JAVA FULL STACK DEVELOPMENT PROGRAM

Session 13: Hibernate Advance

OUTLINE

- Hibernate Query
  - Criteria
  - Native Query
- Hibernate Fetching
- Hibernate Caching
- Hibernate Locking Introduction

CRITERIA

- Legacy Criteria API is deprecated now, but some companies still use them.
  - It comes with org.hibernate.Criteria package
- Now we are going to focus on the new Criteria API, which comes with javax.persistence.criteria.CriteriaQuery since Hibernate 5.
  - The JPA API to be superior as it represents a clean look

CRITERIA

- What is the problem with HQL or SQL?
  - String — It is represented as String, which means we can only validate the correctness of it during runtime.
- Criteria queries are a programmatic, type-safe way to express a query
  - They are type-safe in terms of using interfaces and classes to represent various structural parts of a query such as the query itself, the select clause, or an order-by, etc
  - They can also be type-safe in terms of referencing attributes

CRITERIA

- Step to use Criteria API in Hibernate 5: (One way)
  - Create an instance of *Session* from the *SessionFactory* object
  - Create an instance of C*riteriaBuilde* by calling the *getCriteriaBuilder()* method
  - Create an instance of *CriteriaQuery* by calling the *CriteriaBuilder createQuery() method*
  - Create an instance of *Query* by calling the *Session createQuery()* method
  - Call the *getResultList()* method of the *query* object which gives us the results

CRITERIA

```java
public class Item implements Serializable {

    private Integer itemId;
    private String itemName;
    private String itemDescription;
    private Integer itemPrice;

   // standard setters and getters
}
```

```java
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Item> cr = cb.createQuery(Item.class);
Root<Item> root = cr.from(Item.class);
cr.select(root);

Query<Item> query = session.createQuery(cr);
List<Item> results = query.getResultList();
```

CRITERIA SELECT

- Selecting an Entity

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Item> cr = cb.createQuery(Item.class);
Root<Item> root = cr.from(Item.class);
cr.select(root);
cr.where(cb.equal(root.get("itenName"), "Hibernate"));

Query<Item> query = session.createQuery(cr);
List<Item> results = query.getResultList();
```

- Selecting an attribute

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<String> cr = cb.createQuery(String.class);
Root<Item> root = cr.from(Item.class);
cr.select(root.get("itemDescription"));
cr.where(cb.equal(root.get("itenName"), "Hibernate"));

Query<String> query = session.createQuery(cr);
List<String> results = query.getResultList();
```

CRITERIA SELECT

- Select multiple values
  - using Object array
  - using wrapper (Recommended)

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Object[]> cr = cb.createQuery(String.class);
Root<Item> root = cr.from(Item.class);
cr.multiselect(root.get("itemDescription"), root.get("itemPrice"));
cr.where(cb.equal(root.get("itenName"), "Hibernate"));

Query<Object[]> query = session.createQuery(cr);
List<Object[]> results = query.getResultList();
```

```
public class ItemResultWrapper {

    private String itemDescription;
    private Integer itemPrice;

   // standard setters and getters
}


Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<ItemResultWrapper> cr = cb.createQuery(String.class);
Root<Item> root = cr.from(Item.class);
cr.multiselect(root.get("itemDescription"), root.get("itemPrice"));
cr.where(cb.equal(root.get("itenName"), "Hibernate"));

