand, there may be thousand or million transactions per hour, do you think is this necessary to hit the database to retrieve the Stock object everything save a stock transaction record? After all you just need the Stock’s Id as a reference to StockTransaction.

**load():** It will always return a “proxy” (Hibernate term) without hitting the database. In Hibernate, proxy is an object with the given identifier value, its properties are not initialized yet, it just look like a temporary fake object.

Stock stock = (Stock)session.load(Stock.class, new Integer(2));

StockTransaction stockTransactions = new StockTransaction();

//set stockTransactions detail

stockTransactions.setStock(stock);

session.save(stockTransactions);

Hibernate:

insert into mkyong.stock\_transaction (...)

values (?, ?, ?, ?, ?, ?)

In session.load(), Hibernate will not hit the database (no select statement in output) to retrieve the Stock object, it will return a Stock proxy object – a fake object with given identify value. In this scenario, a proxy object is enough for to save a stock transaction record.

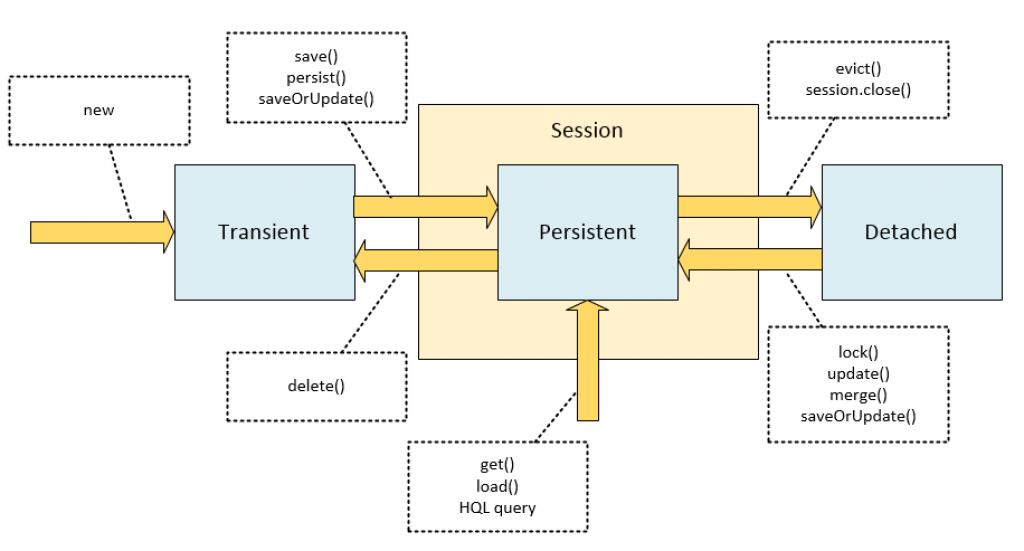
It will always return a proxy object with the given identity value, even the identity value does not exists in database. However, when you try to initialize a proxy by retrieve it’s properties from database, it will hit the database with select statement. If no row is found, a **ObjectNotFoundException**will throw.

Stock stock = (Stock)session.load(Stock.class, new Integer(100)); //proxy

//initialize proxy, no row for id 100, throw ObjectNotFoundException

System.out.println(stock.getStockCode());

**Hibernate Session: save, persist, update, merge, saveOrUpdate**



**persist() :** The persist method is intended for adding a new entity instance to the persistence context, i.e. transitioning an instance from transient to persistent state. We usually call it when we want to add a record to the database (persist an entity instance)

Return type : void

Person person = new Person();

person.setName("John");

session.persist(person);

session.evict(person);

 //**Calling persist on detached object throws PersistenceException**

session.persist(person); // PersistenceException!

**save() :** The save method is an “original” Hibernate method that does not conform to the JPA specification.

Its purpose is basically the same as persist, but it has different implementation details.

Return type : Serializable value of the identifier.

Person person = new Person();

person.setName("John");

Long id1 = (Long) session.save(person);

session.evict(person);

Long id2 = (Long) session.save(person);

The id2 variable will differ from id1. The call of save on a detached instance creates a new persistent instance and assigns it a new identifier, which results in a duplicate record in a database upon committing or flushing.

