Day19: JPA with Hibernate, JPQL

1. JPA with Hibernate

Hibernate is one of the most popular Java ORM frameworks in use today. Its first release was almost twenty years ago, and still has excellent community support and regular releases. Additionally, **Hibernate is a standard implementation of the JPA specification**, with a few additional features that are specific to Hibernate.

2. Entity

Entities in JPA are nothing but POJOs representing data that can be persisted to the database. An entity represents a table stored in a database. Every instance of an entity represents a row in the table.

2.1. The *Entity* Annotation

Let's say we have a POJO called *Student,* which represents the data of a student, and we would like to store it in the database:

```
publicclassStudent {

    // fields, getters and setters

}
```

In order to do this, we should define an entity so that JPA is aware of it.

So let's define it by making use of the *@Entity* annotation. We must specify this annotation at the class level. **We must also ensure that the entity has a no-arg constructor and a primary key:**

```
@Entity
publicclassStudent {
```

```
    // fields, getters and setters

}
```

The entity name defaults to the name of the class. We can change its name using the *name* element:

```
@Entity(name="student")
publicclassStudent {

    // fields, getters and setters

}
```

Because various JPA implementations will try subclassing our entity in order to provide their functionality, **entity classes must not be declared *final*.**

2.2. The *Id* Annotation

Each JPA entity must have a primary key that uniquely identifies it. The *@Id* annotation defines the primary key. We can generate the identifiers in different ways, which are specified by the *@GeneratedValue* annotation.

We can choose from four id generation strategies with the *strategy* element. **The value can be *AUTO, TABLE, SEQUENCE,* or *IDENTITY:***

```
@Entity
publicclassStudent {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
private Long id;

private String name;

    // getters and setters
}
```

If we specify *GenerationType.AUTO*, the JPA provider will use any strategy it wants to generate the identifiers.

If we annotate the entity's fields, the JPA provider will use these fields to get and set the entity's state. In addition to Field Access, we can also do Property Access or Mixed Access, which enables us to use both Field and Property access in the same entity**.**

2.3. The *Table* Annotation

In most cases, **the name of the table in the database and the name of the entity won't be the same.**

In these cases, we can specify the table name using the *@Table* annotation:

```
@Entity
@Table(name="STUDENT")
publicclassStudent {

    // fields, getters and setters

}
```

We can also mention the schema using the *schema* element:

```
@Entity
@Table(name="STUDENT", schema="SCHOOL")
publicclassStudent {

    // fields, getters and setters

}
```

Schema name helps to distinguish one set of tables from another.

If we don't use the *@Table* annotation, the name of the table will be the name of the entity.

2.4. The *Column* Annotation

Just like the *@Table* annotation, we can use the *@Column* annotation to mention the details of a column in the table.

The *@Column* annotation has many elements such as *name, length, nullable, and unique*:

```
    @Column(name="STUDENT_NAME", length=50, nullable=false, unique=false)
private String name;
```

The *name* element specifies the name of the column in the table. The *length* element specifies its length. The *nullable* element specifies whether the column is nullable or not, and the *unique* element specifies whether the column is unique.

If we don't specify this annotation, the name of the column in the table will be the name of the field.

2.5. The *Transient* Annotation

Sometimes, we may want to **make a field non-persistent.** We can use the *@Transient* annotation to do so. It specifies that the field won't be persisted.

For instance, we can calculate the age of a student from the date of birth.

So let's annotate the field *age* with the *@Transient* annotation:

```
    @Transient
private Integer age;
```

As a result, the field *age* won't be persisted to the table.

2.6. The *Temporal* Annotation

In some cases, we may have to save temporal values in our table.

For this, we have the *@Temporal* annotation:

```
    @Temporal(TemporalType.DATE)
private Date birthDate;
```

However, with JPA 2.2, we also have support for *java.time.LocalDate, java.time.LocalTime, java.time.LocalDateTime, java.time.OffsetTime* and *java.time.OffsetDateTime.*

2.7. The *Enumerated* Annotation

Sometimes, we may want to persist a Java *enum* type.

