**Hibernate interview questions(by mohan\_\_oon)**

**What is ORM in Hibernate?**

**What are the advantages of Hibernate over JDBC?**

**What are some of the important interfaces of Hibernate framework?**

**What is a Session in Hibernate?**

**Can you explain what is lazy loading in hibernate?**

**What is the difference between first level cache and second level cache?**

**Can you explain the concept behind Hibernate Inheritance Mapping?**

**What are the most commonly used annotations available to support hibernate mapping?**

**Differentiate between get() and load() in Hibernate session**

**What is criteria API in hibernate?**

**What is HQL?**

**Does Hibernate support Native SQL Queries?**

**Can you tell something about the N+1 SELECT problem in Hibernate?**

**How to solve N+1 SELECT problem in Hibernate?**

**What is Single Table Strategy?**

**-------------------------------------------------------------------------------------------------**

**What is ORM in Hibernate?**

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.

**What are the advantages of Hibernate over JDBC?**

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.

What are some of the important interfaces of Hibernate framework?

Hibernate core interfaces are:

Configuration

SessionFactory

Session

Criteria

Query

Transaction

**What is a Session in Hibernate?**

A session is an object that maintains the connection between Java object application and database. Session also has methods for storing, retrieving, modifying or deleting data from database using methods like persist(), load(), get(), update(), delete(), etc. Additionally, It has factory methods to return Query, Criteria, and Transaction objects.

**What is a SessionFactory?**

SessionFactory provides an instance of Session. It is a factory class that gives the Session objects based on the configuration parameters in order to establish the connection to the database. As a good practice, the application generally has a single instance of SessionFactory. The internal state of a SessionFactory which includes metadata about ORM is immutable, i.e once the instance is created, it cannot be changed.

This also provides the facility to get information like statistics and metadata related to a class, query executions, etc. It also holds second-level cache data if enabled.

**Can you explain what is lazy loading in hibernate?**

Lazy loading is mainly used for improving the application performance by helping to load the child objects on demand.

It is to be noted that, since Hibernate 3 version, this feature has been enabled by default. This signifies that child objects are not loaded until the parent gets loaded.

What is the difference between first level cache and second level cache?

Hibernate has 2 cache types. First level and second level cache for which the difference is given below:

First Level Cache Second Level Cache

This is local to the Session object and cannot be shared between multiple sessions. This cache is maintained at the SessionFactory level and shared among all sessions in Hibernate.

This cache is enabled by default and there is no way to disable it. This is disabled by default, but we can enable it through configuration.

The first level cache is available only until the session is open, once the session is closed, the first level cache is destroyed. The second-level cache is available through the application’s life cycle, it is only destroyed and recreated when an application is restarted.

If an entity or object is loaded by calling the get() method then Hibernate first checked the first level cache, if it doesn’t find the object then it goes to the second level cache if configured. If the object is not found then it finally goes to the database and returns the object, if there is no corresponding row in the table then it returns null.

**Can you explain the concept behind Hibernate Inheritance Mapping?**

Java is an Object-Oriented Programming Language and Inheritance is one of the most important pillars of object-oriented principles. To represent any models in Java, inheritance is most commonly used to simplify and simplify the relationship. But, there is a catch. Relational databases do not support inheritance. They have a flat structure.

There are different inheritance mapping strategies available:

Single Table Strategy

Table Per Class Strategy

Mapped Super Class Strategy

Joined Table Strategy

**What are the most commonly used annotations available to support hibernate mapping?**

javax.persistence.Entity: This annotation is used on the model classes by using “@Entity” and tells that the classes are entity beans.

javax.persistence.Table: This annotation is used on the model classes by using “@Table” and tells that the class maps to the table name in the database.

javax.persistence.Access: This is used as “@Access” and is used for defining the access type of either field or property. When nothing is specified, the default value taken is “field”.

javax.persistence.Id: This is used as “@Id” and is used on the attribute in a class to indicate that attribute is the primary key in the bean entity.

javax.persistence.EmbeddedId: Used as “@EmbeddedId” upon the attribute and indicates it is a composite primary key of the bean entity.

javax.persistence.Column: “@Column” is used for defining the column name in the database table.

javax.persistence.GeneratedValue: “@GeneratedValue” is used for defining the strategy used for primary key generation. This annotation is used along with javax.persistence.GenerationType enum.

javax.persistence.OneToOne: “@OneToOne” is used for defining the one-to-one mapping between two bean entities. Similarly, hibernate provides OneToMany, ManyToOne and ManyToMany annotations for defining different mapping types.

org.hibernate.annotations.Cascade: “@Cascade” annotation is used for defining the cascading action between two bean entities. It is used with org.hibernate.annotations.CascadeType enum to define the type of cascading.

Differentiate between get() and load() in Hibernate session

get() load()

This method gets the data from the database as soon as it is called. This method returns a proxy object and loads the data only when it is required.

The database is hit every time the method is called. The database is hit only when it is really needed and this is called Lazy Loading which makes the method better.

The method returns null if the object is not found. The method throws ObjectNotFoundException if the object is not found.

This method should be used if we are unsure about the existence of data in the database. This method is to be used when we know for sure that the data is present in the database.

**What is criteria API in hibernate?**

Criteria API in Hibernate helps developers to build dynamic criteria queries on the persistence database. Criteria API is a more powerful and flexible alternative to HQL (Hibernate Query Language) queries for creating dynamic queries.

