Advanced Java

Agenda

- Query Methods
- Entity Relations

Spring Data

Query Methods

- Methods available in JpaRepository/CrudRepository provide basic CRUD operations.
- For application specific database queries build queries using keywords.
    - Query expressions are usually property traversals combined with concatenation operators And / Or as well as comparison operators Between, LessThan, Like, etc.
    - IgnoreCase for individual properties or all properties.
    - OrderBy for static ordering.

Query examples

```
Customer findByName(String name);
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
// Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
// Enabling ignoring case for an individual property
List<Person> findByLastnameIgnoreCase(String lastname);
// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByAddressZipCode(ZipCode zipCode);
// Assuming a Person has an Address with a ZipCode. In that case, the method creates the property traversal
x.address.zipCode.
```

```
Stream<Person> findByAgeBetween(int minAge, int maxAge);
// Limiting the result size of a query with Top or First
User findFirstByOrderByLastnameAsc();
User findTopByOrderByAgeDesc();
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);
List<User> findFirst10ByLastname(String lastname, Sort sort);
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

Query expression keywords

- Refer method subject keywords, predicate keywords and sorting keywords
  - https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

Repository implementation

- The implementation of the Spring Data repository hidden (not to implement in code).
- Implemented by SimpleJpaRepository
  - Provide method implementations.
  - Handles transactions using @Transactional (readonly=true).
- @Transactional is not necessary.
- To deal with multiple DAOs use @Transactional on @Service layer.

JPA Entity Relations

- RDBMS tables are designed by process of Normalization.
  - To avoid redundancy, data is organized into multiple tables.
  - These tables are related to each other by means of primary key and foreign key constraints.
  - ER diagram shows relations among the tables.
  - The relations can be OneToOne, OneToMany, ManyToOne, or ManyToMany.
- Java classes/objects are related by means of inheritance and associations.
  - JPA maps RDBMS table relations to Java class/object relations.
  - JPA supports both relations

    - Associations
    - Inheritance
  - Associations are supported with annotations @OneToMany, @ManyToOne, @ManyToMany, and @OneToOne
  - Inheritance is supported with annotations @MappedSuperclass, and @Inheritance.

OneToMany and ManyToOne

- OneToMany & ManyToOne represent parent-child relation between tables.
- Primary key of parent table is mapped to foreign key of child table.
- Example: One dept has Many employees.

```
// CREATE TABLE dept(deptno INT PRIMARY KEY, dname CHAR(20), loc CHAR(20));
@Entity
@Table(name="dept")
class Dept {
    @Id
    @Column private int deptno;
    @Column private String dname;
    @Column private String loc;
    @OneToMany(mappedBy="dept")
    private List<Emp> empList;
    // ...
}

// CREATE TABLE emp(empno INT PRIMARY KEY, ename CHAR(20), sal DOUBLE, deptno INT);
@Entity
@Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    @Column private double sal;
    @ManyToOne
    @JoinColumn(name="deptno")
```

```
    private Dept dept;
    // ...
}
```

- @OneToMany(mappedBy="fkField")
  - @OneToMany(mappedBy="deptno") -- uni-directional relation -- Emp has field int deptno;
  - @OneToMany(mappedBy="dept") -- bi-directional relation -- Emp has field Dept dept;
- @ManyToOne @JoinColumn(name="fkColumn")
  - @ManyToOne @JoinColumn(name="deptno") -- emp table has deptno column.

N+1 Problem

- When ManyToOne and OneToOne is used and multiple entities are searched at once e.g. "blogDao.findAll()", then associated entities are fetched with different query for each entity.
  - If "blogDao.findAll()" --> 3 blogs, Then 3 times queries are fired to get associated Category.
- This slow down execution (due to multiple queries on database).
- It can be done more efficiently by writing JPQL/HQL query.
  - "select b from Blog b join fetch b.category" --> Force to fetch category along with blog in the same join query.

FetchType

- Defines SELECT behaviour of associated entity.
- @OneToOne can be have fetch type LAZY or EAGER.
  - @OneToOne default fetch type = EAGER

CascadeType

- Defines entity life cycle of associated entity.
- Possible values can be PERSIST, MERGE, DETACH, REMOVE, REFRESH, and ALL.
- If Dept class has @OneToMany(cascade = CascadeType.XXX), then
  - PERSIST: insert Emp in list while inserting Dept (persist())
  - REMOVE: delete Emp in dept while deleting Dept (remove())

- DETACH: remove Emp in dept from session while removing Dept from session (detach())
- REFRESH: re-select Emp in dept while re-selecting Dept (refresh())
- MERGE: add Emp in dept intion session while adding Dept into session (merge())

OneToOne

- Example: One Emp have one Address.
- If both tables have same primary key, then use @OneToOne along along with @PrimaryKeyJoinColumn.

