SPRING DATA JPA – POSTGRESQL DATABASE BY AMIGOSCODE

Application Properties

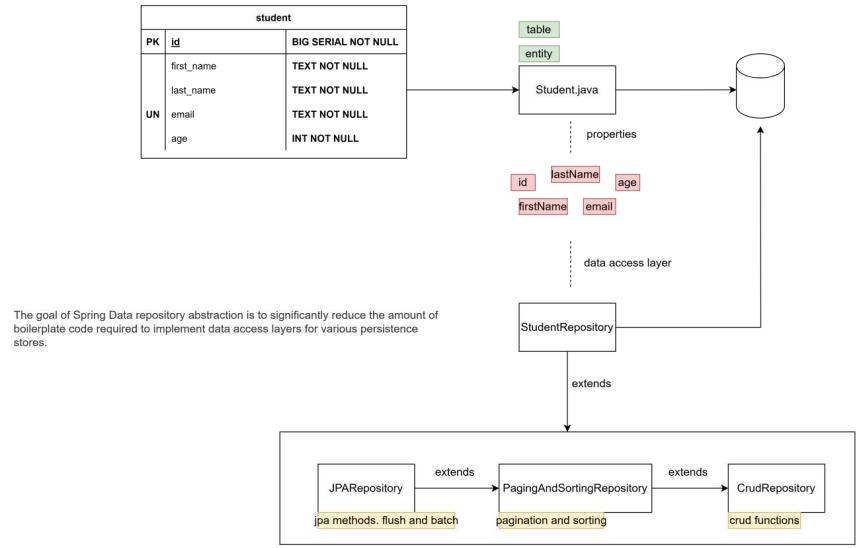
```
spring.application.name=springdatajpa
spring.datasource.url=jdbc:postgresql://localhost:5432/springdatajpa
spring.datasource.username=postgres
spring.datasource.password=root123
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.properties.hibernate.format_sql=true
```

For newer version of spring boot spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect not needed. It will select by default.

Required Dependencies

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
```

```
	<groupId>org.postgresql</groupId>
	<artifactId>postgresql</artifactId>
	<scope>runtime</scope>
</dependency>
```



spring.jpa.hibernate.ddl-auto = create-drop

- Create-drop, create, validate, update, none
- ddl – Data Definition Language

Create - This option will drop and recreate the database schema every time the application starts. This is useful for development and testing, but it should not be used in production.

Update - This option will update the schema to match the entity mappings. This is the recommended option for production use.

Validate - This option will validate the schema to ensure that it matches the entity mappings. This is useful for testing, but it should not be used in production.

none - In production, it's often highly recommended you do not specify this property. This option will disable schema generation. This is not recommended for use in production.

JPQL – Java Persistence Query Language

HQL – Hibernate Query Language

Criteria vs JPQL vs HQL

For more - https://www.baeldung.com/jpql-hql-criteria-query

Criteria Queries

The Criteria API helps in building the Criteria query object by applying different filters and logical conditions on top of it. This is an alternate way to manipulate objects and return the desired data from an RDBMS table.

The *createCriteria()* method from the Hibernate *Session* returns the persistence object instance for running a criteria query in the application. Simply put, the Criteria API builds up a criteria query that applies different filters and logical conditions.

```
<dependency>
        <groupId>org.hibernate.orm</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.4.2.Final</version>
</dependency>
```

Using the Criteria Queries and Expressions

***CriteriaBuilder* controls the query results**. It uses the *where()* method from *CriteriaQuery*, which provides *CriteriaBuilder* expressions.

```
public class Employee {

   private Integer id;
   private String name;
   private Long salary;
  // standard getters and setters
}
```

Let's look at a simple criteria query that will retrieve all the rows of *"Employee"* from the database:

```
Session session = HibernateUtil.getHibernateSession();
CriteriaBuilder cb = session.getCriteriaBuilder();
CriteriaQuery<Employee> cr = cb.createQuery(Employee.class);
Root<Employee> root = cr.from(Employee.class);
cr.select(root);

Query<Employee> query = session.createQuery(cr);
List<Employee> results = query.getResultList();
session.close();
return results;
```

The above Criteria query returns a set of all the items. Let's see how it happens:

- The *SessionFactory* object creates the *Session* instance
- The *Session* returns an instance of *CriteriaBuilder* using the *getCriteriaBuilder()* method
- The *CriteriaBuilder* uses the *createQuery()* method. This creates the *CriteriaQuery()* object that further returns the Query instance
- In the end, we call the *getResult()* method to obtain the query object that holds the results.

