**Hibernate**

**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.

**The Hibernate configuration file**

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.**

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

**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?

**Hibernate State**

**Transient State**

* Transient state means object is neither associated with session nor present in database.

Employee emp = new Employee(1001,”TestA”,”IT”);

* In this state, object is non-transactional that means object is not synchronized with table record in database.
* In this state, changes made to objects don’t save into the database.

**Persistent State**

* Persistent state means object is associated with session and also present in the database.

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

Session.delete(emp); // Delete Persistence Objects

* Here Object is transactional that means object is synchronized with table record in database. Here modifications which are done to the entity, does not save into the database.
* In this state, Changes made to objects are automatically saved into database without invoking session persistence object.

**Detached State**

* Detached state means object is not associated with the session, but present in the database.

Employee emp = new Employee(1002,”TestB”,”IT”);

Session.update(emp);

* We need to call the update method when the object is in detached state.
* Here object is non-transactional that means object is not synch with database, so changes made to detached objects are not saved into the database.

**Dialect**

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

Dialect is used to convert HQL statements to database specific statement.

<property name=”hibernate.dialect”>org.hibernate.dialect.MySQL5Dialect

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

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

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.

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

**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.

**Transaction - org.hibernate.Transaction**

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

EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

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

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

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.

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

**Load () method vs Get () method**

|  |  |
| --- | --- |
| **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.

**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 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.

**Clear ()**

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

session.clear();

**evict ()**

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

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().

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

**Refresh ()**

 It is used to synchronize database data with session data.

session.refresh (employee);

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

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

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

**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.

**FETCH**

It defines whether mapped entity should be lazily or eagerly initialized.

Fetch=FetchType.LAZY

Fetch the related entity lazily from the database.

**CASCADE**

By Default, no operation is cascaded.

It defines the operation to be cascaded.

The **@Entity annotation** defines just the name attribute which is used to give a specific entity name for use in JPQL queries. - javax.persistence.Entity annotation.

The **@Id identifier** attribute does not necessarily need to be mapped to the column(s) that physically define the primary key. However, it should map to column(s) that can uniquely identify each row.

Here we **use @Column** to explicitly map the description attribute to the NOTES column, as opposed to the implicit column name description.

**mappedBy**

* It defines the field which owns the relationship.
* This element is specified on the now owning the side of association. This only required when the relationship is **uni directional**.
* Owning the side is the entity having foreign column.
* mappedBy indicates the inverse of the relationship

**@JoinColumn**

* @JoinColumn defines foreign key column and indicates the owner of the relationship.
* The owner of the relationship contains @JoinColumn annotation to specify for foreign column, the inverse side of the relationship contains mapped by attribute to indicate that the relationship is mapped by other entity.

**@JoinTable**

* @JoinTable defines the join table of two associated entries.
* If the JoinTable annotation is missing, the default values of the annotation elements apply.
* The name of the JoinTable is assumed to be table names of the associated primary table concatenated together (owning side first) using an underscore.

**@MapsId**

* @MapsId defines the embedded primary key.

**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.

**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

**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.

**Hibernate Caching**

**First-level cache**

Hibernate has the concept of first-level cache. 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 session cache caches objects within the current session. It is enabled by default in Hibernate.

**second-level cache**

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.

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.

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

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