**Advance Java**

**Frameworks- (Collection of APIS)**

* **Hibernate-**
* It is invented by **Gavin King** in 2001
* Hibernate is a popular **Object-Relational Mapping (ORM)** framework in Java. It simplifies database interactions by mapping Java objects (entities) to database tables and vice versa.
* With Hibernate, developers can work with data in terms of Java objects rather than writing SQL queries directly, which improves productivity and reduces boilerplate code.
* It is an open source, lightweight, ORM (Object Relational Mapping) tool.
* It is based on jdbc.(only difference is we have to add querys manually in jdbc but in hibernate it will take care of it)
* With the help of mapping we can get on which table are class is map & in that table which column are class values are map.
* So hibernate will directly insert & delete this operations.

**Advantages of Hibernate:**

* **Reduces Boilerplate Code:** Eliminates the need for most of the JDBC code.
* **Portability:** Applications are less tied to a specific database.
* **Scalability:** Designed to handle large-scale applications.
* **Error Reduction:** Minimizes SQL injection risks and reduces errors related to raw SQL.
* **Automatic Table Creation**
* **Simplifies Complex Join:** Fetching data from multiple tables is easy in hibernate framework.
* **APIS –**

API full form is an Application Programming Interface that is a collection of communication protocols & it allows developers to interact with and utilize pre-defined functionalities without having to write the code from scratch. In simpler terms, an API defines a contract or a way for different components, systems, or programs to communicate with each other.

**Hibernate Properties-**

1. **DIALECT-**

* To convert java datatypes into sql datatypes.
* Converts HQL query to used datbase query.

1. **HBM2DDL\_Auto-**

- performs crud operations .

- use to update & create table in database.

1. **SHOW\_SQL-**

* It displays performed sql querys on console.

1. **FORMAT\_SQL-**

* It displays sql querys in formatted form on console.

A screenshot of a computer

AI-generated content may be incorrect.

**Installation -**

Download hibernate version as per your java compatability🡪 extract zip file🡪

Create new project in java🡪right click on created project (to add libraries)🡪

go to build path🡪 configure build path🡪 libraries🡪 click on class path & add external jars🡪

go to hibernate extracted file🡪 lib🡪required🡪select all files🡪 apply🡪apply & save

(add sql connector also if its not present there)

**Steps-**

**3 ways**

* We can use both hibernate.cfg.xml file & hibernate.hbm.xml file.
* We can only use hibernate.cfg.xml file.
* We can do it without both files by java based using annotation.

**File name- hibernate.cfg.xml – (Jdk 1.0-2.0)**

<!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>

<!-- DB config -->

<property name=*"hibernate.connection.driverClassName"*>com.mysql.jdbc.Driver</property>

<property name=*"hibernate.connection.url"*>jdbc:mysql://localhost:3306/hbm</property>

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

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

<!-- Hibernate config -->

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

<property name=*"hibernate.hbm2ddl.auto"*>update</property>

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

<!-- create - it creates everytime new table update - it checks first if

there is existing table then it will add on it -->

<!-- show\_sql - it shows which querys has used -->

<mapping resource=*"hibernate.hbm.xml"* />

</session-factory>

</hibernate-configuration>

**File name- hibernate.hbm.xml- (Jdk 1.0-2.0)**

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name=*"com.expo.Student"*>

<id name=*"sid"* type=*"int"*></id>

<property name=*"sName"*></property>

<property name=*"sAddress"*></property>

</class>

</hibernate-mapping>

**Java based using annotation- (Jdk 3.0)**

**private** **static** StandardServiceRegistry *registry* = **null**;

**private** **static** SessionFactory *sf* = **null**;

**public** **static** SessionFactory getSessionFactory() {

**if** (*sf* == **null**) {

Map<String, Object> map = **new** HashMap<String, Object>();

map.put(Environment.***DRIVER***, "com.mysql.jdbc.Driver");

map.put(Environment.***URL***, "jdbc:mysql://localhost:3306/hbmjavabased");

map.put(Environment.***USER***, "root");

map.put(Environment.***PASS***, "root");

map.put(Environment.***DIALECT***, "org.hibernate.dialect.MySQLDialect");

map.put(Environment.***HBM2DDL\_AUTO***, "update");

map.put(Environment.***SHOW\_SQL***, **true**);

*registry* = **new** StandardServiceRegistryBuilder().applySettings(map).build();

MetadataSources mds = **new** MetadataSources(*registry*).addAnnotatedClass(Driver.**class**); **🡪(POJO Class Name)**

Metadata md = mds.getMetadataBuilder().build();

*sf* = md.buildSessionFactory();

}

**return** *sf*;

}

**What is MetadataSources in Hibernate?**

In Hibernate, MetadataSources is a class that helps **build metadata** required for the Hibernate **SessionFactory**. It collects mapping information (such as entity classes, XML mappings, and annotated classes) before constructing the Hibernate **metadata**.

