MODULE 6

Java – Hibernate Framework

==

1.Introduction to Hibernate Architecture

QUES: What is Hibernate?

Definition and purpose of Hibernate as an ORM (Object Relational Mapping) tool.
Hibernate is an ORM (Object Relational Mapping) framework for Java, allowing developers to map Java objects to database tables. It simplifies database interaction by handling the conversion between Java objects and relational database records.

Comparison between Hibernate and JDBC.
JDBC: A lower-level, manual approach to database interaction. Developers must write SQL queries and handle connections, transactions, and result sets manually.
Hibernate: Provides a higher-level abstraction, eliminating the need for manual SQL and reducing boilerplate code. It automatically handles database connections, transactions, and object-relational mapping.

Why use Hibernate? (Advantages: Database independence, automatic table creation, HQL, etc.)

Database Independence: Hibernate allows applications to be database-agnostic, supporting multiple database systems with minimal changes.

Automatic Table Creation: Hibernate can generate database tables automatically based on Java class definitions.

HQL (Hibernate Query Language): Provides an object-oriented query language that is database-independent.

Caching: Hibernate supports first and second-level caching, improving performance by reducing database access.

Lazy Loading: Efficiently loads related entities on-demand, improving resource management.

QUES: Hibernate Architecture

- **Explanation of the Hibernate architecture components:**

Hibernate’s architecture consists of several components that work together to provide seamless interaction between a Java application and a database. Here’s a breakdown of the key components:

SessionFactory: Configuration of Hibernate and creation of sessions.

Purpose: The SessionFactory is a factory class responsible for creating and managing Session objects. It’s the central point in Hibernate for configuring and initializing the ORM framework.
Configuration: The SessionFactory is created using the configuration files (hibernate.cfg.xml or through annotations) that define the database connection settings, dialects, and mapping details between Java objects and database tables.
Creation of Sessions: Once the configuration is loaded, the SessionFactory can create multiple Session objects, which are used to interact with the database.

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

Purpose: The Session is the primary interface between the Java application and the database. It is responsible for CRUD operations (Create, Read, Update, Delete) and other database-related activities.

Functions: It provides methods for saving, updating, deleting, and querying entities, and is used to begin and commit transactions. A Session is typically used within a single transaction.

Transaction: Handling database transactions in Hibernate.

Purpose: The Transaction interface is used to manage database transactions in Hibernate.

Functions: It allows developers to control the transaction boundaries. Transactions in Hibernate are used to ensure that database operations are performed atomically (all operations in a transaction succeed or fail together). It handles the commit or rollback of operations to the database.

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

Purpose: Hibernate provides Query objects that allow developers to execute queries in the Hibernate Query Language (HQL).

Functions: HQL is similar to SQL but is object-oriented, allowing developers to query entities rather than database tables. Queries can be dynamic, and the results are returned as Java objects.

Criteria: Criteria API for building dynamic queries.

Purpose: The Criteria API is a more programmatic, object-oriented way to create dynamic queries without writing HQL or SQL directly.

Functions: It allows developers to build complex queries using Java code, which makes it easier to create queries dynamically (such as adding conditions or joins) based on runtime values.

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

Configuration File Loading: Hibernate first loads its configuration file (hibernate.cfg.xml or annotations) to establish database connection details and mapping configurations.

SessionFactory Initialization: The SessionFactory is created once during the application startup. It reads the configuration file, loads the necessary metadata about classes and mappings, and builds the connection pool.

Creating a Session: A Session is created from the SessionFactory whenever database operations need to be performed. A session represents a single unit of work and interacts with the database.

Transaction Handling: When performing any CRUD operations or queries, Hibernate initiates a transaction via the Transaction interface. A transaction is started before performing operations, and it is either committed or rolled back based on the outcome.

Query Execution: Queries can be executed using HQL (via Query interface) or dynamically via the Criteria API. Hibernate translates these high-level queries into the appropriate SQL for the underlying database.

Result Handling: The query results are returned as Java objects, allowing the application to work with data in the object-oriented paradigm.

Session Closing: After the transaction completes, the Session is closed, and resources are released.

===

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

- **Object Relationships in Hibernate**

QUES: How Hibernate manages relationships between Java objects and database tables.

Hibernate maps Java object relationships to database tables through the use of annotations or XML configuration.

QUES: Overview of the different types of relationships:

One-to-One Relationship: A single instance of an entity is associated with one instance of another entity.
Example: A person has one passport.

One-to-Many Relationship: One entity can have multiple related entities.
Example: A department has multiple employees.

Many-to-One Relationship: Many entities are associated with a single entity.
Example: Multiple employees belong to one department.

Many-to-Many Relationship: Multiple instances of an entity are associated with multiple instances of another entity.
Example: Students can enroll in multiple courses, and courses can have multiple students.

