Spring Data JPA

Contents

ORM Basics

ORM

- ORM stands for object relational mapping it is the process of mapping a Java class to a database table and its fields or members to the database table columns.
- Once we do that mapping we can synchronized our class objects into database rows.
- We as Object Oriented developers will deal with objects instead of writing sql and invoke methods like SAVE Update Delete on the objects and automatically the sql statements will be generated for us.
- And using JDBC internally these ORM Tools will do the insertion updation deletion etc. so we no longer need to deal with sql directly and also we no longer have to deal with low level API's like JDBC.

JPA

- JPA stands for Java Persistence API and it is a **standard** from Oracle to perform object relational mapping in Java EE applications.
- Like any other standard from Oracle JPA it comes with **specification** and an **API**. The **specification** is for the JPA vendors or the providers. The **API** is for us the developers or the programmers to master it.
- There are several popular JPA **providers like Hibernate, OpenJPA, Eclipse Link,** etc. which implement the JPA API by following the specification.
- The **specification** is nothing but a set of rules written in plain English so that these vendors can implement the API easily and consistently.
- **Before JPA** came into existence, we as developers had to learn Hibernate API, Open JPA API or Eclipse link API, depending on which of these ORM tools we are using for our application. But now we learn one single API (JPA) and all these vendors/providers implement this API and we can switch from one vendor to another without making any changes to our application. That is the power of JPA.
- **Hibernate** is the most popular JPA provider and used in most of the Java EE applications.
- The EntityManagerFactory and EntityManager are specific to JPA. Spring Data will even hide this from us. We need not deal with this.

What and Why Spring Data

- Developers at spring have a habit of making things easier for us and remove a lot of boiler plate coding that we usually do spring data is one such framework which removes a lot of configuration and coding hassle when we deal with data access layer for about applications.
- Two Simple Steps
    - All you need to do as a developer to come up with your data access layer is define your domain object or JPA entity with all the fields on it and map it to the database table using JPA annotations.
    - Once you do it you need not write any DAO classes or interfaces, simply define one single interface. It is not even a class, this interface that we create will extend a `CRUDRepository` interface from spring that is all that is required.
    - Once we do that at runtime we'll be able to access this employee repository in our services layer or in your presentation layer or wherever we want to perform database operations. Spring will dynamically implement a class for us at runtime out of the repository interface that we created.
- Internally spring data will use JPA and the power of ORM tools like Hibernate and it will use entity managers, session, session factory, etc. All that will happen auto magically behind the scenes. But in our application will have only two classes the entity classes and then the repository interfaces and will get everything for free at runtime.
- Spring data offers **finder methods** – so without writing any sql code, we can load data from the database by following simple method convention that it gives us.
- It makes it a lot easier to do **paging and sorting**.
- We can also use **JPQL** (JPA Query Language) and even **Native SQL Queries**.
- Whatever we can do in JPA while using Hibernate, spring data will make it a lot simpler for the developers and it offers several other features.

Simple CRUD Operations

Entity

- @Entity – This is mandatory. It indicates that the class is a JPA Entity.
- @Table – This is optional. We need this only if the database table name is different from the class name.
- If you have camelcase entity name, JPA will use underscores for corr. database table name. e.g. ClinicalData entity class will map to clinical_data database table.
- @Id – to annotate a field which is a primary key in database table.
- @Column – is optional. We need not mark each of these fields with @Column. We only need it if the database column name is different from our field name.

Repository

- Create any repository interface that will extend the CRUDRepository interface from spring data.
- The CRUDRpository expects 2 parameters in the form of Generics. One – for which entity we are creating this repository. And other is – what is the type of @Id field in that entity class.

Configure Data Source

- Now we need to configure the data source information that is the JDBC connection URL, database user name and password.
- Thanks to spring boot, we can configure the data source information in the /src/main/resources/application.properties file.
- Some of the properties are
  spring.datasource.url – JDBC URL of the database.
  spring.datasource.username – Login username of the database.
  spring.datasource.password – Login password of the database.
- That's all! With this information, the database connection will be created when we run our application.
- For all other properties, refer – https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html#common-application-properties-data

How Spring Boot Does Behind The Scenes

- Based on project classpath, Spring boots gets necessary information to create Data Source (using spring data jps jar, hibernate jar, mysql or any other DB driver connector jar, application properties file for configuration).
- With the relevant data source information available, Spring Boot will create EntityManagerFactory in background. And then it will create the EntityManager instance.
- And when we execute any method (save, delete, update, etc.) our custom repository which is an interface, spring internally creates an implementation or a proxy implementation of that repository interface. That on-the-fly generated implementation class will in turn invoke the methods on the EntityManager because entity manager is the one which in the end talks to database.
- So we have avoided all the boiler plate code and configuration by simply using spring data.

ID Generators

ID Generation Strategies

- JPA provides or support four different types of ID generation strategies –
    - Auto – GenerationType.AUTO
    - Identity – GenerationType.IDENTITY
    - Sequence – GenerationType.SEQUENCE
    - Table – GenerationType.TABLE
- We can mark our @Id field in our entity class using @GeneratedValue annotation.
  E.g. @GeneratedValue(strategy = GenerationType.AUTO)
- In addition to above 4 strategies, there is one more way which is "Using custom ID generation".

Auto

- Auto means the persistence providers like Hibernate will have to check with the underlying database which kind of ID generations strategies (identity or sequence or table) does it support.
- Accordingly it will use one of these strategies which is supported by the underlying database to generate our primary key value. That is why it is called AUTO automatic.

Identity

- Here the persistence provider (e.g. Hibernate) will rely on the auto increment field.
- When we create a primary key in the database, will have to also configure it to be an auto increment field (AUTO_INCREMENT). So the persistence providers like Hibernate will use the incremented value every time when a record is inserted.

Sequence

- In this case, you can define a custom logic to generate a value and you can then use that value as an ID.
- So the persistence provider will use a database sequence.
- When you configure your generation type as sequence, you also need to tell hibernate which sequence in the database it should use. Hibernate will run that sequence, take the results from that and use it as a primary key value when your record is saved.
- Databases like mysql do not support sequences.

Table

- Here will be using a special table. And our persistence provider will use this special table.
- It will generate a value, put that value into this special table in a column and it will also use that value as a primary key to our (entity) table.
- The next time we are saving a record, it will look at the previous value in that special table, generate the next value and use that as a primary key for that particular record. That is the reason it is called Generation type table.
- We need to create this special table along with the entities table.

Steps

- Create the special table in database. E.g.

```
create table id_gen(
      gen_name varchar(60) PRIMARY KEY,
      gen_val int(20)
)
```

- Annotate your entity class's @Id field with one more annotation @TableGenerator and pass below info –
  - `name` – Since we have only one ID generation table, we need to use this 'name' attribute to differentiate sequences for different entities (e.g. Employee)
  - `table` – The special ID generation table name.
  - `pkColumnName` – primark key column name of the special generation table,
  - `valueColumnName` – the column of the special generation table which stores the value of the sequence.
  - `allocationSize` – specifies by how much the sequence should be incremented.
  - e.g. `@TableGenerator(name = "employee_gen", table = "id_gen", pkColumnName = "gen_name", valueColumnName="gen_val", allocationSize = 1)`
- Finally, mark the @Id field with table type generation strategy and also provide generator as –

  `@GeneratedValue(strategy = GenerationType.TABLE, generator = "employee_gen")`

  The `generator` attribute here is set equal to what is defined at `@TableGenerator` annotation.