**What is Metadata in Hibernate?**

In Hibernate, Metadata refers to the information about the entity mappings, database configurations, and schema details that Hibernate uses to interact with the database. It helps Hibernate understand how Java objects relate to database tables.

* **Session methods-**

1. **session.get() –** Eager Loading

 Executes SQL query immediately and fetches data from the database.

 If the object is not found, it returns null.

 Best when you are unsure if the object exists.

1. **Session.load()-**

 Does not hit the database immediately. Instead, it returns a proxy object.

 The actual query is executed only when a method is called on the object.

 If the object does not exist, it throws an ObjectNotFoundException.

 Best when you are sure the object exists and want to improve performance.

* **Annotations of Hibernate-**

Annotation provide metadata configuration for mapping Java classes to database tables. Hibernate supports **Java Persistence API (JPA) annotations** and **Hibernate-specific annotations**.

1. **Entity and Table Mapping-**

* **@Entity**

Marks a class as a persistent entity.

* **@Table(name = "employees")**

Specifies the table name in the database.

java

1. **Primary Key Annotations-**

**- @Id**

Marks a field as the primary key.

**- @GeneratedValue**

Generates primary key values automatically.

A screenshot of a phone

AI-generated content may be incorrect.

**3. Column Mapping**

**- @Column(name = "emp\_name", length = 100, nullable = false, unique = true)**

Customizes the column properties.

**4. Relationships**

**- @OneToOne (One-to-One Relationship)**

**- @OneToMany (One-to-Many Relationship)**

**- @ManyToOne (Many-to-One Relationship)**

**- @ManyToMany (Many-to-Many Relationship)**

**@onetomany(cascade = cascadeType.ALL) 🡪 cascade- if we made any work on superclass (table) then it will also reflect on subclass of that created instance class & ALL gives all permission.**

**5. Other Useful Annotations**

**- @Transient**

**Excludes a field from being saved in the database.**

**-@Lob**

**Stores large objects like BLOBs (binary) or CLOBs (text).**

**-@Temporal**

**Stores date/time values.**

**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

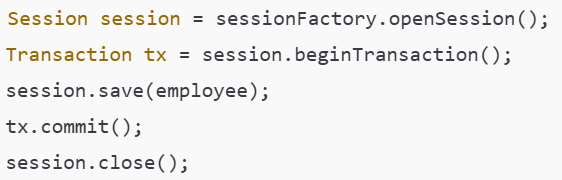
AI-generated content may be incorrect.**

**Core Interfaces of Hibernate-**

1. **SessionFactory-**

* A heavyweight, thread-safe object used to create Session instances.
* It is created **once per database** and shared across multiple sessions.
* Uses a configuration file (hibernate.cfg.xml) to set up the database connection.
* Example: 

1. **Session-**

* Represents a single unit of work with the database.
* Provides methods for CRUD operations (save, update, delete, find).
* Manages transactions and database connections.
* Example: session & transaction 

Session session = sessionFactory.opensession(); 🡪 it every time creates new object (new hashcode)

Session session = sessionFactory.getCurrentSession();

🡪 map.put(Environment.***CURRENT\_SESSION\_CONTEXT\_CLASS***, "thread");

- if already getCurrentSession is there then it will use that session object only (so same hashcode) if not present then it will create new session object.

1. **Transaction-**

* Manages database transactions in Hibernate.
* Allows committing or rolling back changes.

A close-up of a website

AI-generated content may be incorrect.

1. **Configuration-**

* Used to configure Hibernate and load mapping files.
* Creates SessionFactory.
* Example:

A computer screen shot of a computer code

AI-generated content may be incorrect.

**Differece between session.save() & session.persist()-**

1. **Session.save()-**

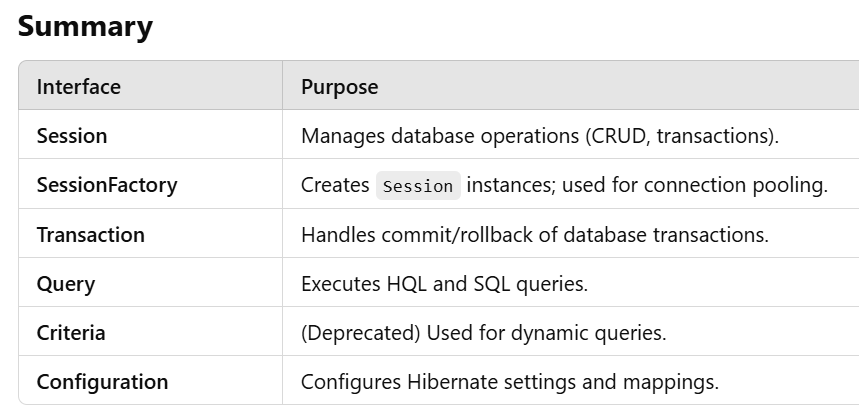
* It returns serializable primary key if its it auto generated primary key then it returns that value or else it will return inputed primary key.
* Can be used outside of a transaction.
* It inserts immediately but reflects in database after commiting.

