Why ORM?

(Object Relational Mapping)

* The requirement is, mapping Java objects to databases. So that a Java application can send or retrive objects from the database.
* But Java objects follow OOP model and databases follow RDBMS model. Because of the style and the patterns of these two models are different, we have some mismatches.
1. In Object model, we have inheritance. But in relational model, it doesn’t exist.
2. In object model, we use reference variables to maintain associations. But relational model, we use foreign keys to maintain the associations.
3. In Object model, either == operator or equals() method can be used to check object equality. But in relational model, primary keys are used to uniquely identify the records.
4. In object model, dot(.) operator is used for data navigation. But in realtional model, join operations are used for navigation.
* To overcome this mismatches and to make object model and relational model work together, we got a technique called ORM.
* ORM maps Java objects to database tables in simple way with simple configurations.

JPA(Java Persistence API):

* JPA is a specification that defines how data persistence tasks are handled using ORM frameworks in Java applications.
* JPA is a specification and Hibernate, OpenJPA, Eclipse Link, etc.. are the implementations of this specification.
* This JPA specification provides some features.
1. provides annotations for mapping object model with relational model.
2. provides API to performs CRUD operations.
3. provides JPQL(Java Persistence Query Language) for querying the database.
* JPA needs Entity classes to perform CRUD operations.
* Entity class is a class in Java that is mapped with a database table.



* creating repository layer with JPA has an advantage that we can change the JPA provider, let say from Hibernate to OpenJPA, by without modifying the repository classes.
* Just some configuration changes are required.

Spring Data JPA:

* The limitations while using JPA are,
1. Programmer has to write the code to perform CRUD operations in the repository class.
2. A developer can define data access methods in a repository class in his own way. It leads to inconsistency in the data access layer.
* To overcome these limitations, Spring Data JPA module is introduced.
* Spring Data JPA eliminates writing the dao implementation classes from the repository layer.
* Only we write interfaces, but not implementation classes.
* Spring Data JPA will generate the implementation classes with required data access opertions at runtime.
* The interfaces are created by extending JpaRepository interface.

for example,

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Integer> {

}

* In spring configuration, we need to add the required configurations for Data JPA.
* The required configurations means, configuring DataSource, configurating EntityManager, configuring JpaProvider, configuring TransactionManager, etc…

Spring Boot Data JPA:

* In Spring Boot Data JPA, there is no need to define any configurations. Just creating repository interface for an entity class is enough.
* So, while working with Spring Boot Data JPA in the repository layer, a developer just has to defne entity classes and their related repository interfaces.

JpaRepository hierarchy:



* Repository is a marker interface, means an empty interface.

CrudRepository methods:

1. save(T entity) : saves the given entity. If entity already exist, it updates it.
2. findById(ID id): retrieves entity by its id.
3. findAll() : returns all entities
4. deleteById(ID id): deletes an entity with the given id.
5. delete(T entity): deletes a given entity.
6. deleteAll(): delets all entities.
7. existsById(ID id): checks if an entity exist with the given id.

PagingAndSorting methods:

1. findAll(Sort sort): returns all entities sorted according to the provied sort object.
2. findAll(Pageable pageable): returns a Page object, with entities meeting the restriction provided in the pageable object.

JpaRepository methods:

* JpaRepository is the most commonly used interface, because it provides crud operations, paging and sorting operations and also some JPA-specific operations.
1. saveAndFlush(T entity): saves and entity and flushes the changes instantly.
2. deleteAllInBatch(): deletes all entities in a batch.
3. getOne(ID id): returns a reference to the entity with the given id, without actually loading it.



DataJpaCrudApplication.java

@SpringBootApplication

**public** **class** DataJpaCrudApplication {

 **public** **static** **void** main(String[] args) {

 SpringApplication.*run*(DataJpaCrudApplication.**class**, args);

 }

}

EmployeeEntity.java

package com.ashokit.demo.entity;

import jakarta.persistence.Column;

import jakarta.persistence.Entity;

import jakarta.persistence.Id;

import jakarta.persistence.Table;

@Entity

@Table(name = "EMP")

public class EmployeeEntity {

 @Id

 @Column(name = "EMPNO")

 private Integer empNumber;

 @Column(name = "ENAME")

 private String empName;

 @Column(name = "SAL")

 private Double empSalary;

 public Integer getEmpNumber() {

 return empNumber;