```
cr.select(root).where(cb.gt(root.get("salary"), 50000));
```

For its result, the above query returns the set of employees having a salary of more than 50000.

JPQL

JPQL stands for Java Persistence Query Language. Spring Data provides multiple ways to create and execute a query, and JPQL is one of these. It defines queries using the *@Query* annotation in Spring to execute both JPQL and native SQL queries. **The query definition uses JPQL by default.**

We use the *@Query* annotation to define a SQL query in Spring. **Any query defined by the *@Query* annotation has higher priority over named queries**, which are annotated with *@NamedQuery*.

Using JPQL Queries

Let's build a dynamic query using JPQL:

```
@Query(value = "SELECT e FROM Employee e")
List<Employee> findAllEmployees(Sort sort);
```

With JPQL queries that have argument parameters, Spring Data passes the method arguments to the query in the same order as the method declaration. Let's look at a couple of examples that pass method arguments into the query:

```
@Query("SELECT e FROM Employee e WHERE e.salary = ?1")
Employee findAllEmployeesWithSalary(Long salary);
```

```
@Query("SELECT e FROM Employee e WHERE e.name = ?1 and e.salary = ?2")
Employee findEmployeeByNameAndSalary(String name, Long salary);
```

For the query above, the *name* method argument is passed as the query parameter with respect to index 1, and the *salary* argument is passed as the index 2 query parameter.

Using the JPQL Native Queries

We can execute these SQL queries directly in our databases using native queries, which refer to real databases and table objects. We need to **set the value of the *nativeQuery* attribute to *true* for defining a native SQL query**. **The native SQL query will be defined in the *value* attribute of the annotation.**

Let's see a native query that shows an indexed parameter to be passed as an argument for the query:

```
@Query(value = "SELECT * FROM Employee e WHERE e.salary = ?1", nativeQuery
= true)
Employee findEmployeeBySalaryNative(Long salary);
```

Using Named Parameters makes the query easier to read and less error-prone in the case of refactoring. Let's see an illustration of a simple Named Query in JPQL and native format:

```
@Query("SELECT e FROM Employee e WHERE e.name = :name and e.salary = :salary")
Employee findEmployeeByNameAndSalaryNamedParameters(
  @Param("name") String name,
  @Param("salary") Long salary);
```

The method parameters are passed to the query using named parameters. We can define named queries by using the @Param annotation inside the repository method declaration. As a result, **the *@Param* annotation must have a string value that matches the corresponding JPQL or SQL query name.**

```
@Query(value = "SELECT * FROM Employee e WHERE e.name = :name and e.salary
= :salary",
  nativeQuery = true)
Employee findUserByNameAndSalaryNamedParamsNative(
  @Param("name") String name,
  @Param("salary") Long salary);
```

HQL

HQL stands for Hibernate Query Language. It's **an object-oriented language similar to SQL** that we can use to query our database. However, the main disadvantage is the code's unreadability. We can define our queries as Named Queries to place them in the actual code that accesses the database.

Using Hibernate Named Query

A Named Query defines a query with a predefined, unchangeable query string. These queries are fail-fast since they were validated during the creation of the session factory. Let's define a Named Query using the *org.hibernate.annotations.NamedQuery* annotation:

```
@NamedQuery(name = "Employee_FindByEmployeeId",
 query = "from Employee where id = :id")
```

Each *@NamedQuery* annotation attaches itself to one entity class only. We can use the *@NamedQueries* annotation to group more than one named query for an entity:

```
@NamedQueries({
        @NamedQuery(name = "Employee_findByEmployeeId",
        query = "from Employee where id = :id"),
        @NamedQuery(name = "Employee_findAllByEmployeeSalary",
        query = "from Employee where salary = :salary")
})
```

Stored Procedures and Expressions

In conclusion, we can use the *@NamedNativeQuery* annotation for storing the procedures and functions:

```
@NamedNativeQuery(
  name = "Employee_FindByEmployeeId",
  query = "select * from employee emp where id=:id",
  resultClass = Employee.class)
```

Advantages of Criteria Queries Over HQL and JPQL Queries

The main advantage of Criteria Queries over HQL is the nice, clean, object-oriented API. As a result, we can detect errors in Criteria API during the compilation time.

In addition, JPQL queries and Criteria queries have the same performance and efficiency.

Criteria queries are more flexible and provide better support for writing dynamic queries as compared to HQL and JPQL.

But HQL and JPQL provide native query support that isn't possible with the Criteria queries. This is one of the disadvantages of the Criteria query.

We can easily write complex joins using JPQL native queries**,** whereas it gets difficult to manage while applying the same with Criteria API.

1) Should I use the NativeQuery or JPQL Query?

Answer: - For Example, if you use the nativeQuery for the PostgreSQL Database, the same might not work for the MySQL Database. But if you use JPQL this will work because this is not the database specific whether it's Postgres, MySQL, MSSQL or any database engines. This will adapt accordingly to that database.

Recommended – Try to use the JPQL as much as possible.

* If you can't use much of the JPQL due to the database specific, then only use the Native Query. *

For more -

https://github.com/eugenp/tutorials/tree/master/persistence-modules/spring-data-jpa-query-3

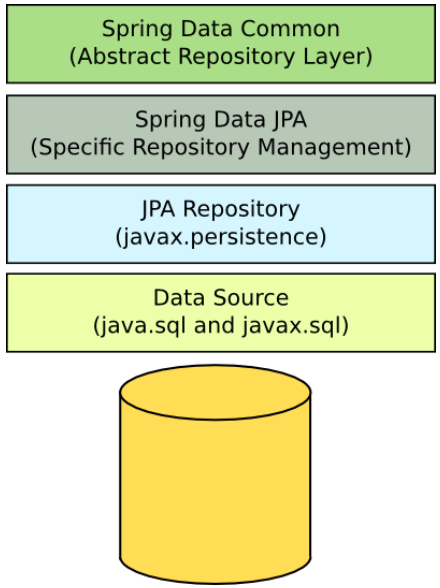
https://github.com/eugenp/tutorials/tree/master/persistence-modules/hibernate-queries

https://github.com/AndriiPiatakha/java-learnit-jpa

https://github.com/amigoscode/spring-data-jpa-course

Named Query, Native Query

@Query – Annotation is a special – because it allows us to write JPQL and Native Queries.



Important – While Inserting/Updating/Deleting a record, include @Transactional annotation.

import org.springframework.transaction.annotation.Transactional;

This library generates fake data. It's useful when you're developing a new project and need some pretty data for a showcase. For more - https://github.com/DiUS/java-faker on GitHub.

```
<dependency>
   <groupId>com.github.javafaker</groupId>
   <artifactId>javafaker</artifactId>
   <version>1.0.2</version>
</dependency>
```

For Sorting and Pagination Concept in Data JPA

```
/**
 * Pagination and Sorting
 * We want to sort by the firstName ASC
 * We want to sort by the firstName DESC
 * We want to sort by the firstName ASC and age DESC
```

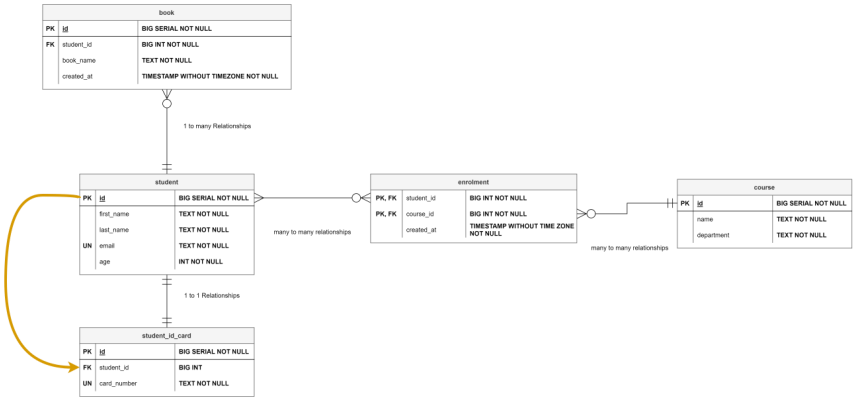
```
 * */
studentRepository.findAll(Sort.by(Direction.ASC, "firstName"));
```

