**Class Notes Java Spring Boot JavaExpress**

**Class-08**

**DSL(Domain Specific Language)/DQ(Derived Query) Methods:**

Whenever we are applying database operations, we are using predefined methods of **JpaRepository** like: findById(), findAll(), save(), deleteById(),delete(entity) ..etc.

In traditional way i.e we are passing Id or entity etc… to perform predefined operations. But instead of traditional way like find User with Id we have to do with other ways to find anything from Database like find User by username or email or with the help of any other field.

**Requirement:** Generally, we are retrieving User record from Database using UserId, instead of that retrieve using Email or Username.

Then in these type of scenario we have to write one Strategy called **DSL(Domain Specific Language) methods** in Child(Repository) of JpaRepository(I). It will generate the SQL Query for us.

**findById(id):** select \* from User where id=1;

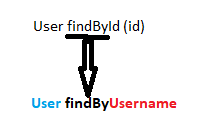
but **we want to fetch using like this**: select \* from User where username=’rahul’; or

select \* from User where username=’rahul’ and password=’rahul123’;

In JpaRepository we don’t have any method like this to fetch the record. Now how we can achieve these types of functionalities. We have to **write DSL methods.**

Now when we fetching with username=’rahul’, we will get single User object as Output.

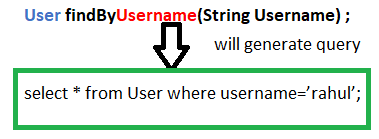
**DSL method with Username:**

**User findByUsername(String Username) ;** 

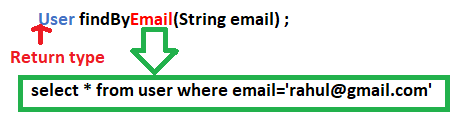
***findBy*** is **starting syntax of DSL method**

On ***User(Entity)*** class on which variable you are defining as Query, that variable will write on Camel Case (**Username**) as above.

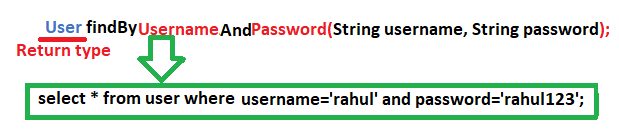
At runtime the above method will generate Query for use like this by JPA.

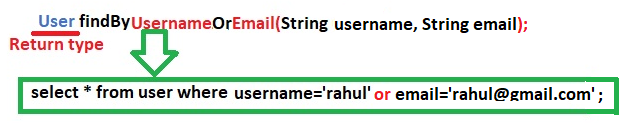


select \* from User where username=’rahul’; 🡪 **Normal SQL Query**



When we are using ***more than One condition*** then Methods**:**





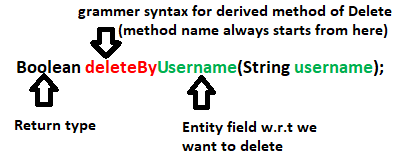
**DSL method for Delete operation:**

Annotate every derived DML method with two annotations

**@Transactional**

**@Modifying**

***deleteBy*** is **starting syntax of DSL method**



**JPQL**

**Native Query(Normal SQL):-** It is Database dependent SQL Query like: Oracle SQL, Postgres SQL.

A Native query is SQL statement that is specific to a particular database like MySQL.

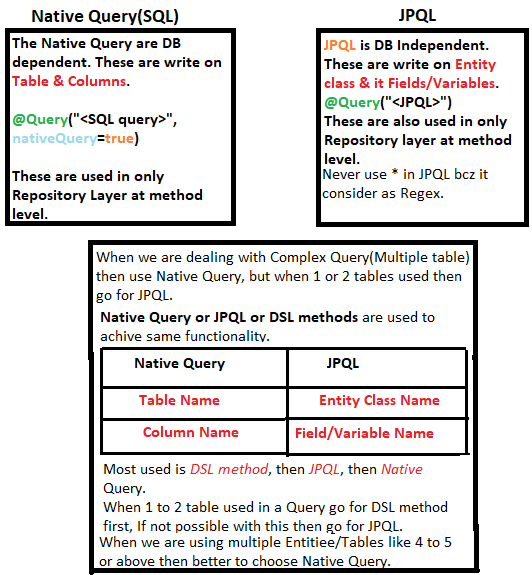
The Native Query is the query which run directly on their specific Database to perform Database operation is called as Native Query.

**JPQL(Java Persistence Query Language):-** JPQL used by Spring Data JPA by default.

JPQL are Database independent. It will execute on any database.