 }

 public void setEmpNumber(Integer empNumber) {

 this.empNumber = empNumber;

 }

 public String getEmpName() {

 return empName;

 }

 public void setEmpName(String empName) {

 this.empName = empName;

 }

 public Double getEmpSalary() {

 return empSalary;

 }

 public void setEmpSalary(Double empSalary) {

 this.empSalary = empSalary;

 }

 @Override

 public String toString() {

 return "EmployeeEntity [empNumber=" + empNumber + ", empName=" + empName + ", empSalary=" + empSalary + "]";

 }

}

EmployeeEntityRepository.java

package com.ashokit.demo.repository;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

import com.ashokit.demo.entity.EmployeeEntity;

@Repository

public interface EmployeeEntityRepository extends CrudRepository<EmployeeEntity, Integer>{

}

MyApplicationRunner.java

**package** com.ashokit.demo.runner;

**import** java.util.Optional;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.ApplicationArguments;

**import** org.springframework.boot.ApplicationRunner;

**import** org.springframework.stereotype.Component;

**import** com.ashokit.demo.entity.EmployeeEntity;

**import** com.ashokit.demo.repository.EmployeeEntityRepository;

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeEntityRepository repo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 String lineSeparator = "=".repeat(40);

 // findById()

 Optional<EmployeeEntity> opt = repo.findById(7788);

 **if** ( opt.isPresent() ) {

 EmployeeEntity entity = opt.get();

 System.***out***.println(entity);

 }

 **else** {

 System.***out***.println("Employee with empno: 7788 doesn't exist!");

 }

 System.***out***.println(lineSeparator);

 //save()

 EmployeeEntity newEntity = **new** EmployeeEntity();

 newEntity.setEmpSalary(4599.0);

 newEntity.setEmpName("WILSON");;

 newEntity.setEmpNumber(7175);

 repo.save(newEntity);

 System.***out***.println("New employee persisted to the database with empno : " +newEntity.getEmpNumber());

 System.***out***.println(lineSeparator);

 //findAll()

 Iterable<EmployeeEntity> empList = repo.findAll();

 empList.forEach(System.***out***::println);

 System.***out***.println(lineSeparator);

 //deleteById()

 repo.deleteById(7175);

 System.***out***.println("Employee deleted from database with empno : 7175 ");

 System.***out***.println(lineSeparator);

 }

}



EmployeeEntityRepository.java

package com.ashokit.demo.repository;

import org.springframework.data.repository.CrudRepository;

import org.springframework.data.repository.PagingAndSortingRepository;

import org.springframework.stereotype.Repository;

import com.ashokit.demo.entity.EmployeeEntity;

@Repository

public interface EmployeeEntityRepository extends PagingAndSortingRepository<EmployeeEntity, Integer>{

}

MyApplicationRunner.java

**package** com.ashokit.demo.runner;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.boot.ApplicationArguments;

**import** org.springframework.boot.ApplicationRunner;

**import** org.springframework.data.domain.PageRequest;

**import** org.springframework.data.domain.Pageable;

**import** org.springframework.data.domain.Sort;

**import** org.springframework.stereotype.Component;

**import** com.ashokit.demo.entity.EmployeeEntity;

**import** com.ashokit.demo.repository.EmployeeEntityRepository;

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeEntityRepository repo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 String lineSeparator = "=".repeat(40);

 /\*

 \* pagination : is a process of getting a chunk of data from a large dataset.

 \* Pageable : an interface

 \* PageRequest: is an implementation class of Pageable interface

 \* PageRequest.of(0,3) : 0 indicates first page, and 3 is the number of records

 \*/

 //findAll(Pageable pageable)

 Pageable pageable = PageRequest.*of*(0, 3);

 Iterable<EmployeeEntity> empList = repo.findAll(pageable);

 empList.forEach(System.***out***::println);

 System.***out***.println(lineSeparator);

 /\*

 \* Sort: is a class and it has static factory methods like

 \* by(), ascending(), descending(), and(),....