Custom ID Generation

- In addition to above 4 strategies, there is one more way which is "Using custom ID generation".
- Create a class which implements Hibernate's `IdentityGenerator` interface and override the only one method called generate().
- One of the parameters to this generate() method is the Entity object and this method should return generated ID. So inside this method, we should implement our custom ID generation logic.
- Now for your entity class, make these changes
    - Add annotation @GenericGenerator to the @Id field of the entity class like this – `@GenericGenerator(name = "emp_id", strategy = "com.sameer.generator.CustomIdGenerator")`
      So for this strategy attribute, pass the name of your custom ID generation class.
    - Also add another annotation @GeneratedValue like this – `@GeneratedValue(generator = "emp_id")`

Spring Data Finder Methods

Introduction

- Finder methods is a very powerful feature that spring data offers where without writing any code or even a SQL query, just by following some naming conventions, we can load data from the database.
- Refer this link to see naming conventions for different ways of fetching data. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation

Paging and Sorting

Introduction

- To enable paging, instead of extending the `CRUDRepository`, our repository interface should extend `PagingAndSortingRepository` which is a child interface of the `CRUDRepository`.
- Once you do this, you will get paging and sorting on the `findAll` method inside your repository and also it is easy to implement paging and sorting on your custom finder methods (e.g. `findByName()`, etc.).
- To implement paging and sorting on your custom finder methods, simply add an additional parameter called `Pagable` as last parameter. E.g. `findByName(String name, Pageable pg).` Now you can pass this method a `PageRequest` object and depending on how you want the page and sort, it will work for you automatically.
- Note – You need not extend `PagingAndSortingRepository` if you want to. If you want your `findAll()` method to support paging and sorting that is when you extend `PagingAndSortingRepository`. Otherwise you can extend `CRUDRepository` and you can pass an additional parameter called `Pagable` to your custom finder methods and that method will then start supporting all the paging and sorting logic.
- The key interfaces when you do **paging**, you use `Pagable` and `PageRequest` is an implementation of that interface.
- So once we enable paging, to use paging in our services layer or any other layers of our application where we are retrieving the database record using finder methods, we use `Pagable` and `PageRequest.`
- For **sorting,** the key classes are `Sort`, `Direction` and `Order` from spring data`. Order` represents a direction and property.
- So all of these are not a part of JPA. They are part of spring data which makes our life a lot easier to implement paging and sorting.
- You can only Paging or only Sorting or both Paging and Sorting depending on your requirement.

JPQL

Introduction

- JPQL stands for a Java Persistence Query Language.
- It is a standard from JPA to perform queries against objects and domain classes. So instead of writing SQL queries against database tables and columns, we write JP queries against our objects and their fields.
- These JP queries are internally converted to native SQL queries by the ORM tools like Hibernate.
- Most of what is there in sql is also provided in JPQL keywords and syntax.
- JPQL is **case sensitive** when it comes to the domain class names and its field names but it is **not case sensitive** to when it comes to keywords in the language syntax itself like 'select', 'count', etc.

Using JPQL

- Mark the method in your repository interface with @Query annotation. And pass your JP query to it. You can also use alias for your entity names in the JPQ like native queries.
- E.g.
  Assuming Entity class name as Student and underlying table name as student_t.
  ```
  @Query("select * from Student")
  List<Student> findAllStudents();

  @Query("from Student")
  List<Student> findAllStudents();
  ```
- Note – @Query takes one optional boolean parameter called `nativeQuery`. If set to true, you can pass native SQL query. Otherwise it is treated as JPQ.
  E.g.
  ```
  @Query(value = "select * from student_t", nativeQuery = true)
  List<Student> findAllStudentNQ();
  ```
- Note –
  - If you are fetching all columns, then writing 'select *' is optional.
  - If you are fetching only specific columns, the return type of your method should be List<Object[]>. And each element in the list is an array of Object type. So to access each columns, use indexes.
    E.g.
    ```
    @Query("select firstName, lastName from Student")
    List<Object[]> findAllStudentsPartialData();
    ```
    When you loop over this returned List, you can access firstName as obj[0], lastName as obj[1].

Using Named Query Parameters

- Using named query parameters, we can dynamically pass in parameters to the repository function and then set those into the query mentioned in @Query.
- To bind the correct method argument to the named parameter in the JPQ, we need to use @Param annotation.
- E.g. 1

```
@Query("from Student where firstName=:firstName")
List<Student> findAllStudentsByFirstName(@Param("firstName") String
firstName);
```

- E.g. 2

```
@Query("from Student where score>:min and score<:max")
List<Student> findStudentsForGivenScores(@Param("min") int min,
@Param("max") int max);
```

Non Select (DML) Operations

- JPQL not only supports select operations but also DML statements like insert, update and delete.
- It is almost the same as writing select/find queries. i.e. it also uses the same `@Query`, `@Param` annocatations.
- In addition to this, we also need to annotate the method with **`@Modifying`** annotation. The reason being spring data by default assumes that all queries in your repository are read queries. So with `@Modifying`, we tell Soring data that this query is going to modify data (insert, update or delete).
- E.g.

```
@Modifying
@Query("delete from Student where firstName = :firstName")
void deleteStudentsByFirstName(@Param("firstName") String firstName);
```

- So if you try to perform a non-select operation without using `@Modifying` annotation, you will get an exception.
- Side Note –
    - If you are writing a test, you need to mark that test as **`@Transactional`** since we are performing a DML operation.
    - A Junit test will rollback the transaction even if you annotate it with `@Transactional`. This is the default behavior for Tests.
    - To avoid this default behavior of rollback in Tests, we need to annotate the test method with **`@Rollback(false)`** annotation. Use it with caution.

Paging and Sorting with JQPL

- Using paging and sorting with JPQL is no different.
  It is described here – Paging and Sorting
- E.g.

```
@Query("from Student")
List<Student> findAllStudents(Pageable pageable);

@Test
public void testFindAllStudents() {
      System.out.println(repository.findAllStudents(new PageRequest(0,
5, Direction.DESC, "id")));
}
```

Native SQL Query

Introduction

- Native SQL queries means the queries which database directly understands.
- When our JPQL query is **complex** involving multiple joins, etc. or if we have a **database expert** who is really good at writing Native SQL queries, then we go for Native SQL queries.
- In addition to the DML and DQL, native queries also support DDL queries. DDL queries are NOT supported in JPQL.
- Link JPQL, native queries also support named parameters.
- If you can't do something using JPQL or if it is getting too complex, use Native SQL queries as the backup.