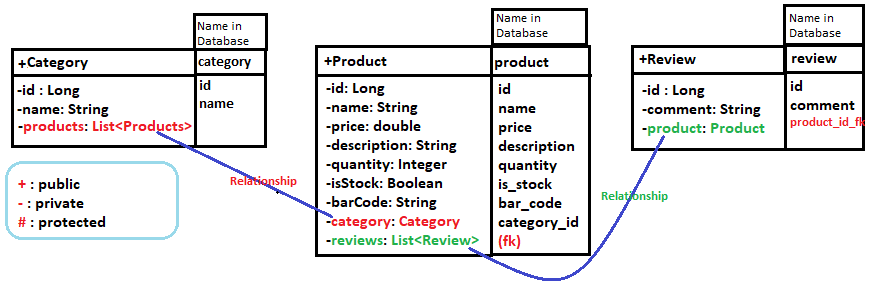
**Ex:- Native Query(Normal SQL Query):-** SELECT \* FROM Student ORDER BY age;

**JPQL:-** select s from Student s order by s.age desc;



**Note:** Whenever we are writing **Native Query** or **JPQL** on any method, then no need to follow any syntax for writing the method name i.e whenever we are using @Query annotation on method, we can use any method name.

**Entities:**

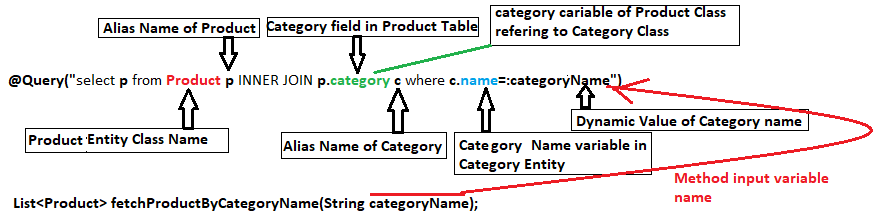


***Ex:* Scenario:** Retrieve All Product from Product table using Category Name.

**Using Native Query: Ex:**

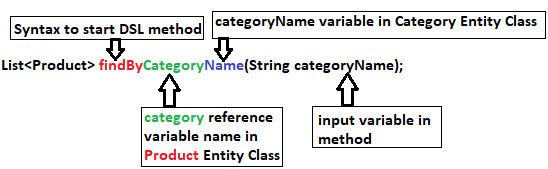


**Using JPQL: Ex:**



Here we **are not writing joining condition** because on Entity Class Level we already **joined** **Product** and **Category** both entities using **@JoinColumn.**

**Using DSL method: Ex:**



Now here in DSL method, JPA will generates own query using DSL method. Here no need to write @Query annotation and no need to write query.

**Scenario: Requirement:** Fetch Product using Barcode

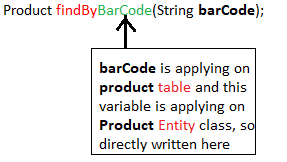
**Using JPQL:**



**Using Native Query:**



**Using DSL method:**

****

**When to use which Method?**

|  |  |  |
| --- | --- | --- |
| **Priority** | **Method** | **When to use** |
| 90-95% | DSL method | When we are using 1 to 2 table then go for DSL method |
| 3-4% | JPQL | When we are using 1 to 2 table then go for JPQL |
| 0-1% | Native SQL | When to write more complex query like 4 to 5 table or above |

**EXCEPTION HANDLING**

We can create our own Exception i.e called as Custom Exception by extending the **RuntimeException(C).**

**Process:**

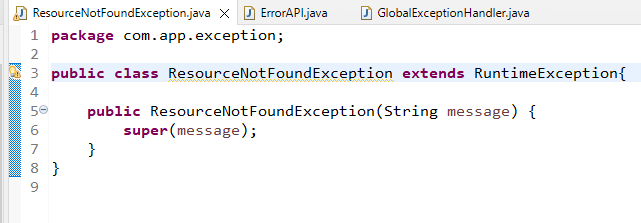
1. ***Create class by extending the RuntimeException. Like this we can create any no. of class.***
2. ***Then create constructor with super(str) in every Custom Exception class.***
3. ***Create a Model Class as ErrorAPI/Error Model class.***
4. ***Create GlobalExceptionHandler, in that create Method for Every Custom Exception class.***

