**Hibernate Notes**

Hibernate is a **Java-based persistence framework** and an **object-relational mapping (ORM)** framework that basically allows a developer to map POJO - plain old Java objects - to relational database tables.

The aim of hibernate framework is to free the developer from the common data persistence-related complex configurations and tasks. It does so by mapping the POJO objects with the database tables efficiently and most importantly in an abstract manner.

The developer need not know the underlying complications involved. Along with abstraction, the queries can be executed in a very efficient manner. All these helps developers to save a lot of time involved in development.

**Hibernate ORM** stands for **Object Relational Mapping**. This is a mapping tool pattern mainly used for converting data stored in a relational database to an object used in object-oriented programming constructs. ***This tool also helps greatly in simplifying data retrieval, creation, and manipulation.***

Diagram

Description automatically generatedObject Relational Mapping

**Advantages of Hibernate**

The advantages of Hibernate over JDBC are listed below:

* **Clean Readable Code:** Using hibernate, helps in eliminating a lot of JDBC API-based boiler-plate codes, thereby making the code look cleaner and readable.
* **HQL (Hibernate Query Language):** Hibernate provides HQL which is closer to Java and is object-oriented in nature. This helps in reducing the burden on developers for writing database independent queries. In JDBC, this is not the case. A developer has to know the database-specific codes.
* **Transaction Management:** JDBC doesn't support implicit transaction management. It is upon the developer to write transaction management code using commit and rollback methods. Whereas, Hibernate implicity provides this feature.
* **Exception Handling:** **Hibernate wraps the JDBC exceptions and throws unchecked exceptions like JDBCException or HibernateException. This along with the built-in transaction management system helps developers to avoid writing multiple try-catch blocks to handle exceptions. In the case of JDBC, it throws a checked exception called SQLException thereby mandating the developer to write try-catch blocks to handle this exception at compile time.**
* **Special Features:** Hibernate supports OOPs features like inheritance, associations and also supports collections. These are not available in JDBC

**States of Hibernate Object**

The [*Session*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Session.html) interface is the main tool used to communicate with Hibernate. It provides an API enabling us to create, read, update, and delete persistent objects. The *session* has a simple lifecycle. We open it, perform some operations, and then close it.

When we operate on the objects during the *session*, they get attached to that *session*. The changes we make are detected and saved upon closing. After closing, Hibernate breaks the connections between the objects and the session.

**Transient**

**An object we haven't attached to any session is in the transient state.** Since it was never persisted, it doesn't have any representation in the database. Because no session is aware of it, it won't be saved automatically.

**Persistent**

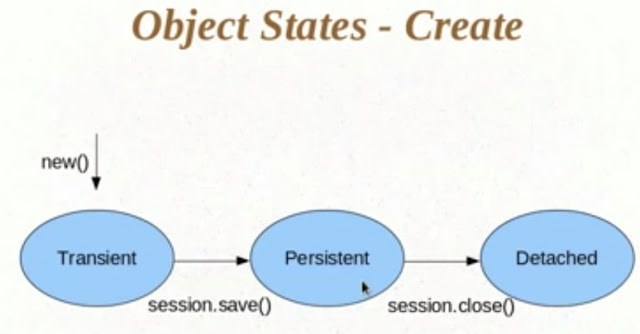
**An object that we've associated with a session is in the persistent state.** We either saved it or read it from a persistence context, so it represents some row in the database.

Alternatively, we may use the save method. The difference is that the persist method will just save an object, and the save method will additionally generate its identifier if that's needed.

**Detached**

When we close the session, all objects inside it become detached. **Although they still represent rows in the database, they're no longer managed by any session:**

## 1. When creating a new Entity Object

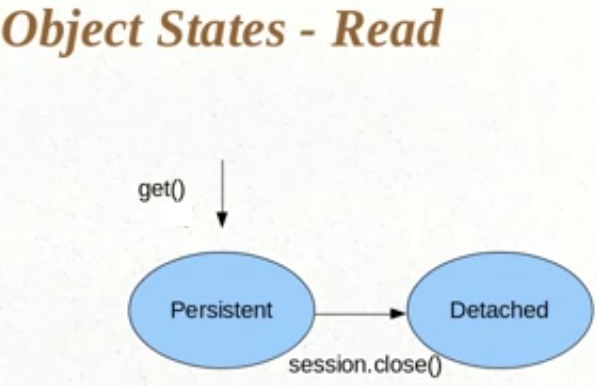


Here we see when create an object it is in **transient**state in this state hibernate does not ask for save this object means that in this state hibernate’s session does not associate with that object in this state.

Once we calling session’s save method now object move to the **persistent**state i.e. hibernate’s session associated with that object in this state if any change made in the object hibernate ask to the database and made change in the database also.