Using Native Queries

- Mark the method in your repository interface with `@Query` annotation, pass your native query to its `value` attribute and also set **`nativeQuery`** parameter to **true**. If `nativeQuery` is not set to true, Spring data will assume it as a JPQL.
- You can also use alias for your native query table names.
- E.g.
  Assuming Entity class name as Student and underlying table name as student_t.
  ```
  @Query(value = "select * from student_t", nativeQuery = true)
  List<Student> findAllStudentNQ();
  ```
- E.g. Native Query **with Named Parameters**
  ```
  @Query(value = "select * from student where fname=:firstName",
  nativeQuery = true)
  List<Student> findByFirstNQ(@Param("firstName")String firstName);
  ```

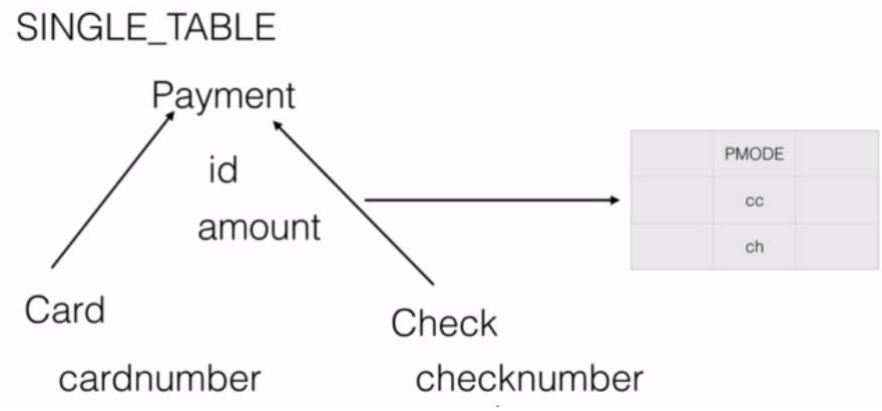
Inheritance Mapping

Introduction

- We use inheritance in Java classes to reuse common fields/methods. So it is common that we use inheritance for Entity classes as well, however the underlying databases do not support inheritance mapping across the database tables. This is known as **sub type problem** in ORM.
- To solve this problem JPA provides inheritance mapping through three types of strategies – SINGLE_TABLE, TABLE_PER_CLASS, JOINED.

JPA Inheritance Strategies

SINGLE_TABLE Strategy

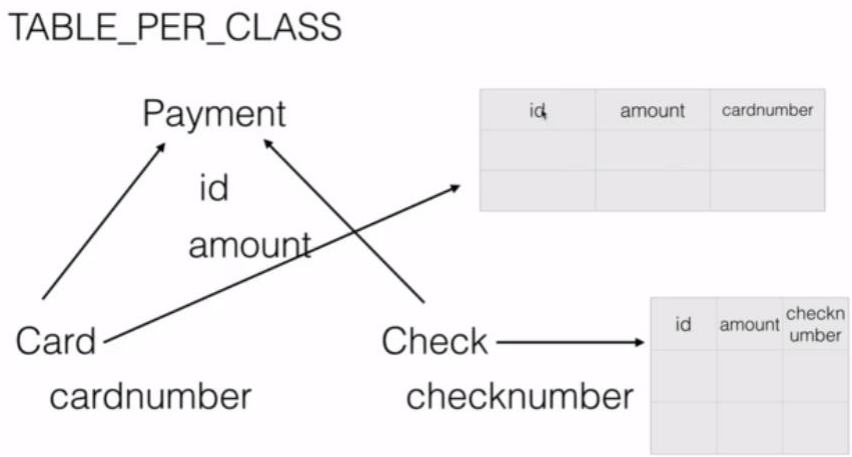


- As the name itself says in this case all the information whether it's card or cheque will go into one single table.
- And since it is going into one single table we need **extra discriminator column** (e.g. pmode in this example) that will differentiate between a card and a cheque.
- So we will provide values for that discriminator column representing a card or cheque so that hibernate will use those values in database to appropriately map both the Entity classes.
- To achieve this, we need to these annotations –
  - On parent class – `@Inheritance, @DiscriminatorColumn`
  - On child class – `@DiscriminatorValue`
- Example –

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = “pmode”, discriminatorType=
DiscriminatorType.STRING) // pmode column is from database table
public abstract class Payment {
      @Id
```

```
        private int id;
        // other common fields apart from the discriminator column
}

// child entity
@Entity
@DiscriminatorValue("cc")
public class CreditCard extends Payment{

        // some fields other than ID column
}

// child entity
@Entity
@DiscriminatorValue("ch")
public class Check extends Payment{

        // some fields other than ID column
}
```

TABLE_PER_CLASS Strategy

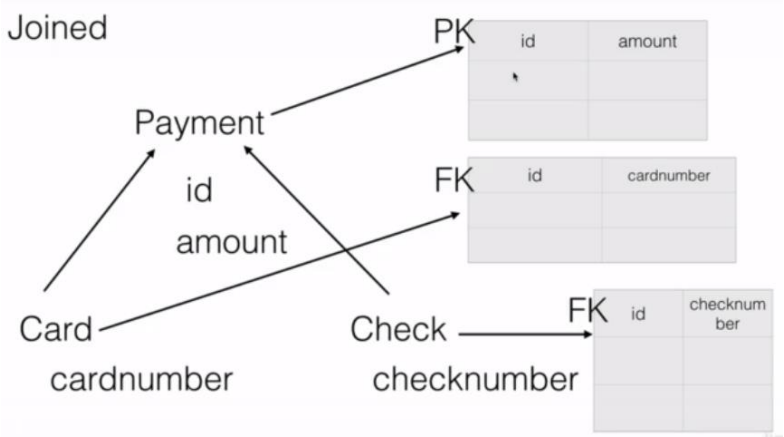


- Here we have **separate tables for child classes** (No table for parent as it is not required).
- So in our example, we will have a table for card as well as a table for check.
- Due to this, the tables of the subclasses will maintain **duplicate columns**.
- Although this approach improves performance, it is not recommended as it duplicates the columns across tables failing the normalization rules.

- To achieve this, we need `@Inheritance` annotation.
- Example –

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Payment {
      @Id
      private int id;
      // other fields and getters and setters
}

// child entity
@Entity
@Table(name="credit_card")
public class CreditCard extends Payment{
      // some fields other than ID column
}

// child entity
@Entity
@Table(name="bank_cheque")
public class Check extends Payment{
      // some fields other than ID column
}
```

JOINED Strategy



- This is the most popular and most used inheritance mapping strategies.
- Here every class in the inheritance hierarchy will have its own database table. The **parent** class will carry the **common fields** across the child classes and each **child** class will have table which will carry the field which is **specific** to that child. And the parent table is connected to each of the children through the primary key and **foreign key** relationship.

