JAVA BACKEND DEVELOPMENT PROGRAM

Hibernate Advance

OUTLINE

- Relationship Mapping
- Hibernate Fetching
- Hibernate Query
  - HQL
  - Criteria
  - Native Query
- Hibernate Caching
- Hibernate Locking
  - Pessimistic
  - Optimistic
- Hibernate in Spring Boot
  - Transaction Management

RELATIONSHIP MAPPING

- In Java, how can we represent the relationship between two class?
  - Inheritance
  - Aggregation / Composition
- In Database, how can we represent the relationship between two table?
  - One to One
  - One to Many / Many to One
  - Many to Many

RELATIONSHIP MAPPING

- Association mappings are one of the key features of JPA and Hibernate.
- They model the relationship between two database tables as attributes in your domain model. (Aggregation)
- Three types of mapping supported:
  - one-to-one
  - many-to-one / one-to-many
  - many-to-many
- Note: We can map each of them as a uni- or bidirectional association.
  - That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and HQL or Criteria queries

ONE TO ONE

- One-to-one relationships are rarely used in relational table models.
  - An example for a one-to-one association could be a *Customer* and the *CurrentAddress*. Each *Customer* has exactly one *ShippingAddress* and each *ShippingAddress* belongs to one *Customer*.
  - On the database level, this mapped by a foreign key column either on the *ShippingAddress* or the *Customer* table
- We can have unidirectional or bidirectional mapping between those two entity.

ONE TO ONE

- Unidirectional mapping

```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```

- Bidirectional mapping

```
@Entity
public class Customer{

    @OneToOne
    @JoinColumn(name = “fk_shippingaddress”)
    private ShippingAddress shippingAddress;

    …
}
```

```
@Entity
public class ShippingAddress{

    @OneToOne(mappedBy = “shippingAddress”)
    private Customer customer;

    …
}
```

MANY TO ONE

- For example, An order consists of multiple items, but each item belongs to only one order.
- On database side, we need to store the primary key of the Order record as a foreign key in the OrderItem table
- With Hibernate or JPA, we can use @ManyToOne and @OneToMany

```
@Entity
public class OrderItem {

    @ManyToOne
    @JoinColumn(name = “fk_order”)
    private Order order;

    …
}
```

```
@Entity
public class Order {

    @OneToMany(mappedBy = “order”)
    private List<OrderItem> items = new ArrayList<OrderItem>();

    …
}
```

MANY TO MANY

- A typical example for such a many-to-many relationship are *Product*s and *Store*s.
  - Each *Store* sells multiple *Product*s and each *Product* gets sold in multiple *Store*s.
- On database side, it requires an additional conjunction table which contains the primary key pairs of the associated entities.
- With Hibernate or JPA, we don't need to map this conjunction table to an entity.

MANY TO MANY

- @ManyToMany

```
@Entity
public class Store {

    @ManyToMany
    @JoinTable(name = “store_product”,
           joinColumns = { @JoinColumn(name = “fk_store”) },
           inverseJoinColumns = { @JoinColumn(name = “fk_product”) })
    private Set<Product> products = new HashSet<Product>();

    …
}
```

```
@Entity
public class Product{

    @ManyToMany(mappedBy=”products”)
    private Set<Store> stores = new HashSet<Store>();

    …
}
```

- If we don’t provide any additional information in @ManyToMany, Hibernate uses its default mapping which expects an association table with the name of both entities and the primary key attributes of both entities. In this case, Hibernate uses the *Store_Product* table with the columns *store_id* and *product_id*.

RELATIONSHIP MAPPING

- What do we miss?
  - With Many-to-Many or Many-to-One relationship, how should we add an item into the collection?
  - store.getProducts().add(product) — is it ok to use the statement in our business logic?

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

HIBERNATE QUERY

- There are several ways to query database using Hibernate:
  - HQL — Hibernate Query Language
  - Criteria
  - Native Query

HQL

- Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties.
- HQL queries are translated by Hibernate into conventional SQL queries, which in turns perform action on database.
- Although the SQL statements are also supported by Hibernate, we should use HQL as much as possible to achieve database portability
- Note: Keywords like SELECT, FROM, and WHERE, etc., are NOT case sensitive, but properties like table and column names are *case sensitive* in HQL
- Link to full reference: https://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/queryhql.html

HQL

```java
@Entity
public class DeptEmployee {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String employeeNumber;

    private String designation;

    private String name;