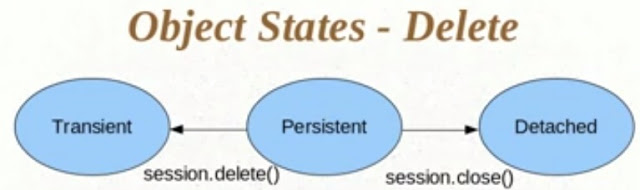
After done our required events when we calling session’s close method then object moves in the  **detached**state. k

## 2. When Reading an Entity Object



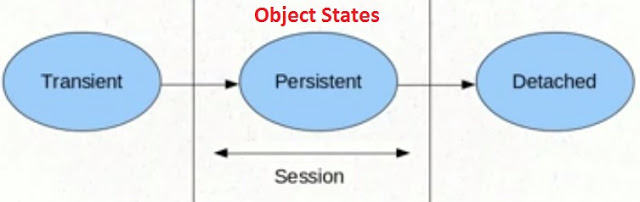
Here we see that we are not getting new object from **new op**erator, we get the object from session’s get method. Here we pass a primary key in the get method and getting a persistent object.

## 3. When Delete an Entity Object



Here after getting persistent object from session of hibernate. When we are calling the delete method of the session object moves from persistent state to the transient state. If we calling the close method of the session then that object moves to the detached state. If once object move to the transient state it never become a persistent object.

Now we look the states diagram of the entity object in the following.



When object in the session area then it is in **Persistent State.**  
When object before the session area then it is in **Transient State.**  
Whenobject after the session area then it is in **Detached State.**

**Component Mapping**

A **Component** mapping is a mapping for a class having a reference to another class as a member variable. We have seen such mapping while having two tables and using <set> element in the mapping file. Now we will use <component> element in the mapping file and a single table would be used to keep the attributes contained inside the class variable.

If one class having reference to another class as a member variable, we can use Component mapping.

## **What is Collection Mapping?**

Sometimes, we will have a Collection of values associated with an entity. we may want to persist the values in the Collection but may not want to create a separate Entity corresponding to the Collection. we can achieve this via Hibernate/JPA Collection mapping. Hibernate Collection Mapping feature allows us to map any type of Collection like List, Set or Map.

## How Collection Mapping works

JPA provides an annotation called

@ElementCollection

 in order to support collection mapping. we need to specify this annotation on the Collection that we want to persist to the database. Using collection mapping, we can map a collection having primitive type of data as well as non-primitive type (object) data

## When should you use Collection Mapping

Although collection mapping looks easy to use, it has some disadvantages. For one, the elements in the collection cannot be saved or queried individually. Secondly, if the collection is modified and the entity is saved, Hibernate deletes all the records from the corresponding table and inserts them all over again. so this is not very efficient So we should use collection mappings only for very small collections. In all other cases, a one-to-many association is the better approach.

**Difference Between save and persist**

|  |  |  |
| --- | --- | --- |
| **Basis of Difference** | **Save in Hibernate** | **Persist in Hibernate** |
| **Return type** | The return type pertaining to the save () | The return type of persist () method is void. |
| is serializable object. |
| **Assigning of identifier value** | Th save() method allows for the assigning of identifier value instantly. | The persist() method fails to guarantee that an identifier value is assigned to its persistent state instantly. |
| **Execution of insert query** | The save() method provides an identifier with the intent of an insert query being executed immediately for getting the identifier. It does not matter whether it is outside or inside a transaction. | The persist() method fails to execute a given insert query in case it is placed outside transaction boundaries. |
| **Utility** | The save method proves to be of less use in a long-running conversation that has extended a given Session context. | As the persist method is called outside the transaction boundaries, it is utilized in long-running conversations that offer an extended Session context. |
| **Support** | Save() method gets support only through Hibernate. | Persist() method is aptly supported by JPA. |

**Difference between get() and load()**

#### session.get()

* session.get() method always hits database and returns actual object
* It returns null in case it does not get object.

#### session.load()

* session.load() method always does not hit database and returns proxy object
* It throws  ObjectNotFoundException  in case it does not get object.

### get vs load in hibernate

|  |  |  |
| --- | --- | --- |
| **Parameter** | **get** | **load** |
| Database retrieval | It always hits the database | It does not hit database |
| If null | If it does not get the object with id, it returns null | If it does not get the object with id, it throws ObjectNotFoundException |
| Proxy | It returns real object | It returns proxy object |
| Use | If you are not sure if object with id exists or not, you can use get | If you are sure about existence of object, you can use load |

**Explain Inheritance strategies in Hibernate**

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

#### **Table Per Hierarchy**

In table per hierarchy mapping, single table is required to map the whole hierarchy, ***an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table.***

#### **Table Per Concrete class**

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

#### **Table Per Subclass**

In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

* **Associan strategies:-**
* **one to one**— it represents the one to one relationship between two tables.
* **one to many/many to one**— it represents the one to many relationship between two tables.
* **many to many**— it represents the many to many relationship between two tables.