**update() :** It acts upon passed object (its return type is void); the update method transitions the passed object from detached to persistent state;

Person person = new Person();

person.setName("John");

session.save(person);

session.evict(person);

person.setName("Mary");

session.update(person);

this method throws an exception if you pass it a transient entity.

Person person = new Person();

person.setName("John");

session.update(person); // PersistenceException!

**saveOrUpdate()** : The main difference of saveOrUpdate method is that it does not throw exception when applied to a transient instance; instead, it makes this transient instance persistent. The following code will persist a newly created instance of Person:

Person person = new Person();

person.setName("John");

session.saveOrUpdate(person);

**merge() :** The main intention of the merge method is to update a persistent entity instance with new field values from a detached entity instance.

finds an entity instance by id taken from the passed object (either an existing entity instance from the persistence context is retrieved, or a new instance loaded from the database);

copies fields from the passed object to this instance;

returns newly updated instance.

Person person = new Person();

person.setName("John");

session.save(person);

session.evict(person);

person.setName("Mary");

Person mergedPerson = (Person) session.merge(person);

**Difference between session.close(), session.evict(Object o) and session.clear() ?**

**clear () :**  When this method get called inside transaction boundary then all objects which are currently associate with particular session will be  disconnected / clean or no longer associate with that Session instance.

Therefore, after calling this method nothing will be performed on persistance layer or DB.

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

**close() :**Close session by calling **session.close()** method, means End the session and releasing the JDBC Connection and clean up.

**Difference between session.flush() and session.getTransaction().commit()**

**flush():** Forces the session to flush. It is used to synchronize session data with database.When you call session.flush(), the statements are executed in database but it will not committed. If you dont call session.flush() and **if you call session.commit() , internally commit() method executes the statement and commits.**

So **commit()= flush+commit.**

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.

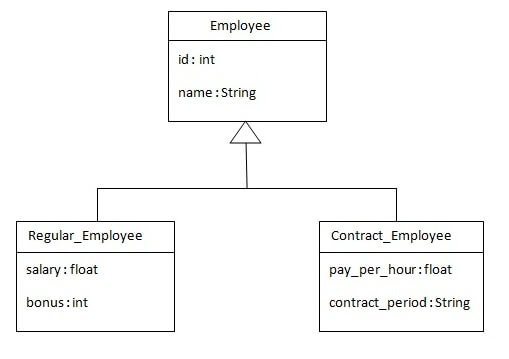
**Hibernate Inheritance Mapping**

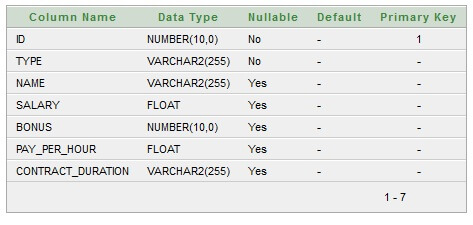
Single Table

Table Per Class

Joined

**Single Table Stratergy :**In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.





@Entity

@Inheritance(strategy=**InheritanceType.SINGLE\_TABLE**)

**@DiscriminatorColumn**(name="type",**discriminatorType=DiscriminatorType.STRING**)

**@DiscriminatorValu**e(value="employee")

public class **Employee**

@Entity

**@DiscriminatorValue("**regularemployee**")**

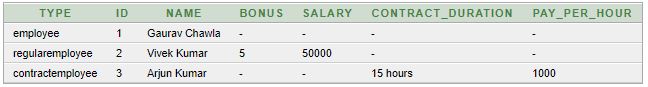
public class **Regular\_Employee extends Employee**

@Entity

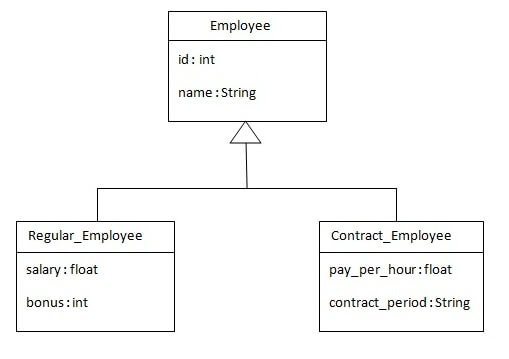
**@DiscriminatorValue("**contractemployee**")**

public class **Contract\_Employee extends Employee**

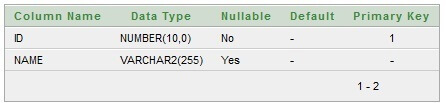
Saving each of the above object results in below table values



