1. Introduction to Hibernate Architecture

Q-1 What is Hibernate?:

- Definition and purpose of Hibernate as an ORM (Object Relational Mapping) tool.

ANS - Definition of Hibernate as an ORM (Object-Relational Mapping) Tool:

Hibernate is an open-source Java-based framework that provides an Object-Relational Mapping (ORM) solution for Java applications. ORM is a technique that allows you to map Java objects to database tables and vice versa, effectively bridging the gap between object-oriented programming (OOP) and relational databases. Hibernate simplifies database interaction by handling the conversion of data between Java objects and database tables, eliminating the need for manual SQL queries for basic CRUD (Create, Read, Update, Delete) operations.

- Purpose of Hibernate as an ORM Tool:

1. Simplifying Database Interaction: Hibernate abstracts the complexity of direct SQL operations and allows developers to interact with databases using high-level object-oriented concepts, such as Java classes, objects, and collections, instead of writing verbose SQL statements.

2.Object-Relational Mapping: Hibernate automatically maps Java objects to relational database tables. The developer doesn't need to manually convert Java objects to SQL queries. It handles the conversion of object attributes to table columns, making the database interaction seamless.

3. Data Persistence: Hibernate provides a robust mechanism for persisting Java objects (i.e., storing object data in a relational database) and managing the state of objects (transient, persistent, detached).

4. Automatic SQL Generation: Hibernate generates efficient SQL queries automatically based on the mapping configuration, eliminating the need for developers to write SQL for standard database operations like INSERT, SELECT, UPDATE, and DELETE.

5. Database Independence: Hibernate provides database independence by abstracting the underlying database operations. This means that Hibernate can work with any relational database like MySQL, Oracle, PostgreSQL, etc., without changing the codebase. The underlying SQL generated by Hibernate is adapted based on the database being used.

6. Cache Management: Hibernate supports caching mechanisms, improving performance by reducing the number of database queries needed. It supports both first-level cache (session cache) and second-level cache (across sessions).

7. Transaction Management: Hibernate integrates with Java's transaction management APIs, ensuring that database operations are done in a reliable and consistent manner (ACID compliance).

8. Lazy Loading: Hibernate allows lazy loading, which means related data can be loaded only when it's needed. This helps optimize application performance by preventing unnecessary database queries.

9. Support for Complex Queries: Hibernate supports powerful querying capabilities using HQL (Hibernate Query Language), Criteria API, and native SQL. These provide advanced querying options beyond simple CRUD operations, such as aggregation, joining tables, and subqueries.

10. Reduction in Boilerplate Code: Hibernate reduces the need for writing repetitive boilerplate code, such as opening and closing database connections, handling transactions, and writing SQL manually.

- Comparison between Hibernate and JDBC.

ANS - Hibernate and JDBC are both popular technologies for interacting with relational databases in Java, but they differ in many aspects. Below is a comparison between Hibernate and JDBC:

1. Level of Abstraction:

Hibernate: Provides a higher level of abstraction over JDBC. It is an Object-Relational Mapping (ORM) framework that automates most of the database interactions. Hibernate allows you to map Java objects to database tables, making it easier to interact with the database through objects instead of raw SQL queries.

JDBC: JDBC (Java Database Connectivity) is a low-level API that allows direct interaction with relational databases. It requires developers to write SQL queries manually for every operation (CRUD) and manage the connection to the database.

2. Ease of Use:

Hibernate: Easier to use for developers who are familiar with object-oriented programming. Hibernate automates many repetitive tasks, such as managing connections, handling transactions, and converting results into objects. This reduces boilerplate code and simplifies database interactions.

JDBC: Requires more boilerplate code. Developers need to write SQL queries, handle connections, manage result sets, and explicitly handle exceptions. It's more manual and time-consuming.

3. Performance:

Hibernate: While Hibernate provides a higher-level abstraction, it can be slower than JDBC due to the overhead introduced by the ORM layer. Hibernate optimizes performance by using techniques like caching and lazy loading, but in some cases, it may not be as fast as direct JDBC calls.

JDBC: Typically offers better raw performance because there is no ORM layer, and queries are executed directly. However, performance depends on how efficiently the queries are written and managed.

4. Portability:

Hibernate: More portable. Hibernate allows you to switch between different databases without changing much of your application code. The ORM framework handles the specific SQL dialects of different databases.

JDBC: Less portable. With JDBC, you write database-specific SQL queries, and switching databases may require significant changes to the SQL code, particularly when dealing with differences in SQL dialects.

5. Transactions Management:

Hibernate: Has built-in transaction management. Hibernate handles transactions automatically and can manage them at the session level. It integrates well with Java's transaction management (JTA) for managing distributed transactions.

JDBC: Requires explicit transaction management. Developers need to manually handle transactions using commit, rollback, and setAutoCommit methods.

6. Query Language:

Hibernate: Uses HQL (Hibernate Query Language) or JPQL (Java Persistence Query Language), which are object-oriented query languages. These queries are database-independent and are converted into SQL by Hibernate.

JDBC: Uses SQL, which is database-specific and requires you to write queries manually.

7. Caching:

Hibernate: Supports built-in caching mechanisms, including first-level cache (session cache) and second-level cache (across sessions). This can improve performance by reducing database access.

JDBC: No built-in caching. You need to implement your own caching mechanism or use third-party libraries.

8. Learning Curve:

Hibernate: Steeper learning curve because you need to understand ORM concepts, entity mappings, and configuration. However, once learned, it can be much easier to use than JDBC.

JDBC: Simpler to get started with, especially for developers familiar with SQL. However, it requires more effort to write and manage database operations.

9. Database Independence:

Hibernate: More database-agnostic. It abstracts away database-specific details and allows you to work with Java objects, which can be mapped to any relational database.

JDBC: Database-dependent. JDBC code is tied to the specific database being used, and you need to write SQL queries specific to that database.

10. Error Handling:

Hibernate: Hibernate provides its own exception hierarchy, which is mapped to specific database exceptions. This simplifies error handling in complex applications.