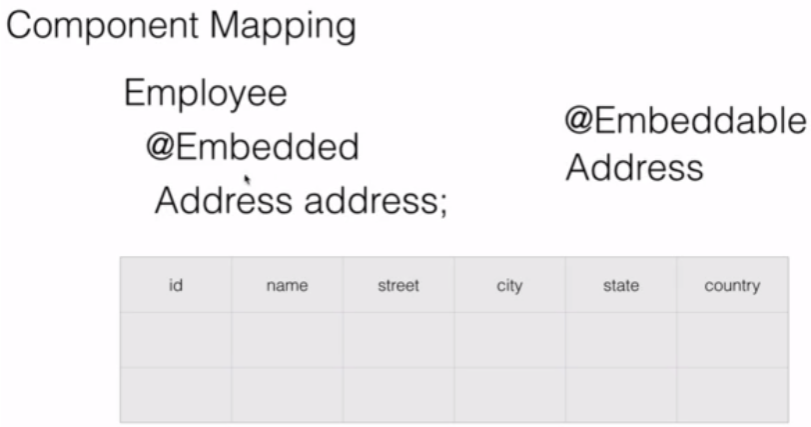
- The advantages is that it is one of the best inheritance mapping strategies which each table storing limited data. It follows normalization. That is the reason each table carries minimal data.
- If there is any disadvantage that is hibernate will have to join these tables to retrieve that data and it loads the data back. But that is expected in enterprise applications.
- To achieve this, we need `@Inheritance` and `@PrimaryKeyJoinColumn` annotations.
- Example –

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Payment {

	@Id
	private int id;
	// other fields
}

// child table

@Entity
@Table(name="card")
@PrimaryKeyJoinColumn(name="id")
public class CreditCard extends Payment{

	// child specific fields
}

// child table

@Entity
@Table(name="bankcheck")
@PrimaryKeyJoinColumn(name="id")
public class Check extends Payment{

	// child specific fields
}
```

Component Mapping

Introduction

- We use component mapping when we want to save one class that has a "HAS A" relationship with another class in the same database table.



- Here we mark the Address class with `@Embeddable` annotation from JPA. This tells the JPA implementations like Hibernate that this class is not an entity of its own but it will be embedded in another class.
- Also the Address field inside Employee class will be marked with `@Embedded` annotation. So hibernate will know that when an Employee is saved the embedded class object (Address) should also be saved into the same table.
- Example –

```
@Entity
public class Employee {
	@Id
	private int id;
	private String name;
	@Embedded
	private Address address;
	// getter and setters
}
@Embeddable
public class Address {
	private String street;
	private String city;
	private String state;
	private String zipcode;
	private String country;
	// getter and setters
}
```

Relationships in Hibernate

Introduction

- We usually normalize our database tables into separate tables to hold the appropriate data like order, customer, product, address and each of these tables will have relationships with other tables usually through primary keys and foreign keys.
- That is where **Association mapping** comes in – to map these database tables to our domain classes or JPA entities along with their associations.
- We need JPA Association mapping using that once the JPA entities are marked with these associations, we can navigate from one entity to another. Behind the scenes, we will have all the SQL queries, joins etc. generated by hibernate or any other JPA implementation.
- JPA support for types of relationships or associations – One to One, One to Many, Many to One, Many to Many.
- Also these relationships can occur in two **modes** – **unidirectional or bidirectional**.
- **Unidirectional** meaning we'll be able to access that relationship in only one order. For example, with customer to phone number (one to many mapping) case, if we configure one to many only from customers to phone number will not be able to access the customer using the phone number entity. But we can access the phone number from the customer. That is unidirectional. We can only configure it on either side of the relationship.
- But if it is **bidirectional** that means we can access either of the objects or entities from either of them. So we can navigate from customer to the phone number and phone number to the customer.
- So we can do all that association, unidirectional, bidirectional configuration using the four annotations provided by JPA and several of its properties – @OneToOne, @OneToMany, @ManyToOne, @ManyToMany.

One To Many AND Many To One Association

- Here an object of one entity is associated to many or multiple objects of another entity.
- Using @OneToMany, we need to pass an attribute called mappedBy which tells who owns the mapping i.e. it tells JPA tools like Hibernate to check the other side of the relationship for the exact information on how to do this join or how to make this many to many association work.
- Use @ManyToOne annotation, on the field of parent entity type.
- Also use @JoinColumn which tells hibernate which column in the child table should be used to associate it with that particular parent entity.

Cascading

- Cascading is the process of propagating the non-select operations like insert update delete from the main object in the association to the Child object.
- Cascading means when parent is saved or deleted, child record should also be saved or deleted automatically.
- We can control how the propagation happens and at what level and on what operations using the `cascade` element on the `@OneToMany, @ManyToOne` annotations.
- There are different **allowed values for the cascade** attribute –
    - cascade="**persist**" means an insert operation on the main object should be propagated to the child object.
    - cascade="**merge**" means an insert or an update operation on the main object should be propagated to the child object.
    - cascade="**remove**" means you delete the main object automatically the child objects should also be deleted.
    - cascade="**refresh**" means if you manually refresh a main object using the underlying JPA entity manager, automatically the child objects will be refreshed.
    - cascade="**detach**" means if you manually detach a main object using the underlying JPA entity manager, automatically the child objects will be detached.
    - cascade="**all**" means it supports all above operations. So when a main object is saved updated removed refreshed or detached that objects will also be affected.

Lazy Loading

- When objects are associated or related with other objects and when the parent object is loaded, the child object can be loaded immediately which is known as **eager loading** or they can be fetched on an on demand basis which is known as **lazy loading**.
- Lazy loading or On-Demand fetching improves the performance of the application because we need not load everything unless it is required.
- So in case of lazy loading, child object will not be loaded when parent is loaded. The child object is loaded only when we specifically invokes method to get the child object. E.g. calling customer.getPhoneNumbers() will load PhoneNumbers rows for given customer.
- To enable lazy loading, we need to use attribute `fetch` of the `@OneToMany, @ManyToOne` annotations. Allowed values for this attribute are `FetchType.EAGER, FetchType.LAZY`.
- By default in one to many associations the child information is loaded lazily.
- For lazy loading to work, make sure you mark your methods as `@Transactional`.

One To Many AND Many To One Example

```
create table customer(
id int PRIMARY KEY AUTO_INCREMENT,
name varchar(20)
);

create table phone_number(
id int PIRMARY KEY AUTO_INCREMENT,
customer_id int,
number varchar(20),
type varchar(20),
 FOREIGN KEY (customer_id)
REFERENCES customer(id)
)
// parent

@Entity
public class Customer {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private long id;
 private String name;
 @OneToMany(mappedBy = "customer", cascade =
CascadeType.ALL,fetch=FetchType.EAGER)
 private Set<PhoneNumber> numbers;

 // getter and setters

 public void addPhoneNumber(PhoneNumber number) {
  if (number != null) {
   if (numbers == null) {
    numbers = new HashSet<>();
   }
   number.setCustomer(this);
   numbers.add(number);
  }
 }

}
// child

@Entity
public class PhoneNumber {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private long id;
 private String number;
 private String type;

 @ManyToOne
 @JoinColumn(name = "customer_id")
 private Customer customer;
```

```
 // getter and setters

}
```

```
@Test
public void testCreateCustomer() {

 Customer customer = new Customer();
 customer.setName("John");

 PhoneNumber ph1 = new PhoneNumber();
 ph1.setNumber("1234567890");
 ph1.setType("cell");