**Table Per Class Stratergy :** In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

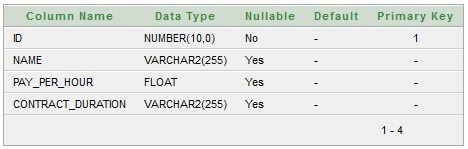


***Employee table***



**Regular\_Employee Table**

***Contract\_Employee Table***



@Entity

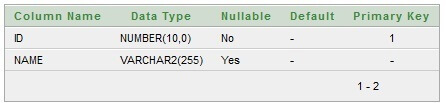
@Table(name = "employee102")

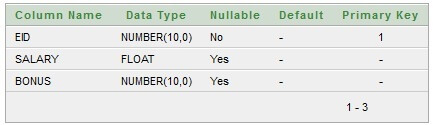
@Inheritance(strategy = **InheritanceType.TABLE\_PER\_CLASS**)

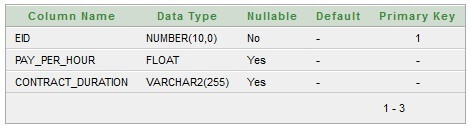
public class **Employee**

**Joined Stratergy :** In case of table per subclass strategy, tables are created as per persistent classes but they are treated using primary and foreign key. So there will not be any duplicate column in the relation.

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







**How can we map the classes as immutable?**  
If we don’t want an application to update or delete objects of a class in hibernate, we can make the class as immutable by setting mutable=false or @Immutable annotation.

If mutable = “false” or @Immutable is declared in class element, it means the **updates to this class will be ignored, but no exception is thrown, only the add and delete operation are allow**.

@Entity

@Immutable

@Table(name = "stock", catalog = "mkyong")

public class Stock implements java.io.Serializable {

If mutable = “false” or @Immutable is declared in collection, it means the **add and delete-orphan are not allow in this collection, with exception throw, only update and ‘cascade delete all’ are allow**.

1. **CascadeType.PERSIST** : cascade type presist means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : cascade type merge means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH** : cascade type refresh does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : cascade type remove removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : cascade type detach detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : cascade type all is shorthand for all of the above cascade operations.

**Hibernate Collection Mapping :**

**https://www.youtube.com/watch?v=Of7qR-WagYk&index=15&list=PL4AFF701184976B25**

**Hibernate Query Language (HQL) :**

Hibernate Query Language (HQL) is same as SQL (Structured Query Language) but it doesn't depends on the table of the database. Instead of table name, we use class name in HQL. So it is database independent query language.

It is an object oriented representation of Hibernate Query. The object of Query can be obtained by calling the createQuery() method Session interface.

The query interface provides many methods. There is given commonly used methods:

1. **public int executeUpdate()** is used to execute the update or delete query.
2. **public List list()** returns the result of the ralation as a list.
3. **public Query setFirstResult(int rowno)** specifies the row number from where record will be retrieved.
4. **public Query setMaxResult(int rowno)** specifies the no. of records to be retrieved from the relation (table).
5. **public Query setParameter(int position, Object value)** it sets the value to the JDBC style query parameter.
6. **public Query setParameter(String name, Object value)** it sets the value to a named query parameter.

**get all the records**

Query query=session.createQuery("from Emp");//here persistent class name is Emp

List list=query.list();

**get records with pagination**

Query query=session.createQuery("from Emp");

query.setFirstResult(5);

query.setMaxResult(10);

List list=query.list();//will return the records from 5 to 10th number

**update query**

Transaction tx=session.beginTransaction();

Query q=session.createQuery("update User set name=:n where id=:i");

q.setParameter("n","Udit Kumar");

q.setParameter("i",111);

int status=q.executeUpdate();

System.out.println(status);

tx.commit();

**delete query**

Query query=session.createQuery("delete from Emp where id=100");

//specifying class name (Emp) not tablename

query.executeUpdate();

**Aggregate functions**

Query q=session.createQuery("select sum(salary) from Emp");

List<Integer> list=q.list();

System.out.println(list.get(0));

**HCQL (Hibernate Criteria Query Language)**

The Hibernate Criteria Query Language (HCQL) is used to fetch the records based on the specific criteria. The Criteria interface provides methods to apply criteria such as retreiving all the records of table whose salary is greater than 50000 etc.