```
Sort firstNameSort = Sort.by(Direction.ASC, "firstName");
Sort firstNameSort1 = Sort.by(Direction.DESC, "firstName");
Sort sort1 = Sort.by("firstName").ascending().and(Sort.by("age").descending());

studentRepository.findAll(firstNameSort).forEach(student ->
System.out.println(student.getFirstName()));

studentRepository.findAll(firstNameSort1).forEach(student ->
System.out.println(student.getFirstName()));

studentRepository.findAll(sort1).forEach(student->
System.out.println(student.getFirstName() + " " + student.getAge()));
```



One to One and Join Column Concept

Student Entity One-to-One Relationship on Student id Card Entity.

(One Student has only One Student id Card)

The PrimaryKey of Student Entity table was referenced in the StudentIdCard Entity as student_id.

```
StudentIdCard Class
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name = "sudent_id",
referencedColumnName = "id")
private Student student;
```

```
(cascade = CascadeType.ALL)
```

It propagates all operations i.e., insert, delete, update, ...

Some of the Cascade types are ALL, DETACH, MERGE, PERSIST, REFRESH, REMOVE.

ALL – Do all the operations on the Child Entity too.

In Hibernate,CascadeType.ALL is a cascade type that automatically propagates the state of an entity across associations. This means it cascades all state transitions from the parent entity to the child entities.

CascadeType.PERSIST: Cascades the create operation from the parent entity to the child entities (to continue to do something or try to do something, even when it's difficult).

PERSIST: Persists the entities present in its fields while persisting an entity (to continue to do something or try to do something, even when it's difficult).

REMOVE: Deletes an entity (To remove an entity from a database).

REFRESH: When an entity is refreshed, all the entities held in this field refresh too.

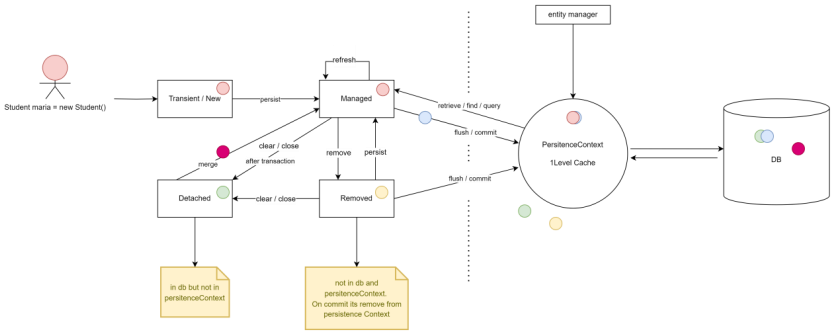
MERGE: When an entity is merged, all the entities held in this field merged too (Copying).

DETACH: The child will also remove from the persistence context.

MOST IMPORTANT Hibernate Entity Life Cycle

Student maria = new Student(); So, this entity right here enters what it's called the Transient/New state. When we invoke safe there is a method called persist which is invoked and then the lifecycle for the entity is now managed. When the entity is in a managed state, the entity will now be in the Persistence Context Entity Manager. Persistence Context is nothing but a 1st Level1 Cache. Hibernate Entity Life Cycle Explanation - https://sl.bing.net/j02ludUWS04

Hibernate Entity Life Cycle



FetchType on One-to-One Relationship is by default fetch = FetchType.Eager

EAGER - strongly wanting to do something

If you don't want the fetch type to EAGER, you can change it too LAZY. I won't load the other entities.

In the case of One-to-One Relationship FetchType.EAGER is fine. But in case of One-to-Many or Many-to-Many Relationships the default fetch type is fetch = FetchType.LAZY

UniDirectional vs BiDirectional

A unidirectional relationship in a database means that data flows in one direction, from one entity to another.

For One-to-One Relationship Uni or Bidirectional Relations are not a problem. Hibernate can handle.

But for One-to-Many or Many-to-Many Relationships.

Orphan Removal

For example, when you have a Student Entity and Student Id Card Entity, if you want to delete a student record from the DB then the student card should also be removed. But if you remove the student Id Card from the DB, it is not feasible to delete the student record.

This is when the concept of Orphan Removal comes into play.

@OneToOne(orphanRemoval = false) it's default. Make it as true to delete the associated records too.

UniDirectional @Many-to-One Relationship on Book & Student. Many books can be owned by one student.

@OneToOne fetch type EAGER default

@OneToMany fetch type LAZY default

@ManyToOne fetch type EAGER default

@ManyToMany fetch type LAZY default

For Adding Constant Enum Values in Data JPA - https://www.baeldung.com/jpa-persisting-enums-in-jpa

Enumerated Types should be in Upper Case Letters.

@Enumerated

There are Two Ways

@Enumerated(EnumType.ORDINAL)

@Enumerated(EnumType.STRING)

***@Converter* Annotation**

Using Enums in JPQL

For Example: -

