Question:

Orm over jdbc

Difference between get() and load(),

Persist() or save()

Inheritance mapping:

get() vs load():

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Get()** | **Load()** |
| 1 | Basic | It  is used to fetch data from the database for the given identifier | It  is also used to fetch data from the database for the given identifier |
| 2 | Null Object | It object not found for the given identifier then it will return null object | It will throw object not found exception |
| 3 | Lazy or Eager loading | It returns fully initialized object so this method eager load the object | It always returns proxy object so this method is lazy load the object |
| 4 | Performance | It is slower than load() because it return fully initialized object which impact the performance of the application | It is slightly faster. |
| 5. | Use Case | If you are not sure that object exist then use get() method | If you are sure that object exist then use load() method |

|  |  |
| --- | --- |
| load() just returns a proxy by default and database won't be hit until the proxy is first invoked.    **Proxy** means, hibernate will prepare some fake object with given identifier value in the memory without hitting a database. | get() will hit the database immediately. |

Persist() vs Save()

1. The first difference between save and persist is there return type. Similar to save method, persist also INSERT records into the database, but **return type of persist is void** while return type of save is Serializable Object which returns primary key value.

1. Another difference between persisting and save is that both methods make a transient instance persistent. However, persist() method doesn't guarantee that the identifier value will be assigned to the persistent instance immediately, the assignment might happen at flush time.

1. One more thing which differentiates persist and save method in Hibernate is that it is their behavior on the outside of transaction boundaries. persist() method guarantees that it will not execute an INSERT statement if it is called outside of transaction boundaries. save() method does not guarantee the same, it returns an identifier, and if an INSERT has to be executed to get the identifier (like "identity" generator), this INSERT happens immediately, no matter if you are inside or outside of a transaction.

1. The fourth difference between save and persist method in Hibernate is related to previous differences in saving vs. persist. Because of its above behavior of persist method outside transaction boundary, it's useful in long-running conversations with an extended Session context. On the other hand, the save method is not good in a long-running conversation with an extended Session context.

Object States in hibernate:

1. 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.

*Eg:get(), load()*

1. 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.

1. 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*:**

Component Mapping:

In component mapping, we will map the dependent object as a component. An component is an object that is stored as an value rather than entity reference. This is mainly used if the dependent object doen't have primary key. It is used in case of composition (HAS-A relation), that is why it is termed as component.

we have marked our **EmployeeAddress** class with **@Embeddable annotation** so that this class is eligible to be an **embeddable** class. **@Embeddable** annotation is used to specify that the **EmployeeAddress** class will be used as a component. **EmployeeAddress** cannot have a primary key of its own and it will be using **Employee** class primary key.

**@Embedded**–**@Embedded** annotation is used to specify that the **EmployeeAddress** entity should be stored in the **EMPLOYEE** table as a component.

@Embeddable: Specifies a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity.Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

Collection Mapping:

In entity class i.e user have collection

* + **@ElementCollection** - This annotation is placed just above the collection object property, which tells the Hibernate that the collection object should be persisted as a collection and should not be embedded in the same table as the Entity objects, but as a separate table in the database.
  + **@JoinTable** - This annotations is **used to name the database table** that contains the collection object or a table that is associated with the Entity object table through mapping(*one-to-one, one-to-many etc*).

*In collection object use @embeddable*

shows that each object of Contact\_Info class is stored in a separate table than an object of the Entity class.(one to many)

Inheritance strategy in hibernate:

We can map the inheritance hierarchy classes with the table of the database.

Single table:

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 .

Diagram

Description automatically generated

For single table:

* + Create the persistent classes
  + Edit pom.xml file
  + Create the configuration file
  + Create the class to store the fetch the data

@Entity

@Table(name = "employee101")

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

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

@DiscriminatorValue(value="employee")

DiscriminatorValue:

In case of table per class hierarchy an discriminator column is added by the hibernate framework that **specifies the type of the record**. It is mainly used to distinguish the record. To specify this, discriminator subelement of class must be specified. The subclass subelement of class, specifies the subclass.

DiscriminatorColumn:

 Since the records for all entities will be in the same table, Hibernate needs a way to differentiate between them. By default, this is done through a discriminator column called **DTYPE that has the name of the entity as a value**.

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.

**@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)** specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

**@AttributeOverrides** defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

Table structure for Employee class

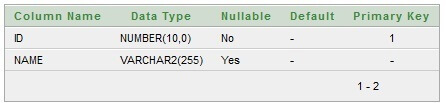


Table structure for Regular\_Employee class



Table structure for Contract\_Employee class

Table

Description automatically generated

@AttributeOverrides({

    @AttributeOverride(name="id", column=@Column(name="id")),

    @AttributeOverride(name="name", column=@Column(name="name"))

})

Table Per Subclass

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

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.

