

Git Repo :: <https://github.com/ashokitschool/SBMS-39.git>

Spring Data JPA

- 1) What is ORM & why
- 2) What is Data JPA & Why ?
- 3) Data JPA Repositories
 - CrudRepository
 - JpaRepository
- 4) What is Entity
- 5) Data JPA Application
- 6) Crud Operations
- 7) Pagination & Sorting
- 8) Query By Example (QBE)
- 9) findByXXX methods
- 10) Custom Queries (@Query)
- 11) Native SQL vs HQL
- 12) Primary Keys & Generators
- 13) Composite Primary Key
 - @Embeddable
 - @EmbeddedId
- 14) Timestamping
 - @CreationTimestamp
 - @UpdateTimestamp
- 15) Transactions
 - @Transactional
- 16) Connection Pooling
- 17) Association Mapping
- 18) Conclusion

What is persistence layer ?

=> It contains set of classes & interfaces to communicate with database.

=> In java, we have several options to develop persistence layer

- 1) JDBC
- 2) Hibernate ORM
- 3) Spring JDBC
- 4) Spring ORM
- 5) Spring Data JPA

Note: Spring Data JPA is the latest trend in the market to develop Persistence layer in java based applications.

Why to use Spring Data JPA ?

=> No need to write boilerplate code (duplicate)

=> No need to write queries

=> Ready made methods support is available

- save () - insert record into db table
- findById ()
- findAll ()
- count ()
- deleteAll ()

=> Reducing development time

What is ORM ?

=====

=> ORM stands for Object relational mapping

=> It is a technique to map java objects with database tables.

=> Using ORM we can deal with objects to perform DB operations.

=> When we are using ORM frameworks we need to map java classes with database tables.

User.java =====> USER_TBL

Product.java =====> PRODUCT_TBL

Note: The java class which is mapped with database table is called as Entity class.

Entity class -----> db table

Entity class variables -----> db tbl columns

Entity class obj -----> db tbl row

=> We will use below annotations to create Entity classes

@Entity : Represent java class as an Entity class

@Table : To map java class name with table name (Optional)

@Id : Represents entity variable mapped with PK column in table

@Column : To map java class variables with table column names
(optional)

=====

What is Jpa Repository ?

=====

=> Data JPA provided repository interfaces to simplify Persistence layer development.

a) CrudRepository

b) JpaRepository (more features)

Note: To perform DB operations we will create interface by extending from JpaRepository.

=====

Assignment

=====

Data JPA App with Oracle : <https://www.youtube.com/watch?v=ZGKHJC3p4hg>

=====

Developing First Data JPA Application

=====

1) DB Setup (MySQL DB Server + MySQL Workbench)

@@ Reference Video : [youtube.com/watch?v=EsAIXPIsyQg](https://www.youtube.com/watch?v=EsAIXPIsyQg)

```
show databases;
create database sbms39;
use sbms39;
show tables;
```

2) Create Spring Boot application with below dependencies

- a) data-jpa-starter
- b) mysql-driver

3) Configure Datasource properties in application.properties

```
spring.datasource.username=root
spring.datasource.password=ashokit@123
spring.datasource.url=jdbc:mysql://localhost:3306/sbms39

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true
```

4) Create Entity class (class to table mapping)

5) Create Repository interface (CrudRepository/JpaRepository)

```
public interface EmpRepository extends CrudRepository<Employee, Integer>{
}
```

6) Create Service class and inject Repository interface

7) Test service class methods from start class.

```
@@ save ( ) = insert + update ==> upsert
```

```
=====
Crud Repository Methods (12)
=====
```

=> CrudRepository is a spring data jpa repository interface

=> CrudRepository providing methods to perform crud operation

@@ Note: To use crud repo methods we need to extend properties

save(T) : one object for Upsert

saveAll(Iterable T) : Collection of objects for upsert

findById(ID id) : To retrieve record based on given pk value

findAllById(Iterable ids) : Retrieve records based on given pks

findAll() : Retrieve all records from table

existsById(ID id) : To check record presence in table

count () : To get total no.of records

deleteById(ID id) : To delete record based on given pk

deleteAllById(Iterable ids) : Delete multiple records based on pks

delete(T entity) : Delete record based on given entity obj

deleteAll(Iterable entities) : delete records based on entities

deleteAll () : To delete all records from table

```
=====
findByXXX methods
=====
```

=> findBy methods are used to retrieve records based on non-primary key column values

Note : findBy methods are used for select operations only

```
// select * from employee where ename=:ename
public List<Employee> findByEname(String ename);

// select * from employee where esalary=:esalary
public List<Employee> findByEsalary(Double salary);

//select * from employee where esalary >= :esalary
public List<Employee> findByEsalaryGreaterThanOrEqual(Double salary);
```

=====

Custom Queries

=====

=> Executing our own queries using data jpa.