**Annotations:**

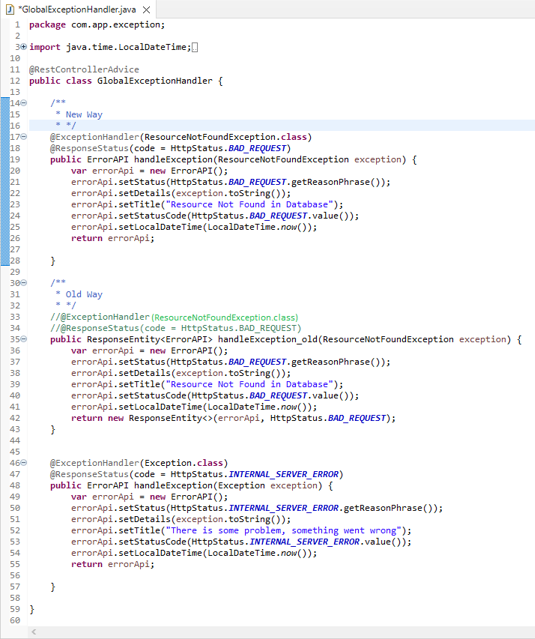
**@RestControllerAdvice (GlobalExceptionHandler class level)**

**@ExceptionHandler (on methods of GlobalExceptionHandler class)**

**@ResponseStatus (on methods of GlobalExceptionHandler class to show the Response status)**





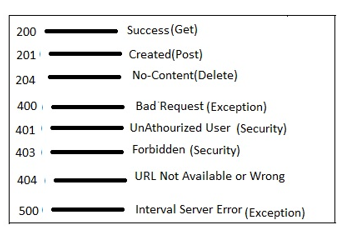


**Status Codes:**

Visit this link: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

<https://umbraco.com/knowledge-base/http-status-codes/>





**Working with ENUM in Entity**

When building our persistence layer with[**Spring Data JPA**](https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa), we often work with entities with [enum](https://www.baeldung.com/a-guide-to-java-enums) fields. These enum fields represent a fixed set of constants, such as the status of an order, the role of a user, or the stage of an article in a publishing system.

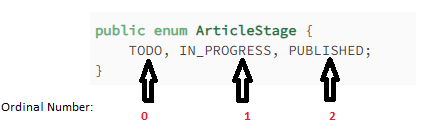
Querying entities based on their Enum fields is a common requirement, and Spring Data JPA provides several ways to accomplish this,

Ordinal Number of Enum constants starts from 0(Zero)

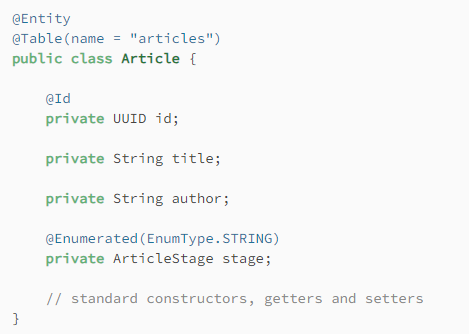
When we are working with Enum in Entity class, that time It store Enum in Database by 2 ways

1. Ordinal Number(0,1,2….) (No need to add @Enumerated annotation)
2. String value of Enum (add **@Enumerated(EnumType.STRING)** on field level in Entity)

**Enum for ArticleStages:**



Enum in Entity class:



We map our *Article* entity class to the *articles* database table. Additionally, **we use the *@Enumerated* annotation to specify that the *stage* field should be persisted as a string in the database.** Otherwise it stores as 0/1/2 in Database as Ordinal Number.

**Auditing Using Spring Data JPA**

Whenever we are creating a table using Entity classes, there is no timestamp like ***at what time record is created, updated or by whom record is created or updated.*** I want to manage these things in all entity classes

Auditing is a fundamental aspect of software development, especially when dealing with data-driven applications. It involves tracking changes to data, monitoring user actions, and maintaining a historical record of these activities. In the context of Spring Boot applications, auditing plays a crucial role in ensuring data integrity, security, and compliance with regulatory requirements.

**What Is Auditing?**

At its core, auditing involves recording and monitoring actions taken within a software application. These actions can include:

* **Creation**: Tracking when a new record or entity is added to the system.
* **Modification**: Monitoring changes made to existing data, including updates and edits.
* **Deletion**: Recording when data is removed or marked as deleted.
* **Access Control**: Keeping a log of who accessed certain data and when.

Auditing serves several important purposes:

* **Data Integrity**: It helps maintain [data consistency](https://dzone.com/articles/consistency-in-databases) and prevents unauthorized or malicious changes.
* **Security**: Auditing aids in identifying and responding to security breaches or suspicious activities.
* **Compliance**: Many industries and applications require auditing to comply with regulations and standards.
* **Troubleshooting**: It simplifies the process of identifying and resolving issues by providing a historical trail of actions.

**Auditing with JPA:**

JPA doesn’t explicitly contain an auditing API, but we can achieve this functionality by using entity lifecycle events.