1. **Session.persist()-**

* It’s return type is void.
* Must be called within an active transaction.
* Insert happens after commiting.

1. **Session.saveOrUpdate()-**

* It update the value if it’s present or else it will save new data.



**How it Works?**

1. **getSessionFactory()** → Retrieves the singleton SessionFactory instance.
2. **openSession()** → Creates a new Session (database connection).
3. **Perform Operations** → Perform CRUD operations (save, update, delete, etc.).
4. **session.close()** → Closes the session after use.
5. **shutdown()** → Closes SessionFactory when the application stops.

**Cache/ first level cache- session level cache (its defaultly on in hibernate )**

* cache is a phenomenon which pulling of an object
* java application(fires query for data fetch ) -> session -> SessionFactory -> DB again DB return the value as per query DB -> SessionFactory -> session -> java application & that value is stored in session
* if we again fire same query for same primary key using same session then the session already have that data so it directly gives that data to java application.
* **Session.clear();** 🡪 it clears all cache of that session.
* **Session.evict(obj);** 🡪 it will clear cache of that specific object.

**second level cache – SessionFactory level cache**

* java application(fires query for data fetch ) -> session -> SessionFactory -> DB again DB return the value as per query DB -> SessionFactory -> session -> java application & that value is stored in SessionFactory
* if we again fire same query for same primary key using different different session then the SessionFactory already have that data so it directly gives that data to session -> java application.

**How to use second level cache-**

* 1. first we have to add jar files for cache.
  2. map.put(Environment.CACHE\_REGION\_FACTORY, ”--use qualified name--”);
  3.  use this annotation on class name where you want to use second level cache.

**To clear cache there is 2 methods-**

1. **evictAll()**

* cache cache = sf.getcache(); 🡪use org.hibernate cache
* cache.evictAll(); 🡪 it will remove all cache from SessionFactory.

1. **evict(pass class)**

* cache cache = sf.getcache(); 🡪use org.hibernate cache
* cache.evict(class name.class); 🡪 it removes cache of that specific class

A screenshot of a computer

AI-generated content may be incorrect.

**HQL- Hibernate Query Language**

* it is an object-oriented query language.
* It is **database-independent** and provides powerful query capabilities.
* it always support JPQL (Java Persist Query Language)
* it is case sensitive
* We cannot use HQL queries to insert values into new table

**Hibernate Life Cycle-**

**A screenshot of a computer

AI-generated content may be incorrect.**

Detached state to Persistent State it goes when we call sessions get() or load() methods.

* **Named Queries-**

We can use @NamedQueries annotation at entity level we can use that query multiple times.

**How to use?**

* Just add one more annotation on pojo class
* *@Entity*
* *@NamedQueries*({*@NamedQuery*(query = "from Employee",name = "getAllEmployee"),*@NamedQuery*(query = "select name,salary from Employee where id=:id",name="getSpecificEmployeeNameAndSalary")})
* After that just use there name for that query use
* Query<Employee> query= session.createNamedQuery("getALLEmployee");
* List<Employee> list1 = query.getResultList();
* System.***out***.println(list1);

**How we can use any DB querys in Hibernate?**

* Just add one more annotation on pojo class
* *@Entity*

*@NamedNativeQueries*({*@NamedNativeQuery*(query="select name,address from employee where id=:id1",name="getSpeificEmployeeNameAndAddress")})

* After that just use there name for that query use

**Disadvantages-**

**Q. Hibernate N+1 Problem-**

🡪 The **N+1 problem** can severely impact performance due to excessive database queries.

* Use **JOIN FETCH** for eager fetching.

**Q. SQL Injection Attack-**

🡪 there is some query if we insert that qery in userid & password at the time of sql opening then it breaks sql.

**Q. Dirty Checking-**

🡪 it occurs at the time of inserting & updating if we are adding only single column value but on console the executed query is using all other columns of that table that is known as dirty checking.

* To avoid this use Annotation on class level(Entity)-
* @DynamicInsert 🡪only this 2 are hibernate annotation
* @DynamicUpdate