 PhoneNumber ph2 = new PhoneNumber();
 ph2.setNumber("0987654321");
 ph2.setType("home");

 customer.addPhoneNumber(ph1);
 customer.addPhoneNumber(ph2);

 repository.save(customer);
}
```

Many To Many Association

- E.g. Patient to Doctor, Programmer to Project.
- To build these type of associations, we have to define collection types both in the parent as well as the child objects. E.g. A patient will have a list of doctors that he is going to and a doctor will have a list of patients that he is treating.
- And in the database will have three tables one for each entity and one mapping table which will map these two in the many to many association. The mapping table will have the primary keys of both entities.
- If we want to save the parent object and you also want to save the child along with it, we can simply link the child objects with the parent. That is while saving patient, if you also want to save new doctor information, you can simply add that to the list of doctors on the patient. And if you save the patient, automatically the doctor information will also be saved.
- We can also apply lazy loading, cascading effects and all that on many to many association.
- Use @ManyToMany annotation on the list/set fields which are joining points in associated tables.
- On the @ManyToMany annotation, we can use cascade, fetch attributes as well.
- For the many to many association, we have a mapping table in the database. We need to let JPA provider know about this mapping table. In order to do so, we need one more

annotation called @JoinTable. This annotation will inform hibernate about the join table using attribute 'name' and also will inform it about the keys and how are they associated. So using 'joinColumns' attribute, we tell about the primary key of this table. And using 'inverseJoinColumns' attribute, we specify about the primary key of the other table.

- By default in many to many associations the child/association information is loaded lazily.
- When we use mappedBy on a many to many association it tells JPA tools like Hibernate to check the other side of the relationship for the exact information on how to do this join or how to make this many to many association work.
- For lazy loading to work, make sure you mark your methods as `@Transactional`.
- Example-

```
create table programmer(
id int PRIMARY KEY AUTO_INCREMENT,
name varchar(20),
salary int
)

create table project(
id int PRIMARY KEY AUTO_INCREMENT,
name varchar(20)
)

create table programmers_projects(
programmer_id int,
project_id int,
FOREIGN KEY (programmer_id)
REFERENCES programmer(id),
FOREIGN KEY (project_id)
REFERENCES project(id)
)
```

```
@Entity
public class Programmer {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 private String name;
 @Column(name = "salary")
 private int sal;
 @ManyToMany(cascade = CascadeType.ALL,fetch=FetchType.LAZY)
 @JoinTable(name = "programmers_projects", joinColumns =
@JoinColumn(name = "programmer_id", referencedColumnName = "id"),
inverseJoinColumns = @JoinColumn(name = "project_id",
referencedColumnName = "id"))
 private Set<Project> projects;

 // getters and setters
}
```

```java
@Entity
public class Project {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private int id;
 private String name;
 @ManyToMany(mappedBy = "projects")
 private Set<Programmer> programmers;

 // getters and setters
}
```

```java
@Test
public void testmtomCreateProgrammer() {
 Programmer programmer = new Programmer();
 programmer.setName("John");
 programmer.setSal(10000);

 HashSet<Project> projects = new HashSet<Project>();
 Project project = new Project();
 project.setName("Hibernate Project");
 projects.add(project);

 programmer.setProjects(projects);

 programmerRepository.save(programmer);
}

@Test
@Transactional
public void testmtomFindProgrammer() {
 Programmer programmer = programmerRepository.findById(1).get();
 System.out.println(programmer);
 System.out.println(programmer.getProjects());
}
```

One To One Association

- E.g. Person to Driving License
- There are two types of one to one relationships.
- The first one is **primary key** or this can also be treated as a **bidirectional one to one relationship**. This is where the parent or the two associated objects can be directly linked or they will share the same primary key. E.g. In this case the person's primary key will also be used as a primary key in the license. We will use this kind of relationship if ALWAYS these two objects associations will exist.
- The other one is **foreign key one to one relationship**. Use this kind of relationship when these two objects associations MAY NOT always exist. For example a person may not have a driver's license but a license should be always associated with that person. It can be **unidirectional**. In this case, we will use the foreign key type of one to one relationship that in each of these entities will have their own primary key column. But child object will have a foreign key column that will reference the primary key from the parent table.
- Example –

```
create table person(
id int PRIMARY KEY AUTO_INCREMENT,
first_name varchar(20),
last_name varchar(20),
age int
)

create table license(
id int PRIMARY KEY AUTO_INCREMENT,
type varchar(20),
valid_from date,
valid_to date,
person_id int,
FOREIGN KEY (person_id)
REFERENCES person(id)
)
```

```
@Entity
public class License {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;
 private String type;
 @Temporal(TemporalType.DATE)
 private Date validFrom;
 @Temporal(TemporalType.DATE)
 private Date validTo;
 @OneToOne(cascade = CascadeType.ALL)
 @JoinColumn(name="person_id")
```

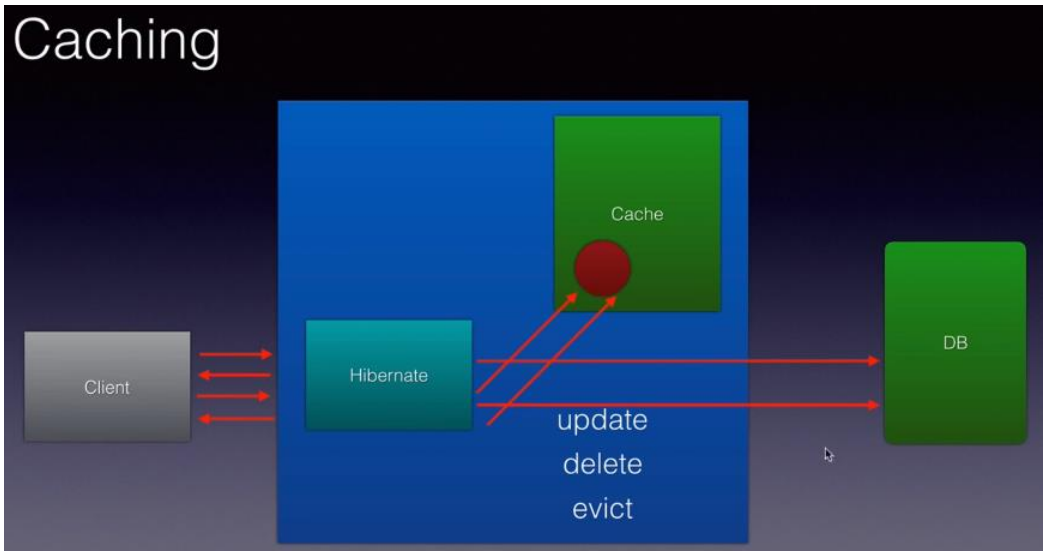
```
 private Person person;

 // getters and setters
}
```

```
@Entity
public class Person {

 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 private Long id;
 private String firstName;
 private String lastName;
 private int age;
 @OneToOne(mappedBy = "person")
 private License license;
 // getter and setters
}
```

```
@Test
public void testOneToOneCreateLicense() {
 License license = new License();
 license.setType("CAR"); license.setValidFrom(new Date());
 license.setValidTo(new Date());