We can use the *@Enumerated* annotation to specify whether the *enum* should be persisted by name or by ordinal (default):

```
publicenumGender {
    MALE,
    FEMALE
}
```

```
    @Enumerated(EnumType.STRING)
private Gender gender;
```

Actually, **we don't have to specify the *@Enumerated* annotation at all if we're going to persist the *Gender* by the *enum*‘s ordinal.**

However, to persist the *Gender* by *enum* name, we've configured the annotation with *EnumType.STRING.*

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="STUDENT_NAME", length=50, nullable=false, unique=false)
    private String name;

    @Transient
```

```
    private Integer age;

    @Temporal(TemporalType.DATE)
    private Date birthDate;

    @Enumerated(EnumType.STRING)
    private Gender gender;

    // other fields, getters and setters
}
```

JPQL

The `javax.persistence.Query` interface is the mechanism for issuing queries in JPA. The primary query language used is the Java Persistence Query Language, or `JPQL`
JPQL is syntactically very similar to SQL, but is object-oriented rather than table-oriented.

Query Basics

```
SELECT x FROM Magazine x
```

The preceding is a simple JPQL query for all `Magazine` entities.

```
public Query createQuery (String jpql);
```

The `EntityManager.createQuery` method creates a `Query` instance from a given JPQL string.

```
public List getResultList ();
```

Invoking `Query.getResultList` executes the query and returns a `List` containing the matching objects. The following example executes our `Magazine` query above:

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x");
List<Magazine> results = (List<Magazine>) q.getResultList ();
```

When selecting entities, you can optionally use the keyword `object`. The clauses `select x` and `SELECT OBJECT(x)` are synonymous.

The optional `where` clause places criteria on matching results. For example:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ'
```

Keywords in JPQL expressions are case-insensitive, but entity, identifier, and member names are not. For example, the expression above could also be expressed as:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ'
```

But it could not be expressed as:

```
SELECT x FROM Magazine x WHERE x.TITLE = 'JDJ'
```

As with the `select` clause, alias names in the `where` clause are resolved to the entity declared in the `from` clause. The query above could be described in English as "for all `Magazine` instances `x`, return a list of every `x` such that `x`'s `title` field is equal to 'JDJ'".

JPQL uses SQL-like syntax for query criteria. The `and` and `or` logical operators chain multiple criteria together:

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ' OR x.title = 'JavaPro'
```

The `=` operator tests for equality. `<>` tests for inequality. JPQL also supports the following arithmetic operators for numeric comparisons: `>, >=, <, <=`. For example:

```
SELECT x FROM Magazine x WHERE x.price > 3.00 AND x.price <= 5.00
```

This query returns all magazines whose price is greater than 3.00 and less than or equal to 5.00.

```
SELECT x FROM Magazine x WHERE x.price <> 3.00
```

This query returns all Magazines whose price is not equals to 3.00.

You can group expressions together using parentheses in order to specify how they are evaluated. This is similar to how parentheses are used in Java. For example:

```
SELECT x FROM Magazine x WHERE (x.price > 3.00 AND x.price <= 5.00) OR x.price = 7.00
```

This expression would match magazines whose price is 4.00, 5.00, or 7.00, but not 6.00. Alternately:

```
SELECT x FROM Magazine x WHERE x.price > 3.00 AND (x.price <= 5.00 OR x.price = 7.00)
```

This expression will magazines whose price is 5.00 or 7.00, but not 4.00 or 6.00.

JPQL also includes the following conditionals:

- `[NOT] BETWEEN`: Shorthand for expressing that a value falls between two other values. The following two statements are synonymous:

  ```
  SELECT x FROM Magazine x WHERE x.price >= 3.00 AND x.price <= 5.00
  ```

  ```
  SELECT x FROM Magazine x WHERE x.price BETWEEN 3.00 AND 5.00
  ```

- `[NOT] LIKE`: Performs a string comparison with wildcard support. The special character '_' in the parameter means to match any single character, and the special character '%' means to match any sequence of characters. The following statement matches title fields "JDJ" and "JavaPro", but not "IT Insider":