 \*/

 //findAll(Sort sort)

 Iterable<EmployeeEntity> it = repo.findAll(Sort.*by*("empSalary").descending().and(Sort.*by*("empName")));

 it.forEach(System.***out***::println);

 }

}

application.properties

spring.application.name=Data-JPA-PaginationAndSorting

spring.main.banner-mode=off

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format\_sql=true

Query approaches in Data JPA:

------------------------------

* When we extend JpaRepository, our DAO layer will get data retrieval operations like findById() and findAll().
* Suppose, if we want more data trieval operations on our DAO layer then we have to add custom method declarations to our repository interface.
* To add custom method declarations, we have to use Query approaches.
* Query approaches are 3 types.
1. query creation by method name
2. using @Query annotation
3. using @NamedQuery annotation.

 query creation by method name:

* we add a method declaration to our repository interface.
* Spring Data Jpa, will automatically generates a query based on method name, and it executes that query to return the entities.
* The method name should follow a structure.

findBy or readBy or getBy + PropertyName + Condition(optional)

example1:

 List<Employee> findByEname(String ename);

 The above method generates a query like this;

 SELECT \* FROM EMP WHERE ENAME = ?

example2:

 List<Employee> findByFirstNameOrLastName(String firstName, String lastName);

 The above method generates a query like this;

 SELECT \* FROM EMP WHERE FIRST\_NAME=? OR LAST\_NAME=?

example3:

 List<Employee> findBySalaryGreaterThan(double sal);

 The above method generates a query like this;

 SELECT \* FROM EMP WHERE SAL > ?;

example4:

List<Employee> findByEnameContaining(String namePart);

 The above method generates a query like this;

 SELECT \* FROM EMP WHERE ENAME LIKE ‘%?%’;

example5:

 List<Employee> findTop5BySalaryGreaterThan(double sal);

 The above method generates a query like this;

 SELECT \* FROM EMP WHERE SAL > ? LIMIT 5;



EmployeeEntityRepository.java

**package** com.ashokit.demo.repository;

**import** java.util.List;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** com.ashokit.demo.entity.EmployeeEntity;

**public** **interface** EmployeeEntityRepository **extends** JpaRepository<EmployeeEntity, Integer> {

 List<EmployeeEntity> findByEmpSalaryGreaterThan(**double** sal);

 List<EmployeeEntity> findByEmpNameContaining(String part);

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeEntityRepository repo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 String lineSeparator = "=".repeat(40);

 //findById

 System.***out***.println("printing employee with id : 7788");

 System.***out***.println(lineSeparator);

 repo.findById(7788).ifPresent(System.***out***::println);

 System.***out***.println(lineSeparator);

 //findAll

 System.***out***.println("printing all employees");

 System.***out***.println(lineSeparator);

 repo.findAll().forEach(System.***out***::println);

 System.***out***.println(lineSeparator);

 //findByEmpSalaryGreaterThan

 System.***out***.println("printing employees with salary > 5000");

 System.***out***.println(lineSeparator);

 repo.findByEmpSalaryGreaterThan(5000.0).forEach(System.***out***::println);

 System.***out***.println(lineSeparator);

 //findByEmpNameContaining

 System.***out***.println("printing employees with name contains 'A' ");

 System.***out***.println(lineSeparator);

 repo.findByEmpNameContaining("A").forEach(System.***out***::println);

 }

}

@Query annotation:

* This annotation in Spring Data JPA allows developers to define customer queries using JPQL or in SQL.
* JPQL – Java Persistence Query Language
* SQL – Structured Query Language
* JPQL queries looks like SQL queries. But in place of column names, we have to write property names and in place of table name, we have to write entity class.
* If we write a JPQL query, it will be translated to a tuned SQL query and then that query will run on the database.
* JPQL queries are database indendent and they improve the application performance.

example1:

 sql : SELECT \* FROM EMP

 jpql: SELECT e FROM EmployeeEntity e

example2:

 sql : SELECT EMPNO, SAL FROM EMP

 jpql: SELECT e.empNumber, e.empSal FROM EmployeeEntity e

example3:

 sql : SELECT \* FROM EMP WHERE SAL > ?

 jpql: SELECT e FROM EmployeeEntity e WHERE

 e.empSal > ?1

 (or)

 SELECT e FROM EmployeeEntity e WHERE

 e.empSal > :salary

 example4:

 sql : SELECT \* FROM EMP WHERE ENAME LIKE ‘%s%’ AND

 SAL > ?

 jpql: SELECT e FROM EmployeeEntity e WHERE

 e.empName LIKE ‘%s%’ AND e.empSal > ?1

 ==================================================



EmployeeEntityRepository.java

**public** **interface** EmployeeEntityRepository **extends** JpaRepository<EmployeeEntity, Integer> {

 @Query( "SELECT e FROM EmployeeEntity e WHERE e.empSalary >= ?1")

 List<EmployeeEntity> fetchEmpsWithSal(**double** sal);

 @Query( nativeQuery = **true**, value = "SELECT \* FROM EMP WHERE DEPTNO = ?")