 Person person = new Person();
 person.setFirstName("John"); person.setLastName("Clinton");
 person.setAge(35);

 license.setPerson(person);

 licenseRepository.save(license);
}
```

Hibernate Caching Mechanism

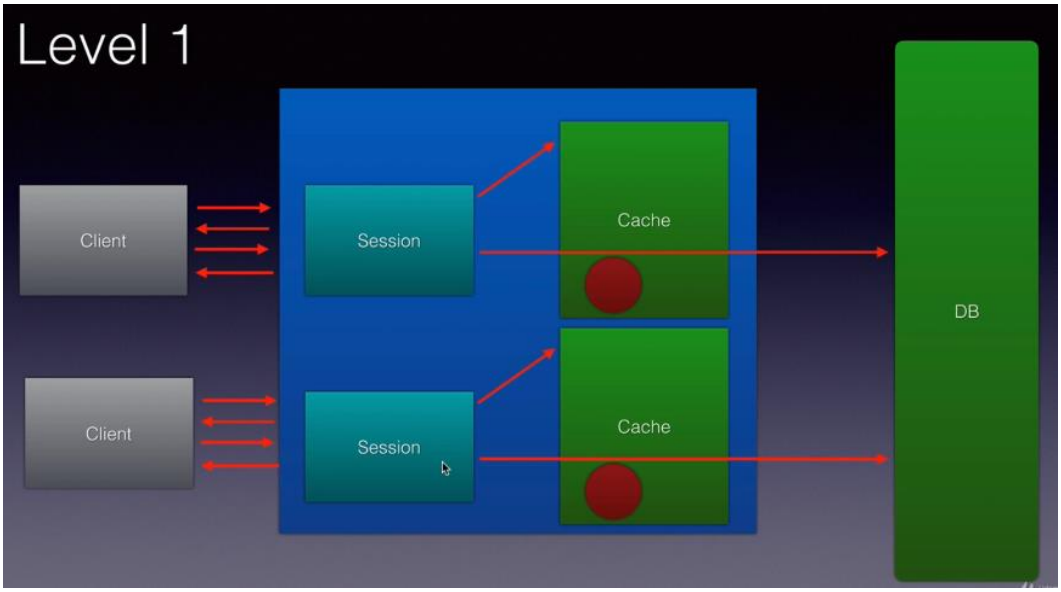
Introduction



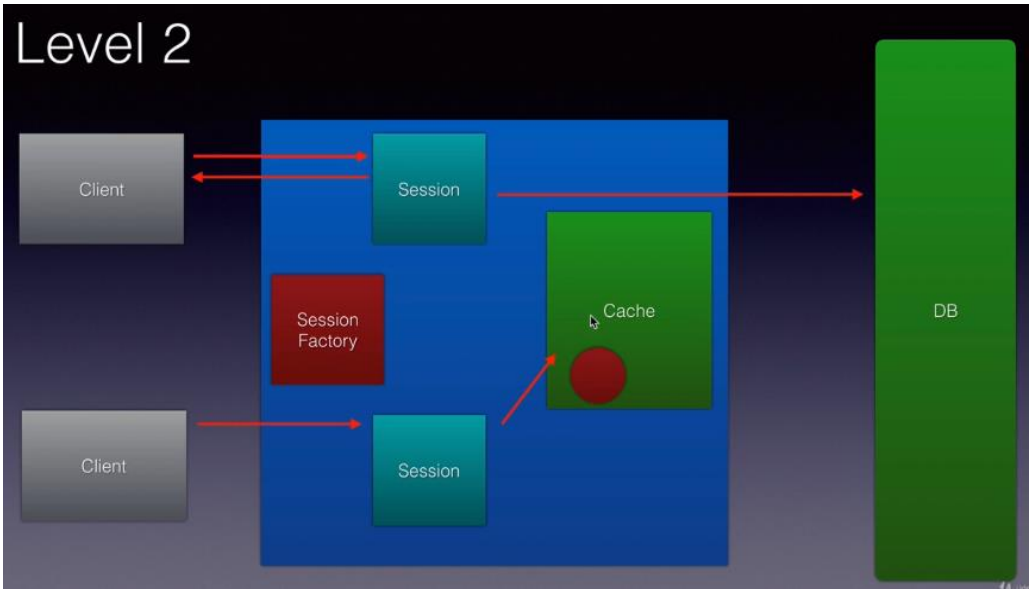
- The clients access our application and our application which is uses a ORM tool like Hibernate to read the data. These ORM tools will execute a select query internally against the database table, get the data, convert the data into object, hand it over to our application so that we can send it back to the client as required.
- Every time the client read some data from our application our application or the ORM tool will execute that select statement. So instead of repeating the same read operation multiple times we use caching or a cache.
- Caching is storing the data or an object in a temporary location.
- So the very first time that request comes in, these ORM tools or caching frameworks will read the data, convert them into object and store that object into a temporary memory location or even to the disk. This process is called caching. The next time the request comes in, these ORM frameworks or caching frameworks will first check if that data for that particular request exists in the cache. If it's there, there will be no database select queries executed .No database communication at all, simply pick the object, process it, send it back to the client, improving the performance of the application.
- And this cache can be refreshed or will be refreshed every time that object is updated that record is updated in the database automatically these caching frameworks or even hibernate will refresh the cache, it will read the data, store the object again in the cache which is updated and if it is deleted, it will be deleted from the cache. We can also programmatically evict the cache by using methods called evict on hibernates session object.

Two Levels of Cache

- Hibernates supports caching at two levels –
    - The level 1 cache is at the hibernate **session** level. (added by default)
    - And level 2 cache is at the **session factory** level. (needs additional configuration)
- Level 1 Cache (Session Level)



- Level 2 Cache (SessionFactory Level)



- Internally if we use Level 1 cache which is free which is always there by default enabled, it happens at the session level.
- A sessionfactory is used to create multiple hibernate sessions. So if we use Level 2 caching, objects will be cached at the session factory level. That is they are shared across hibernate sessions.

- Level 2 cache is very powerful because data here is cached across sessions. But level 2 cache needs some additional work. Hibernate does not have in built support for it. We use caching providers such as **ehcache, Swaram cache, Jboss tree cache, OS cache, Tangosol cache** etc. but **ehcache** is the most popular one and very easy to configure.

Level 1 Cache (Session Level)

- By default, the level one cache will be enabled.
- In order for the hibernate's level 1 caching to work, we need to mark this with @Transactional annotation from spring because spring treats level 1 caching or the spring session is associated with that transaction and for it to work we have to mark it with it @Transactional.
- E.g.