JDBC: Directly uses SQLExceptions and requires more boilerplate code for handling errors.

11. Support for Complex Relationships:

Hibernate: Handles complex relationships between entities (one-to-many, many-to-many, etc.) effortlessly through annotations or XML configuration. It supports lazy and eager fetching strategies for relationships.

JDBC: Handling complex relationships in JDBC requires manually writing joins, and managing entity relationships is cumbersome.

12. Configuration:

Hibernate: Requires configuration files (hibernate.cfg.xml) or annotations to define mappings between Java classes and database tables. It may require some initial setup.

JDBC: Typically requires a database connection string, username, and password, but doesn't require as much configuration compared to Hibernate.

13. Development Speed:

Hibernate: Generally speeds up development because much of the boilerplate code is abstracted away. ORM eliminates the need for repetitive SQL statements and result set handling.

JDBC: Slower development due to more repetitive code and manual handling of database interactions.

14. Support for NoSQL Databases:

Hibernate: Primarily focused on relational databases, though there is support for NoSQL databases through frameworks like Hibernate OGM (Object/Grid Mapper).

JDBC: Also designed for relational databases, and support for NoSQL databases is not inherent in JDBC.

When to Use Hibernate:

When working with Java objects and needing automatic mapping to relational databases.

When you need portability across different database systems.

When you want to reduce boilerplate code and manage relationships and transactions more easily.

For applications where ease of use and development speed are priorities.

When to Use JDBC:

When you need fine-grained control over SQL queries.

For applications with strict performance requirements where you want to minimize overhead.

For simple applications or projects where the added complexity of Hibernate might not be necessary.

When working with a database that doesn't require complex object mappings or relationships.

NOTES:

Hibernate is a higher-level abstraction ideal for object-oriented development and enterprise applications with complex data models. It simplifies database interactions but may introduce some overhead.

JDBC is lower-level, offering more direct and efficient control over database operations, but requires more code and manual handling. It's suitable for smaller applications or when direct performance control is needed.

Both Hibernate and JDBC have their strengths and weaknesses, and the choice between them depends on the complexity of the project, performance requirements, and the developer's familiarity with the tools.

- Why use Hibernate? (Advantages: Database independence, automatic table creation,

HQL, etc.)

ANS - Hibernate is a popular Java framework used for object-relational mapping (ORM), which simplifies the interaction between Java applications and databases. It provides several key advantages:

1. Database Independence

Hibernate abstracts the underlying database operations, which makes it possible to switch between different databases (e.g., MySQL, PostgreSQL, Oracle) with minimal changes to the code.

It automatically handles the differences in SQL dialects between different databases, making applications database-independent.

2. Automatic Table Creation

Hibernate can automatically generate database tables based on the structure of Java classes (using annotations or XML configuration). This reduces the need for manual SQL code for table creation and schema management.

It supports automatic database schema updates, so changes to Java objects are reflected in the database schema without much intervention.

3. Hibernate Query Language (HQL)

HQL is an object-oriented query language similar to SQL but operates on Java objects rather than database tables. It allows developers to write database queries using Java class names and object properties instead of SQL table and column names.

HQL is database-agnostic and more intuitive for Java developers because it works directly with the object model, not the relational model.

4. Automatic Object-Relational Mapping (ORM)

Hibernate automatically handles the conversion between Java objects and database records. Developers can work directly with Java objects without worrying about SQL or database schemas.

It simplifies the management of relationships like one-to-many, many-to-one, and many-to-many associations.

5. Caching Support

Hibernate provides a built-in caching mechanism that improves performance by reducing the number of database queries. It supports first-level cache (session cache) and second-level cache (shared across sessions), which helps in speeding up data retrieval.

By caching frequently accessed data, it can significantly reduce the load on the database.

6. Lazy Loading

Hibernate supports lazy loading, where associated objects or collections are loaded only when they are accessed, reducing memory usage and improving performance.

This is especially useful for managing large data sets where only a subset of data is needed at any time.

7. Transaction Management

Hibernate integrates with Java's transaction management APIs, allowing for smooth management of database transactions. It ensures that operations are atomic, consistent, isolated, and durable (ACID properties).

It also simplifies rollback and commit processes for database operations, ensuring consistency.

8. Reduced Boilerplate Code

Hibernate eliminates the need for writing repetitive JDBC code, such as managing database connections, result sets, and SQL queries, thus reducing the amount of boilerplate code and making development more efficient.

9. Extensibility

Hibernate is highly customizable and extensible. It supports the use of custom data types, user-defined queries, and various configurations, allowing developers to fine-tune the ORM framework to fit their needs.

It can be integrated with other frameworks like Spring to provide additional features like dependency injection, aspect-oriented programming, and more.

10. Comprehensive Support for Associations

Hibernate supports different types of associations, including one-to-one, one-to-many, and many-to-many, along with cascading operations and automatic management of associated objects, making it easier to work with complex relational data structures.

Q-2 Hibernate Architecture:

- Explanation of the Hibernate architecture components:

1. SessionFactory: Configuration of Hibernate and creation of sessions.:

ANS - In Hibernate, the SessionFactory is a critical component that provides a way to configure and create Session objects. The SessionFactory is a thread-safe, heavyweight object, and it is typically created once during the application lifecycle. The Session is a lightweight, non-thread-safe object used to interact with the database.