  ```
  SELECT x FROM Magazine x WHERE x.title LIKE 'J%'
  ```

The following statement matches the title field "JDJ" but not "JavaPro":

```
SELECT x FROM Magazine x WHERE x.title LIKE 'J__'
```

- `[NOT] IN`: Specifies that the member must be equal to one element of the provided list. The following two statements are synonymous:

```
SELECT x FROM Magazine x WHERE x.title IN ('JDJ', 'JavaPro', 'IT Insider')
```

```
SELECT x FROM Magazine x WHERE x.title = 'JDJ' OR x.title = 'JavaPro' OR x.title = 'I
T Insider'
```

- `IS [NOT] EMPTY`: Specifies that the collection field holds no elements. For example:

```
SELECT x FROM Magazine x WHERE x.articles is empty
```

This statement will return all magazines whose `articles` member contains no elements.

- `IS [NOT] NULL`: Specifies that the field is equal to null. For example:

```
SELECT x FROM Magazine x WHERE x.publisher is null
```

This statement will return all Magazine instances whose "publisher" field is set to `null`.

- `NOT`: Negates the contained expression. For example, the following two statements are synonymous:

```
SELECT x FROM Magazine x WHERE NOT(x.price = 10.0)
```

```
SELECT x FROM Magazine x WHERE x.price <> 10.0
```

10.1.2. Relation Traversal

Relations between objects can be traversed using Java-like syntax. For example, if the Magazine class has a field named "publisher" or type Company, that relation can be queried as follows:

```
SELECT x FROM Magazine x WHERE x.publisher.name = 'Random House'
```

This query returns all Magazine instances whose publisher field is set to a Company instance whose name is "Random House".

Single-valued relation traversal implies that the relation is not null. In SQL terms, this is known as an inner join. If you want to also include relations that are null, you can specify:

```
SELECT x FROM Magazine x WHERE x.publisher.name = 'Random House' or x.publisher is null
```

You can also traverse collection fields in queries, but you must declare each traversal in the from clause. Consider:

```
SELECT x FROM Magazine x, IN(x.articles) y WHERE y.authorName = 'John Doe'
```

This query says that for each Magazine x, traverse the articles relation and check each Article y, and pass the filter if y's authorName field is equal to "John Doe". In short, this query will return all magazines that have any articles written by John Doe.

The IN() syntax can also be expressed with the keywords inner join. The statements

```
SELECT x FROM Magazine x, IN(x.articles) y WHERE y.authorName = 'John Doe'
```

and

```
SELECT x FROM Magazine x inner join x.articles y WHERE y.authorName = 'John Doe'
```

are synonymous.

10.1.3. Fetch Joins

JPQL queries may specify one or more join fetch declarations, which allow the query to specify which fields in the returned instances will be pre-fetched.

```
SELECT x FROM Magazine x join fetch x.articles WHERE x.title = 'JDJ'
```

The query above returns Magazine instances and guarantees that the articles field will already be fetched in the returned instances.

Multiple fields may be specified in separate join fetch declarations:

```
SELECT x FROM Magazine x join fetch x.articles join fetch x.authors WHERE x.title = 'JDJ'
```

Specifying the join fetch declaration is functionally equivalent to adding the fields to the Query's FetchConfiguration. See Section 5.6, “Fetch Groups”.

10.1.4. JPQL Functions

As well as supporting direct field and relation comparisons, JPQL supports a pre-defined set of functions that you can apply.

CONCAT(string1, string2): Concatenates two string fields or literals. For example:

```
SELECT x FROM Magazine x WHERE CONCAT(x.title, 's') = 'JDJs'
```

SUBSTRING(string, startIndex, length): Returns the part of the string argument starting at startIndex (1-based) and ending at length characters past startIndex.