```
@Test
@Transactional
public void testCaching() {

 repository.findById(1).get();
 repository.findById(1).get();

}
```

- In above example, if you fire the same select twice, you will see only one select query being fired in the logs (turn on show_sql to true).
- In above example, if you don't use @Transactional, Level 1 caching will not work and you will see 2 select queries in the console logs.
- If we **evict** the loaded object using session.evict(), the object will be removed from the Level 1 cache. And if it is requested again in the same transaction, hibernate will check the object in the level 2 cache if it is configured, otherwise it will fire another select query in the database.

EH cache

- Fast and lightweight.
- EHCache is second level cache provider, it will cache the objects at the session factory level.
- It supports both in memory as well disk based caching. We can store the objects in the memory in the RAM or it could serialize them to the disk and de-serialize them back as required.
- And it's very powerful because it allows us to configure timeout for a particular object in the cache, the total lifetime of an object in the cache etc. using XML.

Caching Concurrency Strategies

- Based on the use cases and our application we can select four different types of cache concurrency strategies when we cache our JPA entities namely READ_ONLY, NONSTRICT_READ_WRITE, READ_WRITE, TRANSACTIONAL.
- Depending on which we choose, the caching will be impacted the way our objects will be cached our application uses caching will be impacted.

READ_ONLY

- If we choose READ_ONLY which should be chosen only when the entities in the application never change pretty much.
- If our application is a read only application which just takes the data in the database and displays it to the end user. That is when we should be using read only.
- If you use read only strategy on your JPA entity and once the data is ready, if you try to update that record in the database an exception will be thrown. So you should be careful.
- You should use read only when the applications are pretty much reading the data and displaying it to the end user and you are pretty sure that no updates will happen.

NONSTRICT_READ_WRITE

- This is not consistent cache but it's OK for several applications.
- So here the cache will be updated only when the particular transaction completely commits the data to the database.
- So in the meantime if another transaction tries to read that data it will get stale data because the other transaction hasn't committed the data to the database that will not be updated in the cache.
- So if your application is OK for **eventual consistency** then you should be using non-strict read write.

READ_WRITE

- If you need **total consistency** across transactions then you should use a read write strategy.
- This is where soft locks will be used.
- So if a transaction is updating the data which is already cached and the transaction is updating that particular record in that database, a soft lock will be associated with that Cache object within the cache. Now if another transaction comes in and it is trying to read the data it will look at this lock and it will know that another transaction is trying to update this particular object. And it should not read it from the cache and it will go directly against the database and get the latest data.
- So this strategy is more consistent So if your application needs more consistency and if it can't use the non strict read write it then go for the read write strategy.

TRANSACTIONAL

- Rarely used.
- You use this when you are dealing with XA transactions or distributed transactions. So cache changes are done in a distributed XA transaction.
- A change in a cache which could be a commit or a rollback will happen across databases and those changes will impact the cache as well.

Steps to using EH Cache

1. Add ehcache maven dependency. E.g. org.hibernate:hibernate-ehcache

```
<dependency>
 <groupId>org.hibernate</groupId>
 <artifactId>hibernate-ehcache</artifactId>
</dependency>
```

2. Enable the caching for our application (in the application.properties).

```
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCacheRegionFactory
spring.cache.ehcache.config=classpath:ehcache.xml
spring.jpa.properties.javax.persistence.sharedCache.mode=ALL
```

3. Create the ehcache.xml.

This is where we specify that timeout, the location to store our objects, the temporary location where the object should be cached and all that will go into that.

```
<ehcache>
 <!-- diskStore - path to temp store where objects will be cached by
ehcache.
   This can be property or files system path.
   java.io.tmpdir => This is inbuilt variable in Java which points to
the temporary directory.
  -->
 <diskStore path="java.io.tmpdir"/>

 <defaultCache
  <!-- max elements in cache at a time -->
  maxElementsInMemory="100"

  <!-- false means they should not live forever.
            The cachable object should be destroyed after some time
otherwise we'll run out of memory  -->
  eternal="false"

  <!-- Remove the object fro mthe cahche if it is not accessed for
more than 5 seconds here. -->
  timeToIdleSeconds="5"

  <!-- after 10 seconds the object should be removed from the cache.
No matter what. -->
  timeToLiveSeconds="10"

  <!-- So if we have any memory issues, it will be over flow to disk.
It will be saved on the disk.-->
   overflowToDisk="true"/>
</ehcache>
```

- We will then mark our entities as cacheable. Simply use an annotation on our JPA entities to put them into the cache.
- Finally test both the first level and second level caching.

4. Make Entities Cacheable

- Annotate the entity with @Cache annotation with appropriate caching strategy.
- It's always a good practice and a requirement for our entities to implement Serializable especially when we are using second level cache so that they can be written to the disk.
- As long as is in memory, it will work even without Serializable. But once it starts writing the objects to the disk our objects should be implementing Serializable.
- E.g.

```
@Entity
@Table
@Cache(usage = CacheConcerrencyStrategy.READ_ONLY)
public class Product implements Serializable {

 private static final long serialVersionUID = 1L;
```

```
 @Id
 private int id;
 // getters and setters
}
```

5. Testing Cache

- Example

```
@Test
@Transactional
public void testCaching() {
 Session session = entityManager.unwrap(Session.class);
 Product product = repository.findById(1).get(); // 1. DB

 repository.findById(1).get(); // 2. 1st Level cache

 session.evict(product);      // 3. Evicted from 1st level cache

 repository.findById(1).get(); // 4. Fetched from 2nd level cache
//and it will also put it in the first level cache

}
```

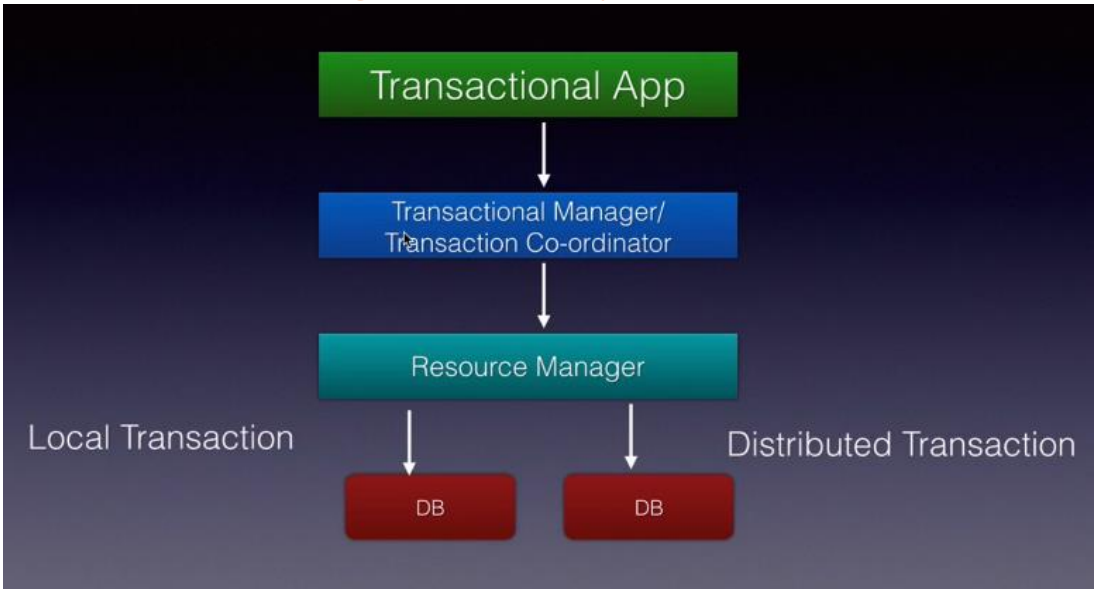
- Following the steps 1 to 4, we have second level cache setup and we already have first level cache by default. So when we run above test, only one select query will be fired. See above highlighted texts at each line.

Transaction Management

Introduction

- A transaction is an automic unit of work in which all off it happens or nothing happens.
- Every transaction has four key properties also known as ACID properties – Automicity, Consistency, Isolation and Durability.
- **Automicity** – either both the updates will happen or nothing will happen that is atomic one single unit of work.
- **Consistency** – meaning the data base should be left in a consistent state at the end of the transaction.
- **Isolation** – meaning if there are multiple transactions happening each transaction should work in isolation without impacting each other.
- **Durability** – means once the transaction changes are committed those changes should stay in the database.

Transaction Management Components



- **Transactional application** is our application which we are building where in we are enabling the transaction management.
- The transaction app will coordinate with a component called **Transactional Manager or Transaction Coordinator**.
- The transaction manager will use **Resource Manager** which knows how to communicate with the underlying resource which is usually a database.
- So the transactional app will perform commits and rollbacks that run through transactional manager or transactional coordinator which in turn will use the resource

manager which knows how to use APIs like JDBC, hibernate ORM, etc. to perform transactions on the underlying database.

- If the underlying resource (database) is just a single resource then it is called a **local transaction**.
- And if there are more than one resources that is if we have multiple databases running on different servers then we call it a **distributed transaction**.

Enabling Transaction Management

- To enable transaction management, simply wrap the method around the transaction using `@Transactional` annotation. Once we do that, everything in that method will be committed at once or will be rolled back in case of exception.

Save and Retrieve Files

- To save files into database, we generally use BLOBs or CLOBs in the database.
- BLOB (Binary Large Object) is typically used for storing word, image or video files, etc.
- In the Entity class, we need to use **`byte[]`** datatype for the field representing BLOB column. And we also need to annotate that blob field with **`@Lob`** annotation from javax.persistence.
- Once you mark this byte array with **`@Lob`** annotation, when you save this entity, automatically this byte array will be saved into the blob column in the database. Also you can read it back.

Calling Stored Procedure

- Example procedures