To configure Hibernate and create sessions, you follow several key steps:

1. Add Hibernate Dependencies

For a Maven-based project, you need to add Hibernate dependencies to your pom.xml file

EX. <dependencies>

```
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.10.Final</version> <!-- or the latest version -->
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
```

```
        <version>5.6.10.Final</version>
    </dependency>
    <!-- Add JDBC driver, e.g., for MySQL -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.29</version>
    </dependency>
</dependencies>
```

2. Create Hibernate Configuration File (hibernate.cfg.xml)

You need to configure Hibernate settings in an XML file, usually named hibernate.cfg.xml. This file specifies Hibernate properties like the database connection, dialect, and mapping classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <!-- JDBC Database connection settings -->
    <session-factory>
        <!-- JDBC Database connection settings -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/your_database</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">password</property>

        <!-- JDBC connection pool settings -->
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
```

```
        <property name="hibernate.c3p0.max_statements">50</property>

        <!-- Specify dialect -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="hibernate.current_session_context_class">thread</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="hibernate.show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hibernate.hbm2ddl.auto">update</property>

        <!-- Mention annotated class -->
        <mapping class="com.example.model.YourEntity"/>
    </session-factory>
</hibernate-configuration>
```

3. Create a Hibernate Utility Class to Manage SessionFactory

EX. package com.example.util;

```
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
   private static final SessionFactory sessionFactory;

   static {
      try {
         // Create SessionFactory
         sessionFactory = new
Configuration().configure().addAnnotatedClass(YourEntity.class).buildSessionFactory();
      } catch (HibernateException e) {
         throw new ExceptionInInitializerError("SessionFactory creation failed " + e);
      }
   }

   public static Session getSession() {
      return sessionFactory.openSession();
   }

   public static void shutdown() {
      // Close caches and connection pools
      sessionFactory.close();
   }
```

}

4. Create a Session and Perform Database Operations

You can now create a Session from the SessionFactory, perform CRUD operations, and manage the transaction.

EX. package com.example;

```
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.example.model.YourEntity;
import com.example.util.HibernateUtil;

public class MainApp {
    public static void main(String[] args) {
        Session session = null;
        Transaction transaction = null;

        try {
            // Obtain session from the session factory
            session = HibernateUtil.getSession();

            // Start a transaction
            transaction = session.beginTransaction();

            // Create a new entity
            YourEntity entity = new YourEntity();
            entity.setName("Sample Entity");

            // Save the entity
```

```
            session.save(entity);


            // Commit the transaction
            transaction.commit();
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback(); // Rollback in case of error
            }
            e.printStackTrace();
        } finally {
            if (session != null) {
                session.close(); // Close the session
            }
        }
    }
}
```

5. Entity Class (YourEntity.java)

Make sure your entity class is properly annotated to map it to a database table.

EX. package com.example.model;

```
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class YourEntity {

    @Id
    private int id;

    private String name;

    // Getters and Setters
}
```

2. Session: The main interface between the Java application and the database.

ANS - In Hibernate, the Session is the main interface used to interact with the database. It provides methods for performing CRUD (Create, Read, Update, Delete) operations and manages the state of objects. The Hibernate Session acts as a bridge between the application and the underlying database, ensuring that changes to the objects in the application are synchronized with the database.

Hibernate Session:

Database Interaction: The Session is used to perform operations such as saving, updating, and deleting entities in the database.

Persistence Context: The Session manages a persistence context, which is essentially a cache of the objects that are loaded or saved during a transaction. It ensures that the same object instance is used during the session, preventing inconsistencies.

Transactions: A Session is typically tied to a transaction. Operations such as saving or updating objects can be performed within a transactional context to ensure data integrity.

Query Execution: The Session provides methods to execute HQL (Hibernate Query Language) or SQL queries to retrieve data from the database.

Lifecycle Management: The Session manages the lifecycle of entities, including their state transitions (e.g., from transient to persistent and vice versa).

Key Methods of a Hibernate Session:

save(): Saves a transient object to the database.

update(): Updates an existing persistent object in the database.

saveOrUpdate(): Either saves a new object or updates an existing one depending on its state.

delete(): Deletes an object from the database.

get() and load(): Retrieves objects from the database by their identifier.

createQuery(): Creates and executes HQL or SQL queries.

beginTransaction(): Starts a transaction.

commit(): Commits the current transaction.

rollback(): Rolls back the transaction if there is an error.

3. Transaction: Handling database transactions in Hibernate.

ANS - A transaction typically represents a single unit of work, and it ensures that operations such as inserting, updating, or deleting data are executed properly, and either fully committed or rolled back in case of errors.

Obtain a Session: Hibernate works with the concept of a Session that serves as a single-threaded unit of work. Each Session is associated with a transaction, and all operations (save, update, delete) must occur within a transaction.

Start a Transaction: Before performing any database operation, you must begin a transaction.

Perform Database Operations: Once the transaction is started, you can perform various CRUD (Create, Read, Update, Delete) operations.

Commit the Transaction: If all operations are successful and you want to save the changes permanently, you commit the transaction.

Rollback the Transaction (if needed): If an error occurs, you can roll back the transaction to undo any changes made during the transaction.

Close the Session: Finally, close the Session to release resources.

Example Code for Transaction Management in Hibernate:

Here is an example of how to handle a basic transaction in Hibernate using the Session and Transaction objects:

```
import org.hibernate.Session;

import org.hibernate.Transaction;

import org.hibernate.HibernateException;

import org.hibernate.cfg.Configuration;


public class TransactionExample {


   public static void main(String[] args) {

      // Create a session factory

      Session session = null;

      Transaction transaction = null;


      try {

         // Obtain session from session factory
```

```
            session = new Configuration().configure().buildSessionFactory().openSession();

            // Begin transaction
            transaction = session.beginTransaction();

            // Perform database operations (example: saving an entity)
            MyEntity entity = new MyEntity();
            entity.setName("John Doe");
            session.save(entity);

            // Commit the transaction
            transaction.commit();
            System.out.println("Transaction committed successfully.");