```
String jpql = "select a from Article a where a.category = com.baeldung.jpa.enums.Category.SPORT";

List<Article> articles = em.createQuery(jpql, Article.class).getResultList();
```

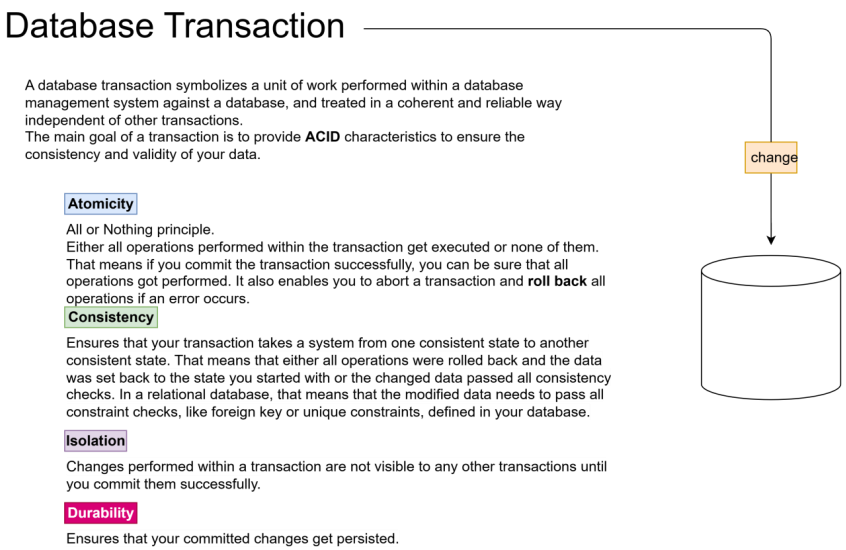
To use the named parameters: -

```
String jpql = "select a from Article a where a.category = :category";

TypedQuery<Article> query = em.createQuery(jpql, Article.class);
query.setParameter("category", Category.TECHNOLOGY);

List<Article> articles = query.getResultList();
```

DATABASE TRANSACTIONS



By default, all the Query methods are Transactional when working with the Spring Data JPA.

For delete Operation Method add @Modifying and @Transactional Annotation's.

Use @Transactional when you are modifying the data.

By default, for Transactional the readOnly is set to False i.e. you can read and modify the data.

If you set to readOnly = true, you can readonly the data. I cannot modify it. Alos, we can configure them at the Interface Level or Class Level.

For more - https://docs.spring.io/spring-data/relational/reference/jdbc/transactions.html

GitHub Repo - https://github.com/BHIMAVARAPU-MANOJ-KUMAR/Spring-Data-Jakarta-Persistence-API