**public** Criteria createCriteria(Class c)

The commonly used methods of Criteria interface are as follows:

1. **public Criteria add(Criterion c)** is used to add restrictions.
2. **public Criteria addOrder(Order o)** specifies ordering.
3. **public Criteria setFirstResult(int firstResult)** specifies the first number of record to be retreived.
4. **public Criteria setMaxResult(int totalResult)** specifies the total number of records to be retreived.
5. **public List list()** returns list containing object.
6. **public Criteria setProjection(Projection projection)** specifies the projection.

**Restrictions class**

1. public static SimpleExpression **lt**(String propertyName,Object value) sets the **less than** constraint to the given property.
2. public static SimpleExpression **le**(String propertyName,Object value) sets the **less than or equal** constraint to the given property.
3. public static SimpleExpression **gt**(String propertyName,Object value) sets the greater than constraint to the given property.
4. public static SimpleExpression **ge**(String propertyName,Object value) sets the greater than or equal than constraint to the given property.
5. public static SimpleExpression **ne**(String propertyName,Object value) sets the **not equal**constraint to the given property.
6. public static SimpleExpression **eq**(String propertyName,Object value) sets the equal constraint to the given property.
7. public static Criterion **between**(String propertyName, Object low, Object high) sets the **between** constraint.
8. public static SimpleExpression **like**(String propertyName, Object value) sets **the like constraint** to the given property.

**Order Class**

1. public static Order **asc**(String propertyName) applies the **ascending order** on the basis of given property.
2. public static Order **desc**(String propertyName) applies the **descending order** on the basis of given property.

**Example of HCQL to get all the records**

Crietria c=session.createCriteria(Emp.class);//passing Class class argument

List list=c.list();

**Example of HCQL to get the 10th to 20th record**

Crietria c=session.createCriteria(Emp.class);

c.setFirstResult(10);

c.setMaxResult(20);

List list=c.list();

**Example of HCQL to get the records whose salary is greater than 10000**

Crietria c=session.createCriteria(Emp.class);

c.add(Restrictions.gt("salary",10000));//salary is the propertyname

List list=c.list();

**Example of HCQL to get the records in ascending order on the basis of salary**

Crietria c=session.createCriteria(Emp.class);

c.addOrder(Order.asc("salary"));

List list=c.list();

**Native SQL**

**public** **SQLQuery** **createSQLQuery(String sqlString) throws HibernateException**

To list all employee

String sql = "SELECT \* FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

List results = query.list();

**What are Callback interfaces?**

These interfaces are used in the application to receive a notification when some object events occur. Like when an object is loaded, saved or deleted. There is no need to implement callbacks in hibernate applications, but they’re useful for implementing certain kinds of generic functionality.

**Hibernate Second Level Cache**

uses a common cache for all the session object of a session factory. It is useful if you have multiple session objects from a session factory.

**SessionFactory holds the second level cache data.** It is global for all the session objects and not enabled by default.

Different vendors have provided the implementation of Second Level Cache.

**EH Cache**

**OS Cache**

**Swarm Cache**

**JBoss Cache**

Cache Usage :

1. **read-only:** caching will work for read only operation.
2. **nonstrict-read-write:** caching will work for read and write but one at a time.
3. **read-write:** caching will work for read and write, can be used simultaneously.
4. **transactional:** caching will work for transaction.

Steps to implement second level cache

Add 2 configuration setting in hibernate.cfg.xml file

**<property** name="cache.provider\_class"**>**org.hibernate.cache.EhCacheProvider**</property>**

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

Add following annotation on entity class

@Cacheable

@org.hibernate.annotations.Cache(usage = **CacheConcurrencyStrategy**.**READ\_WRITE**)

Or we can configure in **ehcache.xml**

<?xml version="1.0"?>

<ehcache>

<defaultCache

maxElementsInMemory="100"

eternal="false"

timeToIdleSeconds="120"

timeToLiveSeconds="200" />

<cache name="com.javatpoint.Employee"

maxElementsInMemory="100"

eternal="false"

timeToIdleSeconds="5"

timeToLiveSeconds="200" />

</ehcache>

https://career.guru99.com/hibernate-interview-questions/