        } catch (HibernateException e) {
            // Handle exceptions and rollback if necessary
            if (transaction != null) {
                transaction.rollback(); // Undo any changes
            }
            System.err.println("Transaction failed: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Close the session
            if (session != null) {
                session.close();
            }
        }
    }
}
```

4) Query: Writing HQL (Hibernate Query Language) queries to interact with the database.

ANS - Hibernate Query Language (HQL) is a powerful object-oriented query language used to interact with databases in a Java-based application, leveraging the Hibernate framework. HQL is similar to SQL but operates on the object model instead of database tables. Here are some examples of writing HQL queries to interact with the database.

1. Select All Records from an Entity

To retrieve all records from a table corresponding to an entity:

FROM Employee

2. Select Specific Columns (Properties)

You can query specific properties of an entity by using a SELECT clause:

SELECT e.name, e.salary FROM Employee e

3. Filtering Data Using WHERE Clause

To filter data based on conditions, use the WHERE clause:

FROM Employee e WHERE e.salary > 50000

4. Using Logical Operators (AND, OR)

You can combine multiple conditions with AND, OR:

FROM Employee e WHERE e.salary > 50000 AND e.department = 'IT'

5. Ordering Results

You can order the result set using the ORDER BY clause:

FROM Employee e ORDER BY e.salary DESC

6. Pagination (LIMIT and OFFSET)

To paginate results, you can use setFirstResult() and setMaxResults() methods in your Query object, although HQL itself doesn't directly support LIMIT or OFFSET keywords like in SQL. Example:

Query query = session.createQuery("FROM Employee e ORDER BY e.salary");

query.setFirstResult(0); // First result (offset)

query.setMaxResults(10); // Max number of results

List results = query.list();

7. JOINs in HQL

HQL supports joins to fetch related entities. Here’s how you can use INNER JOIN:

FROM Employee e INNER JOIN e.department d WHERE d.name = 'IT'

8. Using GROUP BY Clause

To group results based on an attribute and perform aggregate functions:

SELECT e.department, COUNT(e) FROM Employee e GROUP BY e.department

9. Using Aggregate Functions

HQL supports aggregate functions like COUNT, SUM, AVG, MAX, MIN:

SELECT AVG(e.salary) FROM Employee e WHERE e.department = 'HR'

10. UPDATE Query

To update records, use the UPDATE keyword in HQL:

UPDATE Employee e SET e.salary = 60000 WHERE e.id = 1

11. DELETE Query

To delete a record, use the DELETE keyword:

DELETE FROM Employee e WHERE e.id = 1

12. Using Named Parameters

To avoid SQL injection and improve code readability, use named parameters in HQL queries:

FROM Employee e WHERE e.salary > :salary

13. Using IN Clause

You can use the IN operator to filter based on multiple values:

FROM Employee e WHERE e.department IN ('IT', 'HR', 'Finance')

14. Subqueries in HQL

You can use subqueries within HQL to filter results based on another query:

FROM Employee e WHERE e.salary > (SELECT AVG(e.salary) FROM Employee e)

5) Criteria: Criteria API for building dynamic queries.

The Criteria API in Java is a powerful tool used for building dynamic and type-safe queries in a JPA (Java Persistence API) environment. It allows developers to construct complex database queries programmatically without using raw SQL or JPQL (Java Persistence Query Language).

Here are key criteria for building dynamic queries using the Criteria API:

1. Type Safety

The Criteria API allows you to build queries in a type-safe manner. It generates the query based on the structure of your entity classes, ensuring that there are no type mismatches.

This is especially useful as it catches errors at compile time rather than runtime.

2. Dynamic Query Construction

You can build queries dynamically at runtime, which is ideal when the query parameters or conditions are not fixed.

For example, the conditions (like filters or sorting) might depend on user input or application logic.

3. Entities and Paths

In Criteria API, you work with root entities (which represent database tables) and paths (representing fields of these entities).

CriteriaBuilder is used to create various expressions for constructing the query.

Example:

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> cq = cb.createQuery(Person.class);

Root<Person> root = cq.from(Person.class);
```

4. Filtering with Predicates

Predicates represent the conditions of a query (e.g., WHERE clause).

They can be combined using logical operators like and, or, etc.

Example:

```
Predicate namePredicate = cb.equal(root.get("name"), "John");

Predicate agePredicate = cb.greaterThan(root.get("age"), 25);

Predicate finalPredicate = cb.and(namePredicate, agePredicate);

cq.where(finalPredicate);
```

5. Ordering and Sorting

Criteria API allows you to add ordering to your query (equivalent to ORDER BY in SQL).

You can specify ascending or descending order based on one or more fields.

Example:

```
cq.orderBy(cb.asc(root.get("name")));
```

6. Selecting Specific Fields

You can choose to select specific fields (columns) rather than entire entities. This is useful for optimizing performance when you need only a subset of fields.

Example:

```
cq.select(root.get("name"));
```

7. Join Operations

The Criteria API supports inner and left joins, similar to SQL joins, to link related entities (e.g., a Person might be joined with an Address).

Example:

```
Join<Person, Address> addressJoin = root.join("address", JoinType.LEFT);
```

8. Subqueries

The Criteria API supports subqueries, allowing you to create complex queries where one query's result is used in another query.

Example:

```
Subquery<Long> subquery = cq.subquery(Long.class);

Root<Order> subRoot = subquery.from(Order.class);

subquery.select(cb.count(subRoot));

subquery.where(cb.equal(subRoot.get("person"), root));
```

9. Aggregation and Grouping

You can also perform aggregation (such as counting, summing, averaging) and grouping using the Criteria API.

Example:

```
cq.groupBy(root.get("age"));

cq.select(cb.count(root));
```

10. Pagination

The Criteria API allows you to implement pagination by setting the first result and the maximum results.

