HIBERNATE LAB ASSESMENT

Saikumar

1.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

	<session-factory>

		<!-- JDBC Database connection settings -->
		<property
name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
		<property name="connection.url">
			jdbc:mysql://localhost:3306/OneToOneMapping
	</property>
		<property name="connection.username">root</property>
		<property name="connection.password">root</property>

		<!-- JDBC connection pool settings ... using built-in test pool
-->
		<property name="connection.pool_size">1</property>

		<!-- Select our SQL dialect -->
		<property
name="dialect">org.hibernate.dialect.MySQLDialect</property>

		<!-- Echo the SQL to stdout -->
		<property name="show_sql">true</property>

		<!-- Set the current session context -->
		<property
name="current_session_context_class">thread</property>
		<property name="hbm2ddl.auto">update</property>

	</session-factory>

</hibernate-configuration>
```

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.bskgsa</groupId>
  <artifactId>OneToOne</artifactId>
  <version>0.0.1-SNAPSHOT</version>
   <dependencies>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-vibur</artifactId>
            <version>5.6.5.Final</version>
        </dependency>

        <dependency>
```

```
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.28</version>
        </dependency>
    </dependencies>
</project>
```

```
package OneToOne.com;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.JoinColumn;

import javax.persistence.OneToOne;

import javax.persistence.Table;


@Entity

@Table(name = "Book")

public class Book {

        @Id

        @GeneratedValue(strategy = GenerationType.IDENTITY)

        @Column(name = "id")

        private int bid;


        @Column(name = "Title")

        private String Title;


        @Column(name = "Author")

        private String Author;


        @OneToOne(cascade = CascadeType.ALL)
```

```
@JoinColumn(name = "c_id")
private Category Category; //object

public Book() {}

public Book(String Title, String Author)
{
        this.Title = Title;
        this.Author = Author;
}

public int getId() { return bid; }

public void setId(int bid) { this.bid = bid; }

public String getTitle() { return Title; }

public void setTitle(String Title)
{
        this.Title = Title;
}

public String getAuthor() { return Author; }

public void setAuthor(String Author)
{
        this.Author = Author;
}

public Category getCategory()
```

```
	{
		return Category;
	}

	public void
	setCategory(Category Category)
	{
		this.Category = Category;
	}

	@Override public String toString()
	{
		return "Book{"
			+ "bid=" + bid + ", Title='" + Title
			+ '\'' + ", Author='" + Author + '\''
		  + ", Category=" + Category
			+ '}';
	}

}

package OneToOne.com;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "Category")
public class Category {
```

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "c_id")
private int c_id;

@Column(name = "Category") private String Category;


public Category() {}

public Category(String Category)

{
        this.Category = Category;

}

public int getId() { return c_id; }

public void setId(int c_id) { this.c_id = c_id; }

public String getCategory() { return Category; }

public void setCategory(String Category)
{
        this.Category = Category;
}



@Override public String toString()
```

```
	{
		return "Category{"
			+ "Cid=" + c_id + ", Category='" + Category
			+ '}';
	}

}

package OneToOne.Main;
import OneToOne.com.Book;
import OneToOne.com.Category;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class AddingMain {
	public static void main(String[] args)
	{

		// Create session factory
		SessionFactory factory
			= new Configuration()
				.configure("hibernate.cfg.xml")
				.addAnnotatedClass(Book.class)
				.addAnnotatedClass(Category.class)
				.buildSessionFactory();

		// Create session
		try ( Session session = factory.getCurrentSession()) {
			// Get the current session
```

```
                    // Create relevant object.
                    Book Book = new Book("HarryPorter","J.K.Rowling");

                    Category Category
                            = new Category("Fantasy");

                    Book.setCategory(Category);

                    // Begin the transaction
                    session.beginTransaction();

                    // Save the student object.
                    // This will also save the StudentGfgDetail
                    // object as we have used CascadeType.ALL
                    session.save(Book);

                    // Commit the transaction
                    session.getTransaction().commit();

                    System.out.println(
                            "Transaction Successfully Completed!");
                }
                catch (Exception e) {
                    e.printStackTrace();
                }
        }