We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn** annotation in the subclasses.

Table structure for Employee class

Table

Description automatically generated

Table structure for Regular\_Employee class

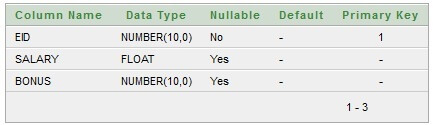


Table structure for Contract\_Employee class

Table

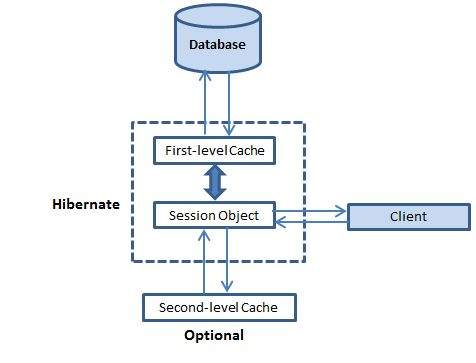
Description automatically generated

@PrimaryKeyJoinColumn(name="ID")  ->in child class.

Caching in Hibernate:

Hibernate caching improves the performance of the application by pooling the object in the cache. It is useful when we have to fetch the same data multiple times.

Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database. Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.



There are mainly two types of caching:

* First Level Cache, and
* Second Level Cache

First Level Cache:

Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second Level Cache:

SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitely.

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

**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.

1. EH Cache(easy hibernate)
2. OS Cache
3. Swarm Cache
4. JBoss Cache

Each implementation provides different cache usage functionality. There are four ways to use second level cache.

**read-only:** caching will work for read only operation.

**nonstrict-read-write:** caching will work for read and write but one at a time.

**read-write:** caching will work for read and write, can be used simultaneously.

**transactional:** caching will work for transaction.

@Entity

@Table(name="emp1012")

@Cacheable

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY)

**public** **class** Employee {

 hibernate does not fire query twice. If you don't use second level cache, hibernate will fire query twice because both query uses different session objects.

@Cacheable =

 Annotation **indicating that the result of invoking a method (or all methods in a class) can be cached**. Each time an advised method is invoked, caching behavior will be applied, checking whether the method has been already invoked for the given arguments.

@Cache=

Types of Association Mapping:

When only one out of the two entities contains a reference to the other, the association is uni-directional. If the association is mutual and both entities refer to one another, it is bi-directional.

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.

Although you can use SQL statements directly with Hibernate using Native SQL, but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in HQL.

Named query:

A major disadvantage of having HQL and SQL scattered across data access objects is that it makes the code unreadable. Hence, it might make sense to group all HQL and SQL in one place and use only their reference in the actual data access code. Fortunately, Hibernate allows us to do this with named queries.

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.

*@NamedQuery:*

**It's important to note that every *@NamedQuery*annotation is attached to exactly one entity class or mapped superclass. But,** **since the scope of named queries is the entire persistence unit, we should select the query name carefully to avoid a collision.** And we have achieved this by using the entity name as a prefix.

@NamedNativeQuery:

**Since this is a native query, we'll have to tell Hibernate what entity class to map the results to. Consequently, we've used the *resultClass*property for doing this.**

**Another way to map the results is to use the *resultSetMapping*property. Here, we can specify the name of a pre-defined *SQLResultSetMapping*.**

Note that we can use only one of *resultClass* and *resultSetMapping*.

@org.hibernate.annotations.NamedNativeQueries( @org.hibernate.annotations.NamedNativeQuery(name = "DeptEmployee\_GetEmployeeByName", query = "select \* from deptemployee emp where name=:name", resultClass = DeptEmployee.class) )

Notice that although this is an update query, we've used the *resultClass*property. This is because Hibernate doesn't support pure native scalar queries. And the way to work around the problem is to either set a *resultClass*or a *resultSetMapping.*

ORM vs JDBC:

|  |  |
| --- | --- |
| Object Relational Mapping | Java Database Connectivity |
| Little slower than JDBC | It is faster compared to ORM |
| SQL queries requirement is comparatively quite less however this doesn’t mean that you have to do less work using ORM | SQL queries are required here |
| Hibernate framework (working on ORM technology) makes it easy to store objects/data to database automatically without writing manual code | We have to write code manually to store objects/ data in the database |
| The flow from Object/data to hibernate i.e. the frontend part is based on the ORM technique | Whereas when the data is stored in the database finally i.e., the backend part is still based on JDBCin |
| There are not many restrictions while dealing with data. Even a single database cell can be retrieved, changed, and saved. | JDBC comes with a lot of restrictions on extracting the result-set, processing it, and then committing it back to the database. |