Example:

```
TypedQuery<Person> query = entityManager.createQuery(cq);
query.setFirstResult(0); // starting index
query.setMaxResults(10); // number of results
List<Person> results = query.getResultList();
```

11. Criteria API in Practice: A Full Example

Here's an example of a dynamic query that filters by name, age, and orders the result by name:

```
public List<Person> findPersons(String nameFilter, Integer minAge, String orderBy) {
   CriteriaBuilder cb = entityManager.getCriteriaBuilder();
   CriteriaQuery<Person> cq = cb.createQuery(Person.class);
   Root<Person> root = cq.from(Person.class);

   List<Predicate> predicates = new ArrayList<>();

   if (nameFilter != null) {
      predicates.add(cb.like(root.get("name"), "%" + nameFilter + "%"));
   }
   if (minAge != null) {
      predicates.add(cb.greaterThanOrEqualTo(root.get("age"), minAge));
   }
   cq.where(cb.and(predicates.toArray(new Predicate[0])));

   if (orderBy != null) {
      cq.orderBy(cb.asc(root.get(orderBy)));
   }

   TypedQuery<Person> query = entityManager.createQuery(cq);
```

```
    return query.getResultList();
}
```

12. Error Handling

While using the Criteria API, it's important to handle errors like IllegalArgumentException and PersistenceException gracefully, especially when dealing with complex queries or invalid parameters.

13. Performance Considerations

Although the Criteria API is powerful and flexible, it can sometimes result in less optimized queries compared to hand-written JPQL or SQL.

Developers should analyze generated queries to ensure performance is not impacted, especially for complex queries involving joins, subqueries, or large datasets.

14. Integration with Other JPA Features

Criteria API integrates well with other JPA features such as transactions, entity managers, and named queries.

It can be combined with the EntityManager to execute the dynamically created query and retrieve results.

- How Hibernate works internally from loading configuration files to executing queries.

ANS - Hibernate is an Object-Relational Mapping (ORM) framework for Java that provides a way to map Java objects to database tables and vice versa. Internally, Hibernate performs a series of steps from loading configuration files to executing queries.

1. Loading Configuration Files

Hibernate relies on a configuration file (typically hibernate.cfg.xml) or programmatic configuration to set up the database connection and other parameters.

Configuration Object: Hibernate starts by creating a Configuration object, which is responsible for reading the hibernate.cfg.xml file or loading properties for database connection, dialect, caching, and mapping files (such as .hbm.xml or annotations).

SessionFactory: After loading the configuration, Hibernate builds a SessionFactory object. This is a heavy-weight object, which is created once and reused throughout the application. The SessionFactory is responsible for managing connections, providing Session objects, and controlling cache behavior.

EX. :

```
Configuration configuration = new Configuration().configure();

SessionFactory sessionFactory = configuration.buildSessionFactory();
```

2. Session Creation

Session Object: Hibernate's Session interface is the primary point of interaction between the application and Hibernate ORM. A Session is created using the SessionFactory object.

A Session represents a single-threaded unit of work with the database. It is used to create, read, update, and delete operations (CRUD operations) on persistent entities.

EX.:

```
Session session = sessionFactory.openSession();
```

3. Transaction Management

Hibernate uses a Transaction object to manage transactions. A Session is associated with a transaction, and the transaction manages the commit and rollback of operations to ensure data consistency.

Typically, a transaction is started at the beginning of the work and committed at the end, or rolled back in case of an error.

EX.:

```
Transaction transaction = session.beginTransaction();

// Perform operations

transaction.commit(); // or transaction.rollback();
```

4. Entity Mapping

Hibernate maps Java objects (POJOs) to database tables. These mappings can be defined through annotations or XML files.

For example, if you have a User class, you can map it to the users table in the database.

The Session interacts with these mapped entities, performing operations like saving or updating the objects.

EX.:

```
User user = new User("John", "Doe");

session.save(user);  // Persists the object in the database
```

5. Query Execution

Hibernate supports multiple ways of querying the database:

a. HQL (Hibernate Query Language)

HQL is an object-oriented query language that is similar to SQL but operates on the entity objects instead of database tables.

The Session object provides methods like createQuery() to execute HQL queries.

EX.:

```
Query query = session.createQuery("FROM User WHERE lastName = :lastName");

query.setParameter("lastName", "Doe");

List<User> users = query.list();
```

b. Criteria API

The Criteria API is a more object-oriented way to create queries. It allows you to build queries dynamically using Java objects instead of writing HQL or SQL.

This is especially useful when the query needs to be constructed dynamically based on user input.

EX.:

```
Criteria criteria = session.createCriteria(User.class);

criteria.add(Restrictions.eq("lastName", "Doe"));

List<User> users = criteria.list();
```

c. Native SQL

If needed, you can also execute native SQL queries directly through Hibernate.

EX.:

Query query = session.createSQLQuery("SELECT * FROM users WHERE last_name = :lastName");

query.setParameter("lastName", "Doe");

List<Object[]> results = query.list();

6. Loading and Caching

Hibernate supports lazy loading, meaning that related entities are loaded only when they are accessed for the first time, which improves performance.

7. Flush and Synchronization

Session Flush: The flush() method is responsible for synchronizing the in-memory objects with the database.

EX.:

session.flush(); // Forces Hibernate to synchronize the session with the database

8. Transaction Commit

After all operations are performed, the transaction is committed. Hibernate then generates the necessary SQL statements (insert, update, delete) to be executed on the database.

EX.:

transaction.commit(); // Commits the transaction

9. Closing Session

After the work is done, the Session object should be closed to release database connections and resources.

EX.:

session.close();

2. Hibernate Relationships (One-to-One, One-to-Many, Many-to-One, Many-to-Many)

- Object Relationships in Hibernate:

Q-1 How Hibernate manages relationships between Java objects and database tables.

ANS - Hibernate is an Object-Relational Mapping (ORM) framework that helps Java applications interact with relational databases. It manages the mapping between Java objects (also known as entities) and database tables in a way that abstracts the complexities of direct SQL queries. Here's how Hibernate handles relationships between Java objects and database tables:

1. Basic Object-Relational Mapping

Entities: In Hibernate, a Java class is mapped to a database table. Each instance of the class corresponds to a row in the table, and each field in the class corresponds to a column in the table.

Annotations or XML: Hibernate allows you to map Java classes to database tables using either annotations or XML configuration files.

2. Managing Relationships

Hibernate supports the four common types of relationships between entities in a database:

One-to-One

One-to-Many

Many-to-One

Many-to-Many

Q-2 Overview of the different types of relationships:

1. One-to-One Relationship:

A single instance of an entity is related to a single instance of another entity.