=> To work with custom queries, we will use @Query annotation

=> Custom Queries we can write in 2 ways

- 1) Native SQL (plain sql)
- 2) HQL

=====

SQL Queries

=====

=> SQL Queries are db dependent queries.

=> In SQL query we will use table names and column names directley

Ex: select * from emp_tbl where emp_sal <= 10000.00

Note: If we want to change from one DB to another DB then we have to modify SQL queries and we have to re-test entire application.

=> To make our application loosely coupled with database we can use HQL queries.

=====

HQL Queries

=====

=> HQL stands for hibernate query language.

=> HQL queries are database independent.

=> HQL queries will make our app loosely coupled with database.

=> In HQL queries we will use entity class name and entity variable names.

Ex: From Employee where esal=1000.00

Note: Database can't understand HQL directley

=> HQL should be converted to SQL for execution.

=> Dialect classes are used to convert HQL to SQL.

=> Every DB will have its own dialect class

Ex: OracleDialect, MySQLDialect.....

SQL : select * from emp_tbl

HQL : From Employee

SQL : select * from emp_tbl where emp_id=101

HQL : From Employee where eid=101

SQL : select emp_id, emp_name from emp_tbl
 HQL : select eid, ename from Employee

```
=====

public interface EmpRepository extends CrudRepository<Employee, Integer> {

    @Query("from Employee")
    public List<Employee> getAllEmpsHQL();

    @Query("from Employee where eid=:id")
    public Employee getEmpById(Integer id);

    @Query(value = "select * from employee", nativeQuery = true)
    public List<Employee> getAllEmpsSQL();

    // select * from employee where ename=:ename
    public List<Employee> findByEname(String ename);

    // select * from employee where esalary=:esalary
    public List<Employee> findByEsalary(Double salary);

    // select * from employee where esalary >= :esalary
    public List<Employee> findByEsalaryGreaterThanEqual(Double salary);
}
```

=====

Q) Which is better SQL or HQL ?

=====

=> Performance wise SQL is better

=> Flexibility wise HQL is better

=====

JpaRepository

=====

=> This is predefined data jpa interface

=> It is providing several methods to perform DB ops

JpaRepo = CrudRepo + Pagination + Sorting + QBE

=====

What is Pagination ?

=====

=> Divide total records into multiple pages for display.

- decide page size (how many records shud display)
- calculage no.of pages required

Scenario-1 :

- Total records in db tbl : 50
- page size : 10
- total pages = total-records/page-size => 5 pages

Scenario-2 :

- Total records in db tbl : 500
- page size : 24
- total pages = total-records/page-size => 21 pages

```
=====
What is Sorting ?
=====
```

=> Sorting the records in ascending or descending order

Ex: display mobile based on price high to low

display emps based on salary low to high

```
=====
What is QBE ?
=====
```

=> QBE means Query By Example

=> It is used to construct query dynamically

=> It is used to implement dynamic search functionalities

```
=====
What is Timestamping
=====
```

=> It is used to populate record creation date and record updated date.

=> We will use below annotations in entity class to enable timestamping

```
@CreationTimestamp
@UpdateTimestamp
```

```
-----

@Entity
public class Product {

    @Id
    private Integer pid;
    private String name;
    private Double price;

    @Column(updatable = false)
    @CreationTimestamp
    private LocalDate createdDate;

    @Column(insertable = false)
    @UpdateTimestamp
    private LocalDate updatedDate;

    //setters & getters

}
```

```
-----

@Entity
@Table
@Id
@Column
@CreationTimestamp
@UpdateTimestamp
```

```
=====
Generators
=====
```

=> Generators are used to set the value for primary key columns

@@ Primary Key = Not null + unique constraints

=> Primary key is used to maintain unique records in table

=> For every table atleast one primary key is required.

=====

Generator Strategies

=====

table : it will maintain separate table for pks

identity : it supports auto_increment (mysql)

sequence : it supports db sequences (oracle)

uuid : alpha-numeric value for pk (datatype string)

@Id

@GeneratedValue(strategy = GenerationType.UUID)

private String pid;

=====

Composite Primary keys

=====

=> If table is having more than one primary key then it is called as composite primary key

```
create table person(
    pid int(10),
    name varchar(100),
    email varchar(100),
    passport varchar(100),
    primary key(pid, passport)
)
```

=====

@@ Custom Generator : <https://www.youtube.com/watch?v=IijGVtT9ZPk>

=====

Database Relationships

=====

=> We can see below relationship with db tables

1) One To One (Ex: Person with Passport)

Note: One record in parent table will have relationship with one record in child table.

Ex: One person will have one passport.