Query<ItemResultWrapper> query = session.createQuery(cr);
List<ItemResultWrapper> results = query.getResultList();
```

CRITERIA JOIN

- See example in Eclipse

NATIVE QUERY

- Execution of native SQL queries is controlled via the NativeQuery interface

- Session.createNativeQuery(“SQL”)

FETCHING

- Fetching, essentially, is the process of grabbing data from the database and making it available to the application.
- Tuning how an application does fetching is one of the biggest factors in determining how an application will perform.
  - Too much — unnecessary overhead in terms of both JDBC communication and ResultSet processing
  - Too little — additional fetching
- So there are two question about FETCHING
  - When should the data be fetched? Now? Later?
  - How should the data be fetched?

FETCHING

- Eager — Fetch all attributes of an entity at once, including Aggregation / Composition

- Lazy — Load the Aggregation / Composition fields when requested.

- For example, one user will have a list of address. If we set the fetch type as EAGER, then Hibernate will load all addresses for the user; while for LAZY fetching, Hibernate will load the address as we explicitly request or when we access the address in the address list.

DYNAMIC FETCHING

- Prerequisite — we should use fetch type as LAZY
- How to dynamic fetch?
  - Loop through the collection
  - Join with certain entity

CACHING

- Hibernate provides two level cache
  - First-level cache
    - Hibernate first level cache is associated with the Session object
    - Hibernate first level cache is enabled by default and there is no way to disable it
    - However we can delete the selected object from it
- Second-level cache
  - Hibernate Second Level cache is disabled by default but we can enable it through configuration
  - Second-level cache is *SessionFactory*-scoped, meaning it is shared by all sessions created with the same session factory

FIRST-LEVEL CACHE

- Hibernate first level cache is session specific, that's why when we are getting the same data in same session there is no query fired whereas in other session query is fired to load the data.
- Hibernate first level cache can have old values — Two session works on the same object, the second session update the object and save to database, but it won't reflect in the first session cache.
- We can use session contains() method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false
- Once the session is closed, first-level cache is terminated as well

SECOND-LEVEL CACHE

- When an entity instance is looked up by its id and if second-level caching is enabled for that entity, the following happens:
  - If an instance is already present in the first-level cache, it is returned from there
  - If an instance is not found in the first-level cache, and the corresponding instance state is cached in the second-level cache, then the data is fetched from there and an instance is assembled and returned
  - Otherwise, the necessary data are loaded from the database and an instance is assembled and returned
- Hibernate provides second-level cache interface, so we have to provide an implementation of the interface
  - EH Cache
  - OS Cache

SECOND-LEVEL CACHE

- Region Factory
  - Hibernate second-level caching is designed to be unaware of the actual cache provider used.
  - Hibernate only needs to be provided with an implementation of the *org.hibernate.cache.spi.RegionFactory* interface which encapsulates all details specific to actual cache providers.
  - Basically, it acts as a bridge between Hibernate and cache providers.

SEE EXAMPLE OF SECOND LEVEL CACHE

LOCKING

- In a relational database, locking refers to actions taken to prevent data from changing between the time it is read and the time is used
- Optimistic
- Pessimistic
  - Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.
  - This type of lock depends on underlying database system, and Hibernate will let database to handle it with proper isolation level provided.

OPTIMISTIC

- Optimistic locking assumes that multiple transactions can complete without affecting each other
- Therefore transactions can proceed without locking the data resources that they affect.
- Before committing, each transaction verifies that no other transaction has modified its data.
- If the check reveals conflicting modifications, the committing transaction rolls back.
- This approach guarantees some isolation, but scales well and works particularly well in *read-often-write-sometimes* situations.

OPTIMISTIC

- There are several rules which we should follow while declaring version attributes:
  - Each entity class must have only one version attribute
  - It must be placed in the primary table for an entity mapped to several tables
  - Type of a version attribute must be one of the following: *int*, *Integer*, *long*, *Long*, *short*, *Short*, *java.sql.Timestamp*

HIBERNATE CONNECTION POOL

- As we know, Hibernate has its own connection in place to get you started.

- However, it is quite rudimentary and should not be used in production.

- For Production environment, consider using the datasource from JNDI like JDBC or using a third party pool for best performance and stability.

SUMMARY

- ORM
- Hibernate Configuration (XML / Annotation)
- Entity Management (States)
- Hibernate Mapping
- Query (HQL / Criteria / SQL / Named Query)
- Fetching
- Caching
- Locking

ANY QUESTION?