**Different types of Caching in Hibernate**

**Hibernate Cache can be very useful in gaining fast application performance if used correctly*. The idea behind cache is to reduce the number of database queries, hence reducing the throughout time of the application****.* Hibernate comes with different types of Cache:

1. **First Level Cache**: Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely. Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.
2. **Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.
3. **Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

# Unidirectional Relationships:

* Unidirectional is a relation where one side does not know about the relation.
* In a unidirectional relationship, only one entity has a relationship field or property that refers to the other. For example, Line Item would have a relationship field that identifies Product, but Product would not have a relationship field or property for Line Item. In other words, Line Item knows about Product, but Product doesn’t know which Line Item instances refer to it.

# Bidirectional Relationships:

* Bidirectional relationship provides navigational access in both directions, so that you can access the other side without explicit queries.
* In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class’s code can access its related object. If an entity has a related field, the entity is said to “know” about its related object. For example, if Order knows what Line Item instances it has and if Line Item knows what Order it belongs to, they have a bidirectional relationship.

**Explain HQL**

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.

**Criteria API**

The Criteria API is a predefined API used to define queries for entities. It is the alternative way of defining a JPQL query. These queries are type-safe, and portable and easy to modify by changing the syntax. Similar to JPQL it follows abstract schema (easy to edit schema) and embedded objects. The metadata API is mingled with criteria API to model persistent entity for criteria queries.

The major advantage of the criteria API is that errors can be detected earlier during compile time. String based JPQL queries and JPA criteria based queries are same in performance and efficiency.

The query demonstrates the basic steps to create a criteria.

* EntityManager instance is used to create a CriteriaBuilder object.
* CriteriaQuery instance is used to create a query object. This query object’s attributes will be modified with the details of the query.
* CriteriaQuery.from method is called to set the query root.
* CriteriaQuery.select is called to set the result list type.
* TypedQuery<T> instance is used to prepare a query for execution and specifying the type of the query result.
* getResultList method on the TypedQuery<T> object to execute a query. This query returns a collection of entities, the result is stored in a List.

A named query is a statically defined query with a predefined unchangeable query string. They're validated when the session factory is created, thus making the application to fail fast in case of an error.

**What are entities in Hibernate?**

An entity **represents a table stored in a database**. Every instance of an entity represents a row in the table

|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| JPA is described in **javax.persistence** package. | Hibernate is described in **org.hibernate** package. |
| It describes the handling of relational data in Java applications. | Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system. |
| It is not an implementation. It is only a Java specification. | Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate. |
| It is a standard API that permits to perform database operations. | It is used in mapping Java data types with SQL data types and database tables. |
| As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations. | As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations. |
| To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager. | To create Session instances, it uses **SessionFactory** interface. |
| To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnect**s** with the persistence condition. | To make, read, and remove actions for instances of mapped entity classes, it uses **Session**interface. It acts as a runtime interface between a Java application and Hibernate. |
| ***Conclusion:****The major difference between Hibernate and JPA is that Hibernate is a framework while JPA is API specifications. Hibernate is the implementation of all the JPA guidelines.* |  |

### ****@Entity****

* @Entity annotation (**javax.persistence.Entity**) over the class tell hibernate to treat this class as our entity class that needs to be saved.
* The hibernate specific Entity annotation **org.hibernate.annotations.Entity has been deprecated** in hibernate 4.
  + We have to use annotations corresponding to its attributes/values to get the same result.
    - For instance, @org.hibernate.annotations.Entity(selectBeforeUpdate=true) is replaced by @SelectBeforeUpdate(true).

### ****@Entity (name=”XYZ”)****

* This will change the identity name to XYZ.
  + By default, hibernate generate the table name same as classname with only an @Enity annotation.
  + With @Entity(name=”XYZ”) and in the absence of an @Table annotation to change the table name, the entity name will be used to create table instead of class name.

### ****@Table(name=”XYZ”))****

* @Table(name=”XYZ”)) annotation (**javax.persistence.Table** /**org.hibernate.annotations.Table**) tell hibernate to create table with name XYZ instead of class name.
* @Table(name=”XYZ”)) is different from @Entity (name=”XYZ”) in that @Table changes only table name and entity name still remains the default, which is class name.
* One of the applications of this difference between entity names and table names is the use of them within HQL(Hibernate Query Language).