```
CREATE PROCEDURE GetAllProducts()
BEGIN
        SELECT *  FROM product;
END //

CREATE PROCEDURE GetAllProductsByPrice(IN price_in decimal)
BEGIN
        SELECT *  FROM product where price>price_in;
END //

CREATE PROCEDURE GetAllProductsCountByPrice(IN price_in decimal)
BEGIN
        SELECT count(*)  FROM product where price>price_in;
END //
```

- Examples using Repository

```
@Query(value = "CALL GetAllStudents", nativeQuery = true)
List<Product> findAllProducts();

@Query(value = "CALL GetAllProductsByPrice (:price)", nativeQuery =
true)
List<Product> findAllProductsByPrice(double price);

@Query(value = "CALL GetAllProductsCountByPrice (:price)", nativeQuery
= true)
int findAllProductsByNameCount(double price);
```

- Another way is using `@NamedStoredProcedureQuery` on entity class.

Working with MongoDB

- While creating project from spring initializr, include **Spring Data MongoDB** (spring-boot-starter-data-mongodb) dependency. This is to do ORM through JPA.
- Also add **Embedded MongoDB Database** dependency (artifactId `de.flapdoodle.embed.mongo`), which is required for Testing.
- Spring Boot supports an embedded Mongo DB, that is without installing an external MongoDB, we can use an embedded Mongo DB.This should be used for Unit Testing just like H2 DB.
- When you run your application, it will launch MongoDB internally for you and all the collections the databases etc. will get created in that embedded MongoDB Server.
- You need not mark your model classes with `@Entity` annotations in case of Mongo DB. Spring provides something like **`@Document`** and `@Id` is optional because Mongo DB generates the Id for every record or object you create. **`@Document`** is from **`org.springframework.data.mongodb.core.mapping`** package.
- Also instead of `JPARepository`, we need to extend `MongoRepository` from `org.springframework.data.mongodb.repository` package.
- Other steps like configuring database details in properties file, etc. are similar.
  E.g.

```
spring.data.mongo.host=localhost
spring.data.mongo.port=27017
spring.data.mongo.database=mydb
```

Composite Primary Keys

Introduction

- A composite key is a combination of more than one database table columns.
- To use a composite key in our spring data JPA applications, you can use **`@IdClass`** annotation or we can go the **`@Embedable and @EmbeddedId`** route.
- So the steps will be –
    - You will create the key class so the composite key will have its own class. Also every key class should implement Serializable interface.
    - And then we will use the class inside the entity through above annotations and
    - Then the other steps typically are to create a repository, configure the data source and test and see the composite key in action.

Using @IdClass

- Create the Composite key class and implement Serializable interface. Include the composite key fields in this key class. E.g. CustomerId class with id, email fields.
- Create entity class and mark it with @Entity, also with @IdClass and pass the key class object. E.g. @IdClass(CustomerId.class)
- In the entity class, again add the composite key field. E.g. id, email. And mark both the fields with @Id annotation.
- Since you have composite key class, so in the repository interface when we extend the JpaRepository interface, we need use the composite key class name in generics. E.g. CustomerId.

Using @Embedable and @EmbeddedId

- Using above approach of @IdClass, we have to duplicate the composite key fields in both the entity class and the composite key class.
- Below is another approach which avoids this duplication.
- Create the Composite key class and implement Serializable interface. Include the composite key fields in this key class. E.g. CustomerId class with id, email fields.
- Mark the Composite key class with **`@Embedable`** annotation.
- Create entity class and mark it with @Entity and include the field with data type as the composite key class nd mark the composite key class field as **`@EmbeddedId.`**

@IdClass Vs (@Embedable and @EmbeddedId)

- With @Embedable and @EmbeddedId annotations, it is super easy. The work as a developer is very less. And it is much more readable as well.
- With @IdClass, you have to duplicate the composite key fields in both the key class and entity class.

Tips and Tricks

- Refer – https://github.com/sameerbhilare/springdatajpausinghibernate
- Spring tool suite (STS) is a very powerful IDE that helps us build spring based projects very easily. It is very similar to eclipse with support for special support for spring based projects.
- `spring-boot-starter-parent` is a **BOM**, which is a special type of pom. BOM stand for **Bill of Materials** within which all the versions of various libraries required for our projects are defined.
- `@RunWith(SpringRunner.class)` – with this we tell Spring Boot to use `SpringRunner` instead of default JUnitRunner class to run our Test classes.
- `@SpringBootTest` - This annotation tells spring boot to search for a class that is marked with spring Boot application and use that class to create a spring application context, a container with all the beans in that application so that we can start testing those beans in that application.
- Typically, your repository interface extends `Repository, CrudRepository, or PagingAndSortingRepository`. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with `@RepositoryDefinition`. Extending `CrudRepository` exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from `CrudRepository` into your domain repository.
- For date type of fields in the entity class, we need to annotate it with @Temporal annotation. Using this, we can set whether its date only, or date and time, etc.
- EntityManager is the JPA equivalent of hibernate Session. This is used internally by the repository class, Spring gives it to the repository at runtime. But if you want to access it we simply inject it using @Autowired.
- Use property **server.servlet.context-path** to provide context root for you application. E.g. server.servlet.context-path=/clinicalservices
- If you have camelcase entity name, JPA will use underscores for corr. database table name. e.g. ClinicalData entity class will map to clinical_data database table.
- To enable CORS on our Spring REST endpoints, just annotate the resource class with **@CrossOrigin** annotation.