2) One To Many (Ex: Employee with Address)

Note: One record in parent table will have relationship with multiple records in child table.

Ex: One Employee can have multiple addresses.

3) Many To One (Ex: Books with Author)

Note : Many records in one table will have relationship with one record.

Ex: Multiple Books belongs to one Author

4) Many To Many (Ex : Users with Roles)

Ex: Multiple Users will have multiple roles.

Note: To represent Many To Many relationship we need 3 tables.

Ex : users, roles, user_roles (join table)

=> When DB tables having relationships then we have to represent those relationships in Entity classes which is called as Association Mapping.

=> To establish association mapping in entity classes we will use below annotations...

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany
- @JoinColumn
- @JoinTable

=====
What is Cascade ?
=====

=> Parent table Operations should reflect on child table or not will be represented by CASCADE.

=====
What is Fetch Type ?
=====

=> Fetch Type represents child records should be loaded along with parent record or not

=> We have below 2 fetch types

- Lazy (default)
- Eager

=> Lazy means child records will be retrieved on demand basis.

=> Eager means child records will be retrieved along with parent record in single query.

=====
One To One Relationship
=====

```
@Entity
@Table(name = "passport_tbl")
public class Passport {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer passportId;
    private String passportNum;
    private LocalDate issuedDate;
    private LocalDate expDate;

    @OneToOne
    @JoinColumn(name = "person_id")
    private Person person;

}
```

```
@Entity
@Table(name = "person_tbl")
public class Person {
```

```

@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer personId;

private String name;

private String gender;

@OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
private Passport passport;

}

```

```

=====
One To Many Relationship
=====

```

```

@Entity
@Table(name = "emp_tbl")
public class Emp {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer eid;

    private String ename;

    private Double esal;

    @OneToMany(
        mappedBy = "emp",
        cascade = CascadeType.ALL
    )

}

```

```

@Entity
@Table(name = "addr_tbl")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer addrId;

    private String city;

    private String state;

    private String country;

    private String type;

    @ManyToOne
    @JoinColumn(name = "eid")
    private Emp emp;

}

```

```

=====
MANY TO MANY RELATIONSHIP
=====

```

```

table - 1 : users_tbl    (users will be stored)

table - 2 : roles_tbl    (roles will be stored)

table - 3 : user_roles   (users & roles mapping will be stored here)

```

=====

Data JPA Specification

=====

=> Data JPA Specification is used to build dynamic queries based on certain conditions.

@Entity

```
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    private Double price;

    private String category;

    // setters & getters
}

public class ProductSpecifications {

    public static Specification<Product> nameLike(String name) {
        return (root, query, criteriaBuilder) -> criteriaBuilder.like(root.get("name"), "%" + name
+ "%");
    }

    public static Specification<Product> priceLessThan(double price) {
        return (root, query, criteriaBuilder) -> criteriaBuilder.lessThan(root.get("price"),
price);
    }

    public static Specification<Product> priceGreaterThan(double price) {
        return (root, query, criteriaBuilder) -> criteriaBuilder.greaterThan(root.get("price"),
price);
    }
}

public interface ProductRepo extends JpaRepository<Product, Long> {
    List<Product> findAll(Specification<Product> spec);
}
```

@Service

```
public class ProductService {

    @Autowired
    private ProductRepo productRepo;

    public List<Product> findProducts(String name, Double minPrice, Double maxPrice) {

        Specification<Product> spec = Specification.where(null);

        if (name != null) {
            spec = spec.and(ProductSpecifications.nameLike(name));
        }

        if (minPrice != null) {
            spec = spec.and(ProductSpecifications.priceGreaterThan(minPrice));
        }

        if (maxPrice != null) {
            spec = spec.and(ProductSpecifications.priceLessThan(maxPrice));
        }

        return productRepo.findAll(spec);
    }
}
```

```
}  
}
```

```
=====  
Assignments  
=====
```

1) Develop Data JPA application to insert person data into db table.

```
person_id  
person_name  
person_gender  
person_dob  
person_photo  
person_resume
```

2) Develop data jpa application to call stored procedure

3) Develop data jpa application to retrieve only emp_name and emp_salary details from employee_tbl using custom query.

4) Insert employee records into table using custom generator..

(Ex: AIT1, AIT2, AIT3.....)

5) Develop data jpa application to retrieve emp data along with address using custom query.

6) Write SQL queries to create PERSON_TBL & PASSPORT_TBL with one to one relationship.

7) Write SQL queries to create EMPLOYEE_TBL & ADDRESS_TBL with One to Many Relationship.

8) Implement Many To Many Relationship Example

Git Repo : https://github.com/ashokitschool/springboot_jpa_many_to_many_app.git

9) Develop Spring Boot application to export database table data into excel file.

10) Develop Spring Boot application to export database table data into pdf file.