Mapping Relationships in Hibernate

QUES: How to map relationships in Hibernate using annotations like @OneToOne, @OneToMany, @ManyToOne, and @ManyToMany.
Hibernate uses annotations to define relationships between entities. Here's how to map different types of relationships:

1. @OneToOne
A One-to-One relationship maps a single instance of an entity to a single instance of another entity.
Example: A Person has one Passport.
2. @OneToMany and @ManyToOne
One-to-Many: One entity is related to multiple entities.
Many-to-One: Many entities are related to a single entity.
Example: A Department has many Employees, and an Employee belongs to one Department.

3. @ManyToMany
A Many-to-Many relationship maps multiple instances of an entity to multiple instances of another entity.
Example: A Student can enroll in multiple Courses, and each Course can have multiple Students.

The concept of owning and inverse sides in relationships.

Owning Side: The entity that contains the foreign key and is responsible for managing the relationship (usually where the @JoinColumn annotation is placed).
Inverse Side: The entity that doesn't hold the foreign key and is not responsible for managing the relationship (usually where the mappedBy attribute is used).
Example:
In a @OneToMany relationship, the owning side is typically the entity that has the foreign key, while the inverse side does not.
In a @ManyToMany relationship, the owning side is the entity that defines the join table, while the inverse side uses mappedBy.

Cascade types and how they affect related entities.

Cascade operations allow the persistence operations (like persist, merge, remove, etc.) to propagate to the related entities.
Common cascade types:
CascadeType.PERSIST: When the parent entity is persisted, the child entity is also persisted.
CascadeType.MERGE: When the parent entity is merged, the child entity is also merged.
CascadeType.REMOVE: When the parent entity is removed, the child entity is also removed.
CascadeType.ALL: All operations (persist, merge, remove, refresh, detach) are cascaded.

==

Hibernate CRUD Example

QUES: Understanding CRUD Operations in Hibernate:

Hibernate is an object-relational mapping (ORM) framework for Java, allowing developers to interact with relational databases in an object-oriented way. CRUD stands for Create, Read, Update, and Delete, which are the four basic operations used to manage data in a database. Here's how Hibernate handles each operation:

Create (Insert): How to use Hibernate to insert records into a database.

In Hibernate, you can insert records into the database by saving an entity object using session.save() or session.persist().

Here, the session.save() method inserts the Employee object into the database.

Read (Select): Fetching data from the database using Hibernate.

To read or fetch data, Hibernate provides methods like session.get(), session.load(), and session.createQuery() for querying the database.

Update: Modifying existing records in the database.

Updating an entity in Hibernate can be done using the session.update() or session.merge() methods. It's important to note that Hibernate uses the identifier (primary key) to identify the record that needs to be updated.

Delete: Removing records from the database.

To delete records, Hibernate provides the session.delete() method.

QUES: Writing HQL (Hibernate Query Language)

Hibernate Query Language (HQL) is an object-oriented query language similar to SQL but designed to work with Hibernate's object-relational mapping (ORM). HQL is used to perform database operations by querying the underlying Java objects (entities) rather than directly dealing with database tables.

Basics of HQL and how it differs from SQL.

Object-Oriented: HQL operates on Java entities (classes) and their properties, not on database tables and columns.

- In SQL, you query tables and columns.
- In HQL, you query entities (Java classes) and their attributes (fields or properties).

Entity Names and Properties: In SQL, you deal with table names and column names, but in HQL, you deal with class names and property names.

- Example:
  - **SQL**: SELECT * FROM employee WHERE salary > 50000
  - **HQL**: FROM Employee WHERE salary > 50000

No Joins with Foreign Keys: HQL uses object navigation instead of foreign keys.

- Example:
  - **SQL**: SELECT * FROM Employee e INNER JOIN Department d ON e.dept_id = d.id
  - **HQL**: FROM Employee e JOIN e.department d

Portability: HQL is database-independent, meaning the query will work across different databases, as Hibernate takes care of the SQL translation.

How to perform CRUD operations using HQL.

Here’s how you can perform CRUD operations using HQL in Hibernate:

1. Create (Insert)

In HQL, we typically use session.save() or session.persist() for inserting records, rather than directly using an INSERT statement.

Note: HQL does not have an explicit INSERT statement. Instead, the object is saved using Hibernate’s save() or persist() methods.

2. Read (Select)

To fetch data using HQL, we can use FROM, WHERE, ORDER BY, and other clauses.

HQL Syntax: FROM EntityName WHERE condition

Selecting Specific Fields: You can use SELECT to retrieve specific fields (columns) instead of full entities.

3. Update

To update records, use HQL's UPDATE syntax.

4. Delete

To delete records, use the DELETE statement in HQL.

QUES: Introduction to the Criteria API for dynamic queries.

The Criteria API is a programmatic approach to building HQL queries dynamically. It’s often used for complex or dynamic queries where using string-based HQL might be cumbersome or error-prone. The Criteria API allows you to create queries without writing HQL directly.

Key Advantages of Criteria API:

Type-Safety: The Criteria API avoids hardcoded strings, reducing the chance of errors like invalid property names.

Dynamic Queries: It allows you to build queries dynamically based on user input or conditions.

Portability: Like HQL, it is database-agnostic.