ANS - A One-to-One relationship means that each row in the first table is linked to exactly one row in the second table, and vice versa.

Example: A Person entity and an Address entity, where each person has exactly one address.

Mapping:

You can use @OneToOne annotation to map this relationship in your Java classes.

You can specify the owning side with @JoinColumn to define the foreign key.

Example:@Entity

```
public class Person {
   @Id
   private Long id;
   private String name;

   @OneToOne
   @JoinColumn(name = "address_id")
   private Address address;
}

@Entity
public class Address {
   @Id
   private Long id;
   private String street;
   private String city;
}
```

2. One-to-Many Relationship: One entity can have multiple related entities.

ANS - A One-to-Many relationship means that one row in a table can be associated with multiple rows in another table, while a Many-to-One relationship is the inverse (i.e., multiple rows in the second table can relate to one row in the first table).

Example: A Department entity and Employee entity, where each department has multiple employees, but each employee works for only one department.

Mapping:

Use @OneToMany on the one-side (Department) and @ManyToOne on the many-side (Employee).

Foreign key is typically placed in the "many" table (Employee).

```
@Entity
public class Department {
   @Id
   private Long id;
   private String name;

   @OneToMany(mappedBy = "department")
   private List<Employee> employees;
}

@Entity
public class Employee {
   @Id
   private Long id;
   private String name;

   @ManyToOne
   @JoinColumn(name = "department_id")
   private Department department;
}
```

3. Many-to-One Relationship: Many entities are associated with a single entity.

ANS - SAME AS ABOVE ONE TO MANY

4. Many-to-Many Relationship

A Many-to-Many relationship means that multiple rows in one table can be associated with multiple rows in another table.

Example: A Student entity and a Course entity, where each student can enroll in multiple courses, and each course can have multiple students.

Mapping:

Use @ManyToMany on both sides of the relationship.

Hibernate automatically creates a join table to manage this relationship.

Example:

```
@Entity
public class Student {
   @Id
   private Long id;
   private String name;

   @ManyToMany
   @JoinTable(
      name = "student_course",
      joinColumns = @JoinColumn(name = "student_id"),
      inverseJoinColumns = @JoinColumn(name = "course_id")
   )
   private List<Course> courses;
}
```

```
@Entity
public class Course {
    @Id
    private Long id;
    private String name;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

2. Multiple instances of an entity are associated with multiple instances of another entity.

ANS - When multiple instances of one entity are associated with multiple instances of another entity, this is typically referred to as a many-to-many relationship.

In a many-to-many relationship, each instance of the first entity can be related to many instances of the second entity, and vice versa.

Example:

Consider two entities: Students and Courses.

A student can enroll in multiple courses.

A course can have multiple students enrolled in it.

This creates a many-to-many relationship between Students and Courses.

Q-3 Mapping Relationships in Hibernate:

1) How to map relationships in Hibernate using annotations like @OneToOne,

@OneToMany, @ManyToOne, and @ManyToMany.

ANS - These annotations are part of the Java Persistence API (JPA) and are used to establish the associations between entities.

1. @OneToOne (One-to-One Relationship)

A @OneToOne relationship is used when one entity is associated with exactly one instance of another entity.

Example:

Let's assume we have a Person and a Passport entity, where each Person has one Passport.

```
import javax.persistence.*;


@Entity
public class Person {


   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;


   private String name;


   @OneToOne(mappedBy = "person")
   private Passport passport;


   // getters and setters
}


@Entity
public class Passport {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String passportNumber;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

    // getters and setters
}
```

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

A @OneToMany relationship is used when one entity is associated with many instances of another entity.

Example:

Let’s assume we have an Author entity and a Book entity, where one Author can write many Books.

```
import javax.persistence.*;
import java.util.List;

@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    private String name;

    @OneToMany(mappedBy = "author")
    private List<Book> books;

    // getters and setters
}

@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToOne
    @JoinColumn(name = "author_id")
    private Author author;

    // getters and setters
}
```

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

A @ManyToOne relationship is the inverse of the @OneToMany relationship, and it is used when many instances of one entity are associated with one instance of another entity.

Example:

In the example above, the Book entity already demonstrates this relationship using @ManyToOne.

4. @ManyToMany (Many-to-Many Relationship)

A @ManyToMany relationship is used when many instances of one entity are associated with many instances of another entity.

Example:

Let's assume we have a Student entity and a Course entity, where a Student can enroll in many Courses and a Course can have many Students.

```
import javax.persistence.*;

import java.util.Set;


@Entity

public class Student {


  @Id

  @GeneratedValue(strategy = GenerationType.IDENTITY)

  private Long id;


  private String name;


  @ManyToMany

  @JoinTable(

    name = "student_course",

    joinColumns = @JoinColumn(name = "student_id"),

    inverseJoinColumns = @JoinColumn(name = "course_id")

  )
```

```
    private Set<Course> courses;

    // getters and setters
}

@Entity
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String courseName;

    @ManyToMany(mappedBy = "courses")
    private Set<Student> students;