}
```

2. write and explain hibernate.cfg and hibernate.hbm file usage in orm.

In Hibernate, which is an Object-Relational Mapping (ORM) framework for Java, the configuration of the database connection and mapping between Java objects and database tables is done through two important files: `hibernate.cfg.xml` and `*.hbm.xml` files.

1. `hibernate.cfg.xml`: The `hibernate.cfg.xml` file is the configuration file that holds all the necessary settings for Hibernate to interact with the database. It is typically placed in the root of the classpath. Here's a typical example of what this file might look like:

```xml
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN" "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <!-- Database connection settings -->
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mydatabase</property>
    <property name="hibernate.connection.username">username</property>
    <property name="hibernate.connection.password">password</property>
    <!-- ORM settings -->
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.format_sql">true</property>
    <!-- Mapping files -->
    <mapping resource="com/example/MyEntity.hbm.xml"/>
    <!-- More mapping files can be added here -->
  </session-factory>
</hibernate-configuration>
```

Explanation of key elements:

- `<session-factory>`: This is the root element of the configuration file.
- `<property>`: These elements set various properties for Hibernate. For example, the database connection properties (driver class, URL, username, password), the SQL dialect used by the database, and whether to show formatted SQL queries or not.
- `<mapping>`: This element specifies the path to the Hibernate mapping file (`*.hbm.xml`) for each entity class. It tells Hibernate how to map the Java objects to database tables.

2. `*.hbm.xml` (Entity Mapping) files: For each entity in your application, you will create a corresponding `*.hbm.xml` file. These files define the mapping between the Java class properties and the corresponding database table columns. Here's an example of a `*.hbm.xml` file:

```xml
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN" "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.example.MyEntity" table="my_entity_table">
    <id name="id" type="long">
      <generator class="native" />
    </id>
    <property name="name" column="name" type="string" />
    <property name="age" column="age" type="integer" />
    <!-- More properties can be defined here -->
  </class>
</hibernate-mapping>
```

Explanation of key elements:

- `<class>`: This element defines the mapping for a specific entity class.
- `name`: The fully qualified class name of the entity.

- `table`: The name of the database table that corresponds to the entity.
- `<id>`: This element maps the primary key property of the entity.
- `type`: The data type of the property.
- `<generator>`: This element defines the strategy to generate the primary key values.

When Hibernate is initialized, it reads the `hibernate.cfg.xml` file to configure the database connection and other settings. It also reads all the `*.hbm.xml` files to understand the mappings between entities and database tables. With this information, Hibernate can perform CRUD (Create, Read, Update, Delete) operations on the database using Java objects, making it easier to work with the database in an object-oriented manner.

3. Explain advantages of HQL and caching in Hibernate.

Advantages of HQL (Hibernate Query Language):

1. Object-Oriented Querying: HQL allows developers to write queries in an object-oriented manner using Java classes and their properties instead of dealing with SQL queries directly. This makes it easier to express complex queries in a more intuitive and natural way.
2. Database Independence: With HQL, you can write database-independent queries. Hibernate takes care of translating the HQL queries to the appropriate SQL dialect for the underlying database, allowing your application to be more portable across different database systems.
3. Abstracts Database-Specific Syntax: HQL abstracts away the database-specific SQL syntax, which helps in avoiding potential SQL injection vulnerabilities and provides a consistent query language for all supported databases.
4. Eager and Lazy Loading Control: HQL allows developers to control the fetching strategy for related entities, giving them the flexibility to fetch only the required data and avoid performance issues related to unnecessary data loading.
5. Code Reusability: HQL queries can be defined as named queries and stored in the mapping files or Java annotations, making them reusable and maintainable across the application.
6. Powerful Aggregation Functions: HQL supports powerful aggregation functions like COUNT, SUM, AVG, MIN, MAX, etc., which simplifies data analysis tasks and allows developers to retrieve calculated results directly from the database.

Advantages of Caching in Hibernate:

1. Improved Performance: Caching helps reduce the number of database round-trips, which can significantly improve application performance. Reusing cached data avoids the overhead of querying the database repeatedly for the same data, resulting in faster response times.
2. Reduced Database Load: By storing frequently accessed data in the cache, Hibernate reduces the load on the database server, improving its overall scalability and allowing it to handle more concurrent users and requests.
3. Offline Access: Caching allows the application to continue serving requests even when the database is temporarily unavailable or during network issues. As long as the required data is available in the cache, the application can function independently of the database.
4. Optimistic Concurrency Control: Hibernate's caching mechanism can be used to implement optimistic concurrency control. By caching the version of an entity, Hibernate can detect if another user has modified the same entity in the database during a user's session, helping to prevent data inconsistency issues.
5. Customizable Cache Strategies: Hibernate offers various caching strategies like read-only, read/write, transactional, and query caching. Developers can choose the appropriate caching strategy based on the specific requirements of their application to strike a balance between performance and data consistency.
6. Second-Level Cache: Hibernate provides a second-level cache that can be shared among multiple sessions. This cache can be configured with external cache providers like Ehcache or Infinispan, which allows caching to be distributed across multiple application nodes, improving performance in clustered environments.

Overall, HQL and caching in Hibernate are powerful features that contribute to better application performance, increased productivity, and reduced database load, making Hibernate a popular choice for Java developers when working with relational databases and object-oriented paradigms.

4. Describe the sessionFactory , session, Transaction objects.

In Hibernate, the `SessionFactory`, `Session`, and `Transaction` are fundamental objects used to interact with the database and manage the persistence of Java objects. Let's explore each object:

1. `SessionFactory`: The `SessionFactory` is a thread-safe object responsible for creating and managing `Session` objects. It is typically instantiated once for the entire application, and it represents a connection pool to the database. The `SessionFactory` is a heavyweight object, and its creation is an expensive operation, so it's usually created during the application's startup and kept throughout its lifecycle.

To create a `SessionFactory`, you typically use the Hibernate configuration settings, such as the database connection properties and the mappings. Once the `SessionFactory` is created, it caches the compiled mapping metadata, so it can efficiently create `Session` objects when needed.

Example of creating a `SessionFactory`:

```
javaCopy code
import org.hibernate.SessionFactory; import org.hibernate.cfg.Configuration; public class HibernateUtil { private static final SessionFactory sessionFactory; static { try { Configuration configuration = new Configuration(); configuration.configure("hibernate.cfg.xml"); sessionFactory = configuration.buildSessionFactory(); } catch (Throwable ex) { throw new ExceptionInInitializerError(ex); } } public static SessionFactory getSessionFactory() { return sessionFactory; } }
```

2. `Session`: The `Session` object represents a single unit of work with the database. It is a lightweight and non-thread-safe object created by the `SessionFactory`. The `Session` is the main object used to interact with the database, including saving, updating, deleting, and querying objects.

A `Session` should typically be used within a single method or transactional context. Once the operation is completed or the transaction is committed or rolled back, the `Session` should be closed to release database resources.

Example of using a `Session`:

```
javaCopy code
import org.hibernate.Session; import org.hibernate.Transaction; public class MainClass { public static void main(String[] args) { try (Session session = HibernateUtil.getSessionFactory().openSession()) { Transaction transaction = session.beginTransaction(); // Perform database operations here // Save, update, delete, or query objects using the session transaction.commit(); // Commit the transaction } catch (Exception e) { // Handle exceptions and rollback the transaction if needed } } }
```

3. `Transaction`: The `Transaction` object represents a unit of work that groups a set of database operations into a single atomic and consistent operation. The `Transaction` is associated with a `Session`, and it allows you to control the persistence of objects and manage their changes to the database.

A transaction can be committed to permanently save the changes to the database or rolled back to discard the changes in case of any errors or failures.

In the above example, you can see that the `Transaction` is created using `session.beginTransaction()` and then later committed using `transaction.commit()`. If an exception occurs, the transaction can be rolled back using `transaction.rollback()`.

The proper handling of transactions ensures data integrity and consistency during the interaction with the database.

These three objects, `SessionFactory`, `Session`, and `Transaction`, play crucial roles in the Hibernate framework, facilitating smooth communication with the database and providing a higher-level abstraction over low-level database operations.