 List<EmployeeEntity> getEmpsByDept(**int** deptno);

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeEntityRepository repo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 String lineSeparator = "=".repeat(40);

 //fetchEmpsWithSal

 repo.fetchEmpsWithSal(6000.0).forEach(System.***out***::println);

 System.***out***.println(lineSeparator);

 //getEmpsByDept

 repo.getEmpsByDept(30).forEach(System.***out***::println);

 }

}

@NamedQuery:

 This annotation can be used to define a named query that can be reused across mutliple places in our application.

 we define a named query at entity level and it is referred in the repository, with its name.

 Named query improves the query readability as the query and entity class used in the query are at one place.

 Without rewriting a query, it can be used at different parts of the application.

 Named queries are precompiled by the JPA provider. So, it improves performance.

 Like @NamedQuery, we also have @NamedNativeQuery to configure native SQL query at entity level.



EmployeeEntity.java

@Entity

@Table(name = "EMP")

@NamedQuery( name = "EmployeeEntity.findByDepartment",

 query = "SELECT e FROM EmployeeEntity e WHERE e.deptNumber = ?1")

@NamedQuery( name = "EmployeeEntity.findBySalGreaterThan",

 query = "SELECT e FROM EmployeeEntity e WHERE e.empSalary > ?1")

@NamedNativeQuery ( name = "EmployeeEntity.retrieveByEnameLike",

 query = "SELECT \* FROM EMP WHERE ENAME LIKE ?",

 resultClass = EmployeeEntity.**class**)

**public** **class** EmployeeEntity {

 @Id

 @Column(name = "EMPNO")

 **private** Integer empNumber;

 @Column(name = "ENAME")

 **private** String empName;

 @Column(name = "SAL")

 **private** Double empSalary;

 @Column(name = "DEPTNO")

 **private** Integer deptNumber;

 **public** Integer getDeptNumber() {

 **return** deptNumber;

 }

 **public** **void** setDeptNumber(Integer deptNumber) {

 **this**.deptNumber = deptNumber;

 }

 **public** Integer getEmpNumber() {

 **return** empNumber;

 }

 **public** **void** setEmpNumber(Integer empNumber) {

 **this**.empNumber = empNumber;

 }

 **public** String getEmpName() {

 **return** empName;

 }

 **public** **void** setEmpName(String empName) {

 **this**.empName = empName;

 }

 **public** Double getEmpSalary() {

 **return** empSalary;

 }

 **public** **void** setEmpSalary(Double empSalary) {

 **this**.empSalary = empSalary;

 }

 @Override

 **public** String toString() {

 **return** "EmployeeEntity [empNumber=" + empNumber + ", empName=" + empName + ", empSalary=" + empSalary

 + ", deptNumber=" + deptNumber + "]";

 }

}

EmployeeEntityRepository.java

**public** **interface** EmployeeEntityRepository **extends** JpaRepository<EmployeeEntity, Integer> {

 //No need to define the query again.

 //Just use the name of the @NamedQuery

 List<EmployeeEntity> findByDepartment(**int** deptno);

 List<EmployeeEntity> findBySalGreaterThan(**double** sal);

 List<EmployeeEntity> retrieveByEnameLike(String part);

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeEntityRepository repo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 repo.findByDepartment(30).forEach(System.***out***::println);

 System.***out***.println("=================================");

 repo.findBySalGreaterThan(6000.0).forEach(System.***out***::println);

 System.***out***.println("================================");

 repo.retrieveByEnameLike("%S%").forEach(System.***out***::println);

 }

}

 Entity relationships

1. one to many
2. many to one
3. one to one
4. many to many
* Defining the relationships between the entities reduces duplication of data and saves storage space and we can organize the data in a meaningful way.