```
SELECT x FROM Magazine x WHERE SUBSTRING(x.title, 1, 1) = 'J'
```

TRIM([LEADING | TRAILING | BOTH] [character FROM] string: Trims the specified character from either the beginning (LEADING), the ending (TRAILING), or both (BOTH) the beginning and ending of the string argument. If no trim character is specified, the space character will be trimmed.

```
SELECT x FROM Magazine x WHERE TRIM(BOTH 'J' FROM x.title) = 'D'
```

LOWER(string): Returns the lower-case of the specified string argument.

```
SELECT x FROM Magazine x WHERE LOWER(x.title) = 'jdj'
```

UPPER(string): Returns the upper-case of the specified string argument.

```
SELECT x FROM Magazine x WHERE UPPER(x.title) = 'JAVAPRO'
```

LENGTH(string): Returns the number of characters in the specified string argument.

```
SELECT x FROM Magazine x WHERE LENGTH(x.title) = 3
```

LOCATE(searchString, candidateString [, startIndex]): Returns the first index of searchString in candidateString. Positions are 1-based. If the string is not found, returns 0.

```
SELECT x FROM Magazine x WHERE LOCATE('D', x.title) = 2
```

ABS(number): Returns the absolute value of the argument.

```
SELECT x FROM Magazine x WHERE ABS(x.price) >= 5.00
```

SQRT(number): Returns the square root of the argument.

```
SELECT x FROM Magazine x WHERE SQRT(x.price) >= 1.00
```

MOD(number, divisor): Returns the modulo of number and divisor.

```
SELECT x FROM Magazine x WHERE MOD(x.price, 10) = 0
```

CURRENT_DATE: Returns the current date.

CURRENT_TIME: Returns the current time.

CURRENT_TIMESTAMP: Returns the current timestamp.

10.1.5. Polymorphic Queries

All JPQL queries are polymorphic, which means the from clause of a query includes not only instances of the specific entity class to which it refers, but all subclasses of that class as well. The instances returned by a query include instances of the subclasses that satisfy the query conditions. For example, the following query may return instances of Magazine, as well as Tabloid and Digest instances, where Tabloid and Digest are Magazine subclasses.

```
SELECT x FROM Magazine x WHERE x.price < 5
```

10.1.6. Query Parameters

JPQL provides support for parameterized queries. Either named parameters or positional parameters may be specified in the query string. Parameters allow you to re-use query templates where only the input parameters vary. A single query can declare either named parameters or positional parameters, but is not allowed to declare both named and positional parameters.

```
public Query setParameter (int pos, Object value);
```

Specify positional parameters in your JPQL string using an integer prefixed by a question mark. You can then populate the Query object with positional parameter values via calls to the setParameter method above. The method returns the Query instance for optional method chaining.

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x WHERE x.title = ?1 and x.price > ?2");
q.setParameter (1, "JDJ").setParameter (2, 5.0);

List<Magazine> results = (List<Magazine>) q.getResultList ();
```

This code will substitute JDJ for the ?1 parameter and 5.0 for the ?2 parameter, then execute the query with those values.

```
public Query setParameter (String name, Object value);
```

Named parameter are denoted by prefixing an arbitrary name with a colon in your JPQL string. You can then populate the Query object with parameter values using the method above. Like the positional parameter method, this method returns the Query instance for optional method chaining.

```
EntityManager em = ...
Query q = em.createQuery ("SELECT x FROM Magazine x WHERE x.title = :titleParam and x.price >
:priceParam");
q.setParameter ("titleParam", "JDJ").setParameter ("priceParam", 5.0);

List<Magazine> results = (List<Magazine>) q.getResultList ();
```

This code substitutes JDJ for the :titleParam parameter and 5.0 for the :priceParam parameter, then executes the query with those values.