JPA, the Java Persistence API, plays a pivotal role in enabling auditing within your Spring Boot application. It introduces the concept of entity listeners, which allows you to define methods that respond to lifecycle events of JPA entities. For auditing purposes, we are particularly interested in three key lifecycle events:

* **@PrePersist:** This event occurs just before an entity is first saved in the database, making it suitable for capturing creation-related information.
* **@PreUpdate**: This event takes place just before an entity is updated in the database, making it ideal for tracking modifications.
* **@PreRemove**: This event occurs just before an entity is removed from the database.

**How to work with Auditing in JPA (Notes by Me):**

Annotations for Auditing

**@EnableJpaAuditing** ->(Class Level Annotation) Write on Stater/Entry class of Spring Boot App

**@CreatedBy** ->(Field/Variable level) Write on Auditable class(Abstract class), createdBy field

**@CreatedDate**->(Field/Variable level) Write on Auditable class(Abstract class), createdDate field

**@LastModifiedBy**->(Field/Variable level) Write on Auditable class(Abstract class), updatedBy field

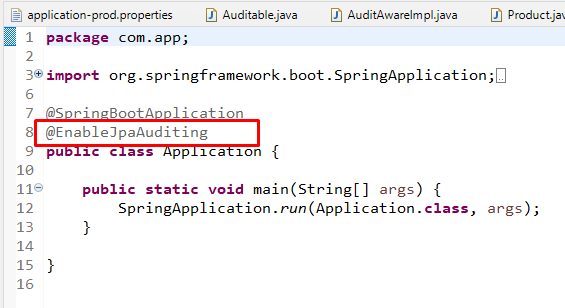
**@LastModifiedDate**->(Field/Variable level) Write on Auditable class(Abstract class), updatedDate field

**@MappedSuperClass** -> Write on the Auditable class(Because Every Entity class Extend this)

**@EntityListener(AuditingEntityListener.class)** -> Write at Class Level of Every Entity class to Listen the Activity which on or what time some one performing changes on Database

**Steps to Enable JPA Auditing in Our Spring boot Applications:**

Step1: First, we have to add **@EnableJpaAuditing** on **starter(main)** class of Spring boot App for enabling Auditing in our application



Step2: Create a package like “**com.app.auditing**”, in that package create a **abstract class** named as **Auditable.java** and annotate this class with **@MappedSuperClass (every Abstact class which will we extend by any other class should be annotate with this annotation )**

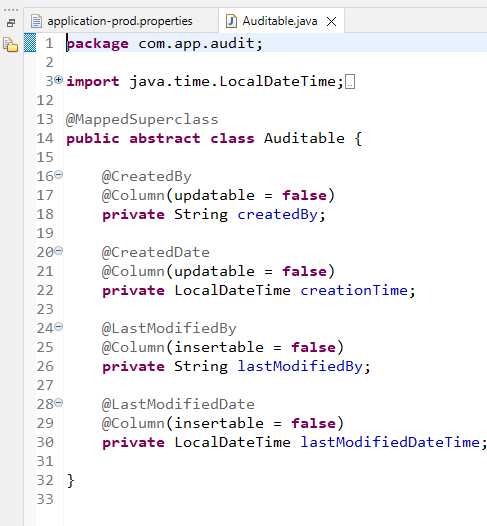
**@CreatedBy, @CreatedDate (these fields should not be updatable )**

**@LastModifiedBy, @LastModifiedDate (these fields should not be insertable)**

**Updatable = false :-> means during insertion time it should not insert any value in that column**

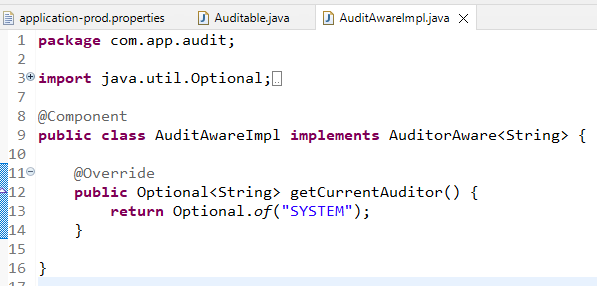
**Insertable = false :-> means during updating time it should not insert any value in that column**

Create class as per below image:

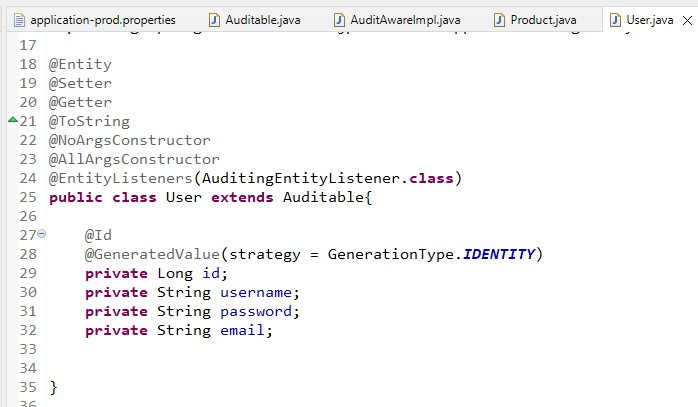


Step3: Create a class **AuditAwareImpl.java** which implements **AuditAware.java** (***Interface***) and make this class as a Spring Bean i.e annotate this class with **@Component** and implement the

**Method** “***getCurrentAuditor()”,*** actually we have to write ***security related logic*** here means which Logged In User created the Data in Database and which Logged In User updated the Data in Database and time of every activity we have to write



Step4: Every ***Entity*** class must ***extend*** ***AuditAwareImpl.java*** class and ***annotate every Entity class*** with ***@EntityListener(AuditingEntityListener.class)***



**Mapping in JPA**

**OneToOne Mapping:**

**Asynchronous Execution in Spring Boot**

By using this, the performance of the Application will improve.

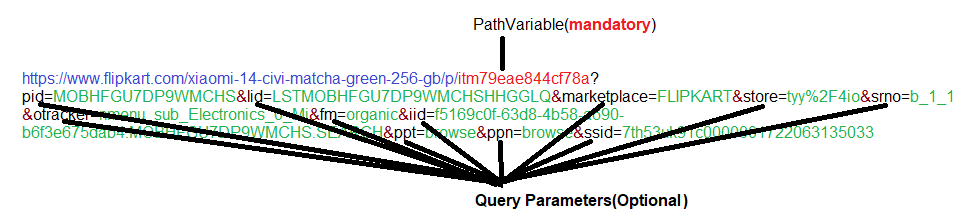
**Pagination, Sorting & Versioning in JPA**

**Properties To check for Pagination:**

* Total no of pages
* Total no. of Records
* Records Per Pages
* Sorting Types🡪 Ascending and Descending (ASC and DESC)
* Default Property(s)

**@RequestParam**🡪 Query parameters are optional but we can use for Mandatory also

**@PathVariable**🡪 Path Variable(s) is/are Mandatory



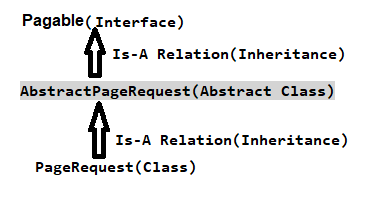
<https://www.flipkart.com/xiaomi-14-civi-matcha-green-256-gb/p/itm79eae844cf78a?pid=MOBHFGU7DP9WMCHS&lid=LSTMOBHFGU7DP9WMCHSHHGGLQ&marketplace=FLIPKART&store=tyy%2F4io&srno=b_1_1&otracker=nmenu_sub_Electronics_0_Mi&fm=organic&iid=f5169c0f-63d8-4b58-a690-b6f3e675dab4.MOBHFGU7DP9WMCHS.SEARCH&ppt=browse&ppn=browse&ssid=7th53uk91c0000001722063135033>

URL with Request Parameters

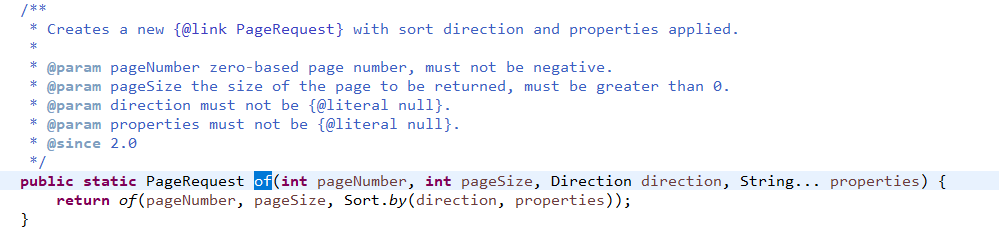
***OR***

<https://www.flipkart.com/xiaomi-14-civi-matcha-green-256-gb/p/itm79eae844cf78a>

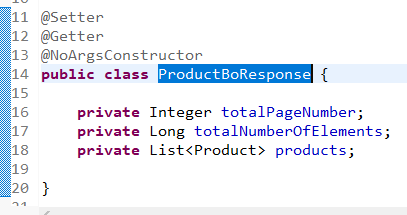
URL with Path Variable