One-to-Many:

* one entity can be related to many other entities.
* The entity on the one side can be associated with many entities on the many side.
* For ex, one Department can have many Employees.
* Another example, one Customer can place many Orders.
* To define any relationship, first we need to identify the source and the target entities.
* Now, we can define a relationship from the source to the target entity.
* For example, to define one to many relationship between Department and Employees, the source is the Department entity and the target is the Employee entity.
* To provide one-to-many relationship, in the source entity class, we need to declare a Collection reference. This is a one-to-many field.



* If we define one to many relationship, the operations at one side are cascaded to the many side.
* In the above example, we can save a department with its employees, or we can delete a department with its employees, or we can fetch a department with its employees.

Lombok api:

* It is a popular java library used to reduce boiler plate code by providing annotations.
* It avoids writing getter and setters, constructors, toString, hashCode and equals methods.
* To use Lombok, first we need to add the dependency to the pom.xml file.

<dependency>

 <groupId>org.projectlombok</groupId>

 <artifactId>lombok</artifactId>

 <version>1.18.34</version>

</dependency>

* annotations in lombok.

 @Getter: generates getter methods for each field.

 @Setter: generates setter method for each field

 @ToString: generates toString() method.

 @EqualsAndHashCode: generates equals() and hashCode() methods.

 @NoArgsConstructor : generates no-args constructor

 @AllArgsConstructor: generates all args constructor

 @Data: A convenient annotation that bundles

 @Getter, @Setter, @ToString and @EqualsAndHashCode

configuring lombok for IDE:

* first add the dependency in pom.xml
* Goto lombok jar file stored in local repository of maven.

(C:\Users\WINDOWS\.m2\repository\org\projectlombok\lombok\1.18.34)

* type cmd in the location bar
* in the command prompt, run the jar file.

java -jar lombok-1.18.34.jar

* A window is opened, click on specify location and choose SpringToolSuite4.exe, then click on install button. Finally click on finish button.
* Now restart the STS ide.



DepartmentEntity.java

@Entity

@Table(name = "DEPT")

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** DepartmentEntity {

 @Id

 **private** Integer deptno;

 **private** String dname;

 **private** String loc;

 /\*

 \* cascade attribute tells about what operations

 \* should be propagated to the child entities from the

 \* parent entity.

 \*

 \* fetch attribute tells about with parent, its childs

 \* should be loaded from the database or not.

 \*

 \* In one-to-many, the default value of fetch is FetchType.LAZY

 \*/

 @OneToMany(cascade = CascadeType.***ALL***, fetch = FetchType.***EAGER***)

 @JoinColumn( name = "deptno" )

 **private** List<EmployeeEntity> empList;

}

EmployeeEntity.java

@Entity

@Table(name = "EMP")

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** EmployeeEntity {

 @Id

 **private** Integer empno;

 **private** String ename;

 **private** Double sal;

}

DepartmentRepository.java

**public** **interface** DepartmentRepository **extends** JpaRepository<DepartmentEntity, Integer> {

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 DepartmentRepository deptRepo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 /\*

 DepartmentEntity deptEntity = new DepartmentEntity();

 EmployeeEntity e1 = new EmployeeEntity(7080, "Blake", 4000.0);

 EmployeeEntity e2 = new EmployeeEntity(7344, "Allen", 7000.0);

 EmployeeEntity e3 = new EmployeeEntity(7910, "David", 3000.0);

 deptEntity.setDeptno(20);

 deptEntity.setDname("RESEARCH");

 deptEntity.setLoc("NEW YORK");

 deptEntity.setEmpList(Arrays.asList(e1, e2, e3));

 deptRepo.save(deptEntity);

 \*/

 System.***out***.println("===================================");

 Optional<DepartmentEntity> opt = deptRepo.findById(10);

 **if**(opt.isPresent() ) {

 DepartmentEntity dept = opt.get();

 System.***out***.println(dept.getDeptno() + " " + dept.getDname() + " " + dept.getLoc());

 }

 }

}

Many-to-One:

* Many entities of “many” side are related to one entity at “one” side.
* Many entities of many side are linked to single entity at one side.
* It is same as one to many, but operations are navigated from child entities to parent entity.
* To define many-to-one relationship, create parent class reference in the child class.
* For example, create Department class reference in the Employee class.