    @ManyToOne
    private Department department;

    // constructor, getters and setters
}
```

```java
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String name;

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

    public Department(String name) {
        this.name = name;
    }

    // getters and setters
}
```

```java
Query<DeptEmployee> query = session.createQuery("from DeptEmployee");
List<DeptEmployee> deptEmployees = query.list();
```

HQL

```java
@Entity
public class DeptEmployee {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String employeeNumber;

    private String designation;

    private String name;

    @ManyToOne
    private Department department;

    // constructor, getters and setters
}
```

```java
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private long id;

    private String name;

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

    public Department(String name) {
        this.name = name;
    }

    // getters and setters
}
```

```java
Query query = session.createQuery("select m.name from DeptEmployee m where m.id = 1");
List employees = query.list();
Object[] employee = (Object[]) employees.get(0);
String name = employee[0]
```

HQL

- Custom Query Result
  - Using a Constructor in HQL
  - Using a ResultTransformer

```java
public class Result {
    private String employeeName;

    public Result(String employeeName) {
        this.employeeName = employeeName;
    }

    public Result() {
    }

    // getters and setters
}
```

```java
Query<Result> query = session.createQuery("select new Result(m.name)"
  + " from DeptEmployee m where m.id = 1");
List<Result> results = query.list();
Result result = results.get(0);


Query query = session.createQuery("select new Result(m.name)"
  + " from DeptEmployee m where m.id = 1");
query.setResultTransformer(Transformers.aliasToBean(Result.class));
List<Result> results = query.list();
Result result = results.get(0);
```

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 is more recommended as it is cleaner

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*riteriaBuilder* 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();
```

NATIVE QUERY

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

- Session.createNativeQuery(“SQL”)

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.

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 LOCKING

- 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 LOCKING

- 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 SUMMARY

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

HOW TO USE HIBERNATE IN SPRING BOOT APPLICATION?

@PRIMARY

- Sometimes we need to define multiple beans of the same type. In these cases, the injection will be unsuccessful because Spring has no clue which bean we need.

```java
@Component
@Primary
class Car implements Vehicle {}

@Component
class Bike implements Vehicle {}
```

```java
@Component
class Driver {
    @Autowired
    Vehicle vehicle;
}


@Component
class Biker {
    @Autowired
    @Qualifier("bike")
    Vehicle vehicle;
}
```

SPRING DATA ACCESS

- Spring Data Access
- DAO Support
- JdbcTemplate
- Hibernate Integration

HIBERNATE INTEGRATION

- Bootstrapping a SessionFactory with the native Hibernate API is a bit complicated and would take us quite a few lines of code

- Spring supports bootstrapping the SessionFactory – so that we only need a few lines of Java code or XML configuration

- Note: Hibernate Template is deprecated since Spring 3.0 and Hibernate 3.0.1 — We can use Hibernate API with Spring Transaction Management directly.

SPRING TRANSACTION MANAGEMENT

- Spring provides support for both programmatic and declarative transactions
- Programmatic Transactions
  - With programmatic transactions, transaction management code needs to be explicitly written so as to commit when everything is successful and rolling back if anything goes wrong. The transaction management code is tightly bound to the business logic in this case.
- Declarative Transactions
  - Declarative transactions separates transaction management code from the business logic. Spring supports declarative transactions using transaction advice (using AOP) via XML configuration in the spring context or with *@Transactional* annotation.

```xml
<beans>

    <bean id="myTxManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="mySessionFactory"/>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="transactionManager" ref="myTxManager"/>
        <property name="productDao" ref="myProductDao"/>
    </bean>

</beans>
```

```java
public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus status) {
                List productsToChange = this.productDao.loadProductsByCategory(category);
                // do the price increase...
            }
        });
    }
}
```

DECLARATIVE TRANSACTIONS

- *@Transactional* can be either at the class or method level with further configuration
  - The *Propagation Type* of the transaction — that the same transaction will propagate from a transactional caller to transactional callee
    - e.g. if a read-only transaction calls a read-write transaction method, the whole transaction will be read-only
- The *Isolation Level* of the transaction — Isolation level defines a contract between transactions.
- A *Timeout* for the operation wrapped by the transaction
- A *readOnly flag* – a hint for the persistence provider that the transaction should be read only
- The *Rollback* rules for the transaction — by default, rollback happens for runtime, unchecked exceptions only

ANY QUESTIONS?