This API allows to programmatically development criteria query objects. The org.hibernate.Criteria interface is used for these purposes. The Session interface of hibernate framework has createCriteria() method that takes the persistent object’s class or its entity name as the parameters and returns persistence object instance the criteria query is executed.

It also makes it very easy to incorporate restrictions to selectively retrieve data from the database. It can be achieved by using the add() method which accepts the org.hibernate.criterion.Criterion object representing individual restriction.

To return all the data of InterviewBitEmployee entity class.

Criteria criteria = session.createCriteria(InterviewBitEmployee.class);

List<InterviewBitEmployee> results = criteria.list();

To retrive objects whose property has value equal to the restriction, we use Restrictions.eq() method. For example, to fetch all records with name ‘Hibernate’:

Criteria criteria= session.createCriteria(InterviewBitEmployee.class);

criteria.add(Restrictions.eq("fullName","Hibernate"));

List<InterviewBitEmployee> results = criteria.list();

To get objects whose property has the value “not equal to” the restriction, we use Restrictions.ne() method. For example, to fetch all the records whose employee’s name is not Hibernate:

Criteria criteria= session.createCriteria(InterviewBitEmployee.class);

criteria.add(Restrictions.ne("fullName","Hibernate"));

List<Employee> results = criteria.list()

To retrieve all objects whose property matches a given pattern, we use Restrictions.like() (for case sensitivenes) and Restrictions.ilike()(for case insensitiveness)

Criteria criteria= session.createCriteria(InterviewBitEmployee.class);

criteria.add(Restrictions.like("fullName","Hib%",MatchMode.ANYWHERE));

List<InterviewBitEmployee> results = criteria.list();

**What is HQL?**

Hibernate Query Language (HQL) is used as an extension of SQL. It is very simple, efficient, and very flexible for performing complex operations on relational databases without writing complicated queries. HQL is the object-oriented representation of query language, i.e instead of using table name, we make use of the class name which makes this language independent of any database.

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

List<InterviewBitEmployee> list=query.list();

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

**Does Hibernate support Native SQL Queries?**

Yes, it does. Hibernate provides the createSQLQuery() method to let a developer call the native SQL statement directly and returns a Query object.

Consider the example where you want to get employee data with the full name “Hibernate”. We don’t want to use HQL-based features, instead, we want to write our own SQL queries. In this case, the code would be:

Query query = session.createSQLQuery( "select \* from interviewbit\_employee ibe where ibe.fullName = :fullName")

.addEntity(InterviewBitEmployee.class)

.setParameter("fullName", "Hibernate"); //named parameters

List result = query.list();

Alternatively, native queries can also be supported when using NamedQueries.

**Can you tell something about the N+1 SELECT problem in Hibernate?**

N+1 SELECT problem is due to the result of using lazy loading and on-demand fetching strategy. Let's take an example. If you have an N items list and each item from the list has a dependency on a collection of another object, say bid. In order to find the highest bid for each item while using the lazy loading strategy, hibernate has to first fire 1 query to load all items and then subsequently fire N queries to load big of each item. Hence, hibernate actually ends up executing N+1 queries.

**How to solve N+1 SELECT problem in Hibernate?**

Some of the strategies followed for solving the N+1 SELECT problem are:

Pre-fetch the records in batches which helps us to reduce the problem of N+1 to (N/K) + 1 where K refers to the size of the batch.

Subselect the fetching strategy

As last resort, try to avoid or disable lazy loading altogether.

**What is Single Table Strategy?**

Single Table Strategy is a hibernate’s strategy for performing inheritance mapping. This strategy is considered to be the best among all the other existing ones. Here, the inheritance data hierarchy is stored in the single table by making use of a discriminator column which determines to what class the record belongs.

@Entity

@Table(name = "InterviewBitEmployee")

@Inheritance(strategy = InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name = "employee\_type")

@NoArgsConstructor

@AllArgsConstructor

public class InterviewBitEmployee {

@Id

@Column(name = "employee\_id")

private String employeeId;

private String fullName;

private String email;

}

**Can you tell something about Table Per Class Strategy.**

Table Per Class Strategy is another type of inheritance mapping strategy where each class in the hierarchy has a corresponding mapping database table. Hibernate provides @Inheritance annotation which takes strategy as the parameter. This is used for defining what strategy we would be using. By giving them value, InheritanceType.TABLE\_PER\_CLASS, it signifies that we are using a table per class strategy for mapping.

@Entity(name = "interviewbit\_employee")

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

@NoArgsConstructor

@AllArgsConstructor

public class InterviewBitEmployee {

@Id

@Column(name = "employee\_id")

private String employeeId;

private String fullName;

private String email;

}

**Can you tell something about Named SQL Query**

A named SQL query is an expression represented in the form of a table. Here, SQL expressions to select/retrieve rows and columns from one or more tables in one or more databases can be specified. This is like using aliases to the queries.

In hibernate, we can make use of @NameQueries and @NameQuery annotations.

@NameQueries annotation is used for defining multiple named queries. @NameQuery annotation is used for defining a single named query.

@NamedQueries(

{

@NamedQuery(

name = "findIBEmployeeByFullName",

query = "from InterviewBitEmployee e where e.fullName = :fullName"

)

}

)

Usage:

TypedQuery query = session.getNamedQuery("findIBEmployeeByFullName");

query.setParameter("fullName","Hibernate");

List<InterviewBitEmployee> ibEmployees = query.getResultList();