DepartmentEntity.java

@Entity

@Table(name = "DEPT")

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** DepartmentEntity {

 @Id

 **private** Integer deptno;

 **private** String dname;

 **private** String loc;

}

EmployeeEntity.java

@Entity

@Table(name = "EMP")

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** EmployeeEntity {

 @Id

 **private** Integer empno;

 **private** String ename;

 **private** Double sal;

 @ManyToOne(cascade = CascadeType.***ALL***, fetch = FetchType.***LAZY***)

 @JoinColumn( name = "deptno" )

 DepartmentEntity department;

}

EmployeeRepository.java

**public** **interface** EmployeeRepository **extends** JpaRepository<EmployeeEntity, Integer> {

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 EmployeeRepository empRepo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 /\*

 EmployeeEntity empEntity = new EmployeeEntity();

 empEntity.setEmpno(7767);

 empEntity.setEname("Kathey");

 empEntity.setSal(4000.0);

 DepartmentEntity deptEntity = new DepartmentEntity(30, "Forest", "New Jersey");

 empEntity.setDepartment(deptEntity);

 empRepo.save(empEntity);

 \*/

 /\*

 empRepo.deleteById(7767);

 \*/

 empRepo.findById(7344);

 }

}

Many-to-Many relationship:

* Many-to-Many is a one-to-many with inverse one-to-many relationship.
* For ex, one Book is linked to Many Authors and one Author is linked to Many Books.
* Here, the relationship is one-to-many from Book to Author and again one-to-many from Author to Book. So, it is called Many-to-Many.
* A minimum of 3 tables is required in the database to execute many-to-many relationship.
* Two tables are mapped to the entity classes and a join table is also required to store the many-to-many relationship.
* This join table stores the foreign keys of the two tables.



Book.java

@Entity

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** Book {

 @Id

 **private** Integer bid;

 **private** String name;

 @ManyToMany(cascade = CascadeType.***ALL***)

 @JoinTable( name = "BOOK\_AUTHORS",

 joinColumns = @JoinColumn(name = "BID\_FK"),

 inverseJoinColumns = @JoinColumn(name = "AID\_FK")

 )

 **private** List<Author> authorsList;

}

Author.java

@Entity

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** Author {

 @Id

 **private** Integer aid;

 **private** String name;

}

BookRepository.java

**public** **interface** BookRepository **extends** JpaRepository<Book, Integer> {

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

 @Autowired

 BookRepository bookRepo;

 @Override

 **public** **void** run(ApplicationArguments args) **throws** Exception {

 Author a1 = **new** Author(1, "author1");

 Author a2 = **new** Author(2, "author2");

 Author a3 = **new** Author(3, "author3");

 Book b1 = **new** Book();

 b1.setBid(101); b1.setName("Book1");

 Book b2 = **new** Book();

 b2.setBid(102); b2.setName("Book2");

 List<Author> authorsListForBook1 = Arrays.*asList*(a1, a2);

 List<Author> authorsListForBook2 = Arrays.*asList*(a2, a3);

 b1.setAuthorsList(authorsListForBook1);

 b2.setAuthorsList(authorsListForBook2);

 bookRepo.save(b1);

 bookRepo.save(b2);

 }

}



Person.java

@Entity

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** Person {

 @Id

 **private** Integer id;

 **private** String name;

 @OneToOne(cascade = CascadeType.***ALL***, fetch = FetchType.***LAZY***)

 @JoinColumn(name="passport\_id")

 Passport passport;

}

Passport.java

@Entity

@NoArgsConstructor

@AllArgsConstructor

@Data

**public** **class** Passport {

 @Id

 **private** Integer id;

 **private** LocalDate expireDate;

}

PersonRepository.java

**public** **interface** PersonRepository **extends** JpaRepository<Person, Integer> {

}

MyCommandLineRunner.java

@Component

**public** **class** MyCommandLineRunner **implements** CommandLineRunner {

 @Autowired

 PersonRepository repo;

 @Transactional

 @Override

 **public** **void** run(String... args) **throws** Exception {

 /\*

 Person person = new Person();

 person.setId(101); person.setName("John");

 Passport passport = new Passport(509231, LocalDate.of(2024, 12, 31));

 person.setPassport(passport);

 repo.save(person);

 \*/

 /\*

 Optional<Person> opt = repo.findById(101);

 opt.ifPresent(System.out::println);

 \*/

 repo.deleteById(101);

 }

}