    // getters and setters
}
```

2. The concept of owning and inverse sides in relationships.

ANS - The concept of "owning" and "inverse sides" in relationships can be interpreted in various ways, depending on the context—whether it pertains to emotional dynamics, power structures, or philosophical viewpoints about identity and connection. Below are two possible ways to interpret these ideas:

1. Owning and Power Dynamics in Relationships

In this context, "owning" refers to control or possessiveness in relationships. This could relate to one person feeling as if they "own" or control the other person, dictating their actions, decisions, or life choices. The "inverse side" in this situation would be the reaction or counter-response to this control, which could involve feelings of resentment, rebellion, or a desire for freedom.

For example, in romantic relationships, one person might dominate or control the other's behavior, decisions, or social interactions.

This dynamic can lead to an unhealthy relationship if one person consistently tries to "own" the other, leading to an imbalance of power.

2. Owning and Emotional Responsibility

"Owning" in relationships can also refer to taking responsibility for one's emotions, actions, and decisions. In this interpretation, each person in a relationship "owns" their feelings and how they react to situations. The inverse side, then, might refer to projection or blame—where one partner avoids ownership of their emotional experience and instead places responsibility on the other person.

For example, in conflicts, one partner might take full responsibility for their actions and emotions ("I feel hurt because of what I did"), while the other person might deflect blame, saying things like "You made me feel this way." The inverse side of owning would be the tendency to externalize and shift responsibility rather than confronting one’s own role in the situation.

3. Philosophical or Existential Perspective

From a more philosophical or existential viewpoint, "owning" could be seen as a form of personal identity and self-determination in relationships. Here, ownership is not about possession but about having agency and control over one’s identity, desires, and boundaries within a relationship.

For instance, in any meaningful connection, individuals should be able to "own" who they are and their choices without being subsumed by the other person's identity or desires. The inverse would be a relationship where one person loses themselves in the other, neglecting their own needs, beliefs, or identity in favor of pleasing or accommodating the other.

4. Mutual Ownership and Balance

In healthy relationships, both individuals might "own" their own roles and responsibilities while respecting each other's autonomy. The inverse side in such relationships would be the tension or conflict that arises when one person tries to dominate or control the other.

3. Cascade types and how they affect related entities.

ANS - Cascade types in the context of databases and object-relational mapping (ORM) frameworks, such as Hibernate in Java or Entity Framework in .NET, define the rules for how operations on an entity (like save, delete, update, etc.) should propagate to related entities.

1. CascadeType.PERSIST

Effect: When the parent entity is saved (persisted), all associated child entities are also persisted (saved).

Use Case: This is used when you want to automatically save the associated entities when you save the parent entity. It eliminates the need to explicitly save the related entities separately.

Example: In a Book entity that has a collection of Chapter entities, if you save a Book, the associated Chapters will also be saved automatically.

2. CascadeType.MERGE

Effect: When the parent entity is merged (updated), all associated child entities are also merged (updated).

Use Case: This cascade type is used when you want to propagate updates from the parent to the child. If you update the parent entity, the changes will be reflected in the associated child entities.

Example: If a User entity has an associated Profile, and the User entity is updated, this cascade type ensures that the changes to User will also be applied to Profile.

3. CascadeType.REMOVE

Effect: When the parent entity is removed (deleted), all associated child entities are also deleted.

Use Case: This is used to ensure that when the parent entity is deleted, the related child entities are also deleted to maintain referential integrity.

Example: In a Department entity with associated Employee entities, deleting the Department will also delete all associated Employees.

4. CascadeType.REFRESH

Effect: When the parent entity is refreshed (refreshed from the database), all associated child entities are also refreshed.

Use Case: This is used when you want to reload the child entities from the database when the parent entity is refreshed.

Example: If the Product entity is refreshed, any associated Review entities will also be reloaded from the database.

5. CascadeType.DETACH

Effect: When the parent entity is detached (removed from the persistence context), all associated child entities are also detached.

Use Case: This is used to remove the related child entities from the current persistence context when the parent is detached. It does not remove the child entities from the database but detaches them from the current session.

Example: If a Customer is detached, the associated Order entities will also be detached from the current session but remain in the database.

6. CascadeType.ALL

Effect: This is a shorthand to apply all the above cascade types: PERSIST, MERGE, REMOVE, REFRESH, and DETACH.

Use Case: It is useful when you want all operations (save, update, delete, etc.) on the parent entity to automatically cascade to the related child entities.

Example: If a BlogPost entity is saved, updated, or deleted, the same actions will be applied to all associated Comment entities.

3. Hibernate CRUD Example

Q-1 Understanding CRUD Operations in Hibernate:

ANS - Hibernate is an object-relational mapping (ORM) framework that simplifies the interaction between Java applications and relational databases. CRUD operations represent the four basic functions of persistent storage: Create, Read, Update, and Delete.

1. Create (Insert):

The Create operation inserts a new record into the database. It is performed using the save() or persist() method in Hibernate.

save(): This method generates a new identifier for the entity and inserts the object into the database. It returns the generated identifier.

persist(): This is used for the same purpose as save() but is generally more specific in terms of managing the entity's lifecycle. The difference is subtle, but persist() is part of the JPA (Java Persistence API) standard, whereas save() is Hibernate-specific.

Example:

```
Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();


Employee employee = new Employee();

employee.setName("John");

employee.setSalary(50000);


session.save(employee);  // Save the employee object to the database

transaction.commit();


session.close();
```

2. Read (Select):

The Read operation retrieves data from the database. In Hibernate, this is done using HQL (Hibernate Query Language), Criteria API, or directly using the get() and load() methods.

get(): This method fetches the entity by its identifier and returns the object. If the entity does not exist, it returns null.

load(): Similar to get(), but it throws an exception (ObjectNotFoundException) if the entity is not found.

HQL: Hibernate Query Language is used to write queries to retrieve data in a more flexible way, using object-oriented syntax.

Example using get():

```
Session session = sessionFactory.openSession();

Employee employee = (Employee) session.get(Employee.class, 1);  // Retrieve employee with id 1

session.close();
```

3. Update:

The Update operation modifies an existing record in the database. In Hibernate, this is done using the update(), merge(), or saveOrUpdate() methods.

update(): This method updates an existing entity in the database.

merge(): This is a more flexible method that synchronizes the state of the given object with the database. It can be used for both detached and persistent objects.

saveOrUpdate(): This method either inserts or updates an entity depending on whether the entity is already in the session context.

Example using update():

```
Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();