```
// create table emp (empno integer not null, deptno integer not null, ename varchar(20), sal double not null, primary
key (empno))
@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    // ...
    @OneToOne(mappedBy = "emp")
    private Address addr;
}

// create table address (addr varchar(60), country varchar(20), pin integer not null, state varchar(20), emp_empno
integer not null, primary key (emp_empno))
@Entity @Table(name="address")
class Address {
    @Id
    @Column private int empid;
    @Column private String country;
    // ...
    @OneToOne
    @PrimaryKeyJoinColumn
    @MapsId // PK of this table is taken from Emp class PK
    private Emp emp;
}
```

- Alternatively @OneToOne with @JoinColumn to map PK of a table to PK of another table.

```
// create table address (empid integer not null, addr varchar(60), country varchar(20), pin integer not null, state varchar(20), primary key (empid))
@Entity @Table(name="address")
class Address {
    @Id
    @Column private int empid;
    @Column private String country;
    // ...
    @OneToOne
    @JoinColumn(name="empid")
    @MapsId
    private Emp emp;
}

// create table emp (empno integer not null, deptno integer not null, ename varchar(20), sal double not null, primary key (empno))
@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    // ...
    @OneToOne(mappedBy = "emp")
    private Address addr;
}
```

- Use @OneToOne with mappedBy in second class to setup bidirectional relation.

ManyToMany

- Many-to-many relation is established into two tables via an additional table (auxilary table).
    - One Emp can have many Meetings.

    - One Meeting will have many Emps.
    - emp "1" ---- "n" emp_meeting "n" ---- "1" meeting
    - emp "n" ---- "n" meeting

- The emp_meeting table holds FK of both tables to establish the relation.
- In first class (e.g. Emp) use @ManyToMany along with @JoinTable (refering auxilary table & FK column in it).
    - joinColumn – first table's FK in aux table
    - inverseJoinColumn – second table's FK in aux table
- In second class (e.g. Meeting) use @ManyToMany with mappedBy to setup bi-directional relation.

```
// CREATE TABLE emp(empno INT PRIMARY KEY, ename CHAR(20), sal DOUBLE, deptno INT);
@Entity @Table(name="emp")
class Emp {
    @Id
    @Column private int empno;
    @Column private String ename;
    // ...
    @ManyToMany
    @JoinTable(name = "emp_meeting",
        joinColumns = {@JoinColumn (name="emp_id")},
        inverseJoinColumns = {@JoinColumn (name="meeting_id")} )
    private List<Meeting> meetingList;
    // ...
}

// CREATE TABLE meeting(id INT AUTO_INCREMENT PRIMARY KEY, subject CHAR(30), meetdate DATETIME);
@Entity @Table(name="meeting")
class Meeting {
    @Id
    @Column private int id;
    @Column private String subject;
    // ...
```

```
    @ManyToMany(mappedBy="meetingList")
    private List<Emp> empList;
}

// CREATE TABLE emp_meeting(emp_id INT REFERENCES emp(empno), meeting_id INT REFERENCES meeting(id));
```