### ****@Id****

* The @Id (**javax.persistence.Id**) over the id field tell hibernate to make this field as the primary key of the table.

### ****@GeneratedValue****

* If you use @GeneratedValue, then hibernate will automatically generate values for that using an internal sequence.
  + Therefore you don’t have to set it manually.
* You can use it along with @Id and make your primary key automatically generated rather than setting each time.
* You can also specify a strategy on how the values will be generated like
  + @GeneratedValue(strategy=GenerationType.AUTO).
    - AUTO is the default if you don’t give a strategy.
    - AUTO is the preferred option as hibernate will chose the best strategy for us automatically.
  + Other values are IDENTITY, SEQUENCE and TABLE, which are actually dependent on the database that you use.

### ****@Column****

Can be used to specify column mappings. For example, to change the column name in the associated table in database.

### ****@Basic****

* @Basic without any parameters is same as without having it.
* It just tells hibernate to treat it is a field that needs to be saved and is the same behavior without it.
* However the use comes when it is used along with its parameters.
* By default it is same as @Basic (optional=true)
  + the field may not be supplied any value (not calling getter) or supplied with a null value.
* @Basic (optional=false) makes it a non-null field.
  + If not supplied any value (not calling getter) or supplied with a null value with @Basic (optional=false), you will get an exception: org.hibernate.PropertyValueException: not-null property references a null or transient value.

### ****@Transient****

* This tells hibernate not to save this field.
  + Even without this annotation, a static variable or a transient variable is not saved.
  + So the behavior is same for a static variable, transient variable or any other variable with @Transient annotation.
* If annotations are placed over getters, those placed over fields are ignored and only getters are considered.
  + Even other fields are ignored and all other getters are considered.
  + Hence if you have annotations over any of your getter, then a static variable, transient variable or any other variable will be saved as usual if it has a getter.

### ****@Temporal****

* @Temporal over a date field tells hibernate the format in which the date needs to be saved. For instance, by default the date will be saved as a timestamp, but to save date alone we can use @Temporal(TemporalType.DATA).

### ****@Lob****

* @Lob tells hibernate that this is a large object, not a simple object.
  + So hibernate creates a CLOB or BLOB based on the type of the object.
    - For instance, if @Lob comes over a string, then hibernate assumes to use a character large object (CLOB).

**@OnetoOne**

* Use @PrimaryKeyJoinColumn for associated entities sharing the same primary key.
* Use @JoinColumn & @OneToOne mappedBy attribute when foreign key is held by one of the entities.
* Use @JoinTable and mappedBy entities linked through an association table.
* Persist two entities with shared key using @MapsId

### @ManyToOne

* Use @JoinColumn when foreign key is held by one of the entities.
* Use @JoinTable for entities linked through an association table.

**@OneToMany**

|  |  |
| --- | --- |
|  | * Use mappedBy attribute for bi-directional associations with ManyToOne being the owner. * OneToMany being the owner or unidirectional with foreign key - try to avoid such associations but can be achieved with @JoinColumn * @JoinTable for Unidirectional with association table |

### @ManyToMany

|  |  |
| --- | --- |
| Hibernate Annotation Tip | * Use @JoinTable for entities linked through an association table. * Use mappedBy attribute for bi-directional association. |

**@JoinColumn**

Use @JoinColumn annotation for one-to-one or many-to-one associations when foreign key is held by one of the entities. We can use @OneToOne or @ManyToOne mappedBy attribute for bi-directional relations. Also see [OneToOne](https://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "OneToOne" \o "Hibernate JPA Annotation @OneToOne) and [ManyToOne](https://www.techferry.com/articles/hibernate-jpa-annotations.html" \l "ManyToOne" \o "Hibernate JPA Annotation @ManyToOne) sections for more details.

@ManyToOne

@JoinColumn(name = "statusId")

private CompanyStatus status;

**@JoinTable**

Use @JoinTable and mappedBy for entities linked through an association table.

| **Sr. No.** | **Key** | **JPARepository** | **CrudRepository** |
| --- | --- | --- | --- |
| 1 | Hierarchy | JPA extend crudRepository and PagingAndSorting repository | Crud Repository is the base interface and it acts as a marker interface. |
| 2 | Batch support | JPA also provides some extra methods related to JPA such as delete records in batch and flushing data directly to a database. | It provides only CRUD functions like findOne, saves, etc. |
| 3 | Pagination support | JPA repository also extends the PagingAndSorting repository. It provides all the method for which are useful for implementing pagination. | Crud Repository doesn't provide methods for implementing pagination and sorting. |
| 4 | Use Case | JpaRepository ties your repositories to the JPA persistence technology so it should be avoided. | We should use CrudRepository or PagingAndSortingRepository depending on whether you need sorting and paging or not. |