Employee employee = (Employee) session.get(Employee.class, 1);

employee.setSalary(55000);


session.update(employee);  // Update the employee object in the database

transaction.commit();


session.close();
```

4. Delete:

The Delete operation removes a record from the database. In Hibernate, this is performed using the delete() method.

Q-2 Writing HQL (Hibernate Query Language):

1) Basics of HQL and how it differs from SQL.

ANS - Hibernate Query Language (HQL) is an object-oriented query language used to perform database operations in Hibernate, a popular Java-based ORM framework. HQL is similar to SQL, but it operates on entities (Java objects) rather than database tables. Here are some basic examples and concepts of HQL:

1. Select Query

EX.:

from Employee e where e.department = 'Sales'

This query retrieves all Employee objects where the department is "Sales". Here, Employee is an entity class, and e is an alias used to reference the object.

2. Select with Specific Columns

EX.:

select e.name, e.salary from Employee e where e.department = 'Sales'

This query retrieves only the name and salary of employees in the "Sales" department.

3. Using join

EX.:

from Employee e join e.department d where d.name = 'HR'

This query performs an inner join between Employee and Department entities and returns employees who belong to the "HR" department.

4. Using like (Pattern Matching)

EX.:

from Employee e where e.name like 'J%'

This query retrieves all employees whose name starts with "J".

5. Using group by and having

EX.:

select e.department, count(e) from Employee e group by e.department having count(e) > 10

This query groups employees by their department and returns departments that have more than 10 employees.

6. Using order by

EX.:

from Employee e order by e.salary desc

This query retrieves all employees and orders them by salary in descending order.

7. Using IN

EX.:

from Employee e where e.department in ('Sales', 'Marketing')

This query retrieves employees from either the "Sales" or "Marketing" department.

8. Using distinct

EX.:

select distinct e.department from Employee e

This query retrieves all distinct departments.

9. Using fetch for Eager Loading

EX.:

from Employee e fetch join e.department

This query fetches employees along with their associated department in a single query (eager loading).

10. Using setParameter for Named Parameters

EX.:

Query query = session.createQuery("from Employee e where e.department = :dept");

query.setParameter("dept", "Sales");

List<Employee> results = query.list();

This query retrieves employees from the "Sales" department using a named parameter (:dept).

11. Using update and delete Queries

Update Query

EX.:

update Employee e set e.salary = e.salary + 1000 where e.department = 'HR'

This query increases the salary of employees in the "HR" department by 1000.

Delete Query

EX.:

delete from Employee e where e.department = 'Intern'

This query deletes employees who belong to the "Intern" department.

12. Using Native SQL

Sometimes, you may need to use raw SQL in Hibernate when certain operations cannot be expressed easily in HQL. For this, you can use createSQLQuery() to write native SQL.

EX.:

SQLQuery query = session.createSQLQuery("SELECT * FROM employee WHERE department = :dept");

query.setParameter("dept", "Sales");

List<Object[]> results = query.list();

2) How to perform CRUD operations using HQL.

ANS -

Create (Insert): Typically done using save() or persist(), but can also use INSERT INTO.

Read (Select): FROM for all records, WHERE for filtering, and SELECT for specific fields.

Update (Modify): UPDATE <entity> SET <field> = <value> WHERE <condition>.

Delete (Remove): DELETE FROM <entity> WHERE <condition>.

EX. Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

// 1. Create (Insert)

Employee emp = new Employee("John Doe", 50000);

session.save(emp);

// 2. Read (Select)

String hql = "FROM Employee e WHERE e.name = :name";

Query query = session.createQuery(hql);

query.setParameter("name", "John Doe");

List<Employee> employees = query.list();

```
// 3. Update (Modify)
String updateHql = "UPDATE Employee e SET e.salary = :newSalary WHERE e.id = :id";
Query updateQuery = session.createQuery(updateHql);
updateQuery.setParameter("newSalary", 55000);
updateQuery.setParameter("id", emp.getId());
updateQuery.executeUpdate();

// 4. Delete (Remove)
String deleteHql = "DELETE FROM Employee e WHERE e.id = :id";
Query deleteQuery = session.createQuery(deleteHql);
deleteQuery.setParameter("id", emp.getId());
deleteQuery.executeUpdate();

tx.commit();
session.close();
```

3) Introduction to the Criteria API for dynamic queries.

ANS - The Criteria API is a part of the Java Persistence API (JPA), designed to allow the creation of dynamic queries in a type-safe and object-oriented way. It provides an alternative to the traditional JPQL (Java Persistence Query Language) or SQL queries, enabling developers to construct queries programmatically rather than writing them as strings.

Example of a Basic Criteria Query:

Here's a simple example showing how to use the Criteria API to create a dynamic query.

```
import javax.persistence.*;
import javax.persistence.criteria.*;

public class CriteriaQueryExample {

    private EntityManager entityManager;
```

```
    public CriteriaQueryExample(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    public List<Employee> getEmployeesByDepartment(String departmentName) {
        // Step 1: Get the CriteriaBuilder
        CriteriaBuilder cb = entityManager.getCriteriaBuilder();

        // Step 2: Create the CriteriaQuery object
        CriteriaQuery<Employee> query = cb.createQuery(Employee.class);

        // Step 3: Define the root (main entity)
        Root<Employee> root = query.from(Employee.class);

        // Step 4: Define the predicate (condition)
        Predicate condition = cb.equal(root.get("department"), departmentName);

        // Step 5: Apply the condition to the query
        query.where(condition);

        // Step 6: Execute the query
        TypedQuery<Employee> typedQuery = entityManager.createQuery(query);
        return typedQuery.getResultList();
    }
}
```

1.CriteriaBuilder cb = entityManager.getCriteriaBuilder();: Obtain the CriteriaBuilder, which is used to create query elements.

2.CriteriaQuery<Employee> query = cb.createQuery(Employee.class);: Create a CriteriaQuery object to represent a query that will return Employee entities.

3.Root<Employee> root = query.from(Employee.class);: Define the root of the query, which corresponds to the Employee entity.

4.Predicate condition = cb.equal(root.get("department"), departmentName);: Define the query condition, in this case, filtering employees by their department.

5.query.where(condition);: Add the condition to the query.

6.TypedQuery<Employee> typedQuery = entityManager.createQuery(query);: Create a typed query to execute and fetch the result.