10.1.7. Ordering

JPQL queries may optionally contain an order by clause which specifies one or more fields to order by when returning query results. You may follow the order by field clause with the asc or desc keywords, which indicate that ordering should be ascending or descending, respectively. If the direction is omitted, ordering is ascending by default.

```
SELECT x FROM Magazine x order by x.title asc, x.price desc
```

The query above returns Magazine instances sorted by their title in ascending order. In cases where the titles of two or more magazines are the same, those instances will be sorted by price in descending order.

10.1.8. Aggregates

JPQL queries can select aggregate data as well as objects. JPQL includes the min, max, avg, and count aggregates. These functions can be used for reporting and summary queries.

The following query will return the average of all the prices of all the magazines:

```
EntityManager em = ...
Query q = em.createQuery ("SELECT AVG(x.price) FROM Magazine x");
Number result = (Number) q.getSingleResult ();
```

The following query will return the highest price of all the magazines titled "JDJ":

```
EntityManager em = ...
Query q = em.createQuery ("SELECT MAX(x.price) FROM Magazine x WHERE x.title = 'JDJ'");
Number result = (Number) q.getSingleResult ();
```

10.1.9. Named Queries

Query templates can be statically declared using the NamedQuery and NamedQueries annotations. For example:

```
@Entity
@NamedQueries({
@NamedQuery(name="magsOverPrice",
query="SELECT x FROM Magazine x WHERE x.price > ?1"),
@NamedQuery(name"magsByTitle",
query="SELECT x FROM Magazine x WHERE x.title = :titleParam")
})
public class Magazine
{
...
}
```

These declarations will define two named queries called magsOverPrice and magsByTitle.

```
public Query createNamedQuery (String name);
```

You retrieve named queries with the above EntityManager method. For example:

```
EntityManager em = ...
Query q = em.createNamedQuery ("magsOverPrice");
q.setParameter (1, 5.0f);
List<Magazine> results = (List<Magazine>) q.getResultList ();
EntityManager em = ...
Query q = em.createNamedQuery ("magsByTitle");
q.setParameter ("titleParam", "JDJ");
```

```
List<Magazine> results = (List<Magazine>) q.getResultList ();
```

10.1.10. Delete By Query

Queries are useful not only for finding objects, but for efficiently deleting them as well. For example, you might delete all records created before a certain date. Rather than bringing these objects into memory and delete them individually, JPA allows you to perform a single bulk delete based on JPQL criteria.

Delete by query uses the same JPQL syntax as normal queries, with one exception: begin your query string with the delete keyword instead of the select keyword. To then execute the delete, you call the following Query method:

public int executeUpdate ();

This method returns the number of objects deleted. The following example deletes all subscriptions whose expiration date has passed.

Example 10.1. Delete by Query

```
Query q = em.createQuery ("DELETE s FROM Subscription s WHERE s.subscriptionDate < :today");
q.setParameter ("today", new Date ());
```

```
int deleted = q.executeUpdate ();
```

10.1.11. Update By Query

Similar to bulk deletes, it is sometimes necessary to perform updates against a large number of queries in a single operation, without having to bring all the instances down to the client. Rather than bring these objects into memory and modifying them individually, JPA allows you to perform a single bulk update based on JPQL criteria.

Update by query uses the same JPQL syntax as normal queries, except that the query string begins with the update keyword instead of select. To execute the update, you call the following Query method:

```
public int executeUpdate ();
```

This method returns the number of objects updated. The following example updates all subscriptions whose expiration date has passed to have the "paid" field set to true.

Example 10.2. Update by Query

```
Query q = em.createQuery ("UPDATE Subscription s SET s.paid = :paid WHERE s.subscriptionDate
< :today");
q.setParameter ("today", new Date ());
q.setParameter ("paid", true);
int updated = q.executeUpdate ();
```

References:

https://www.baeldung.com/learn-jpa-hibernate

https://www.baeldung.com/jpa-entities

https://docs.oracle.com/html/E13946_04/ejb3_overview_query.html

https://docs.oracle.com/html/E13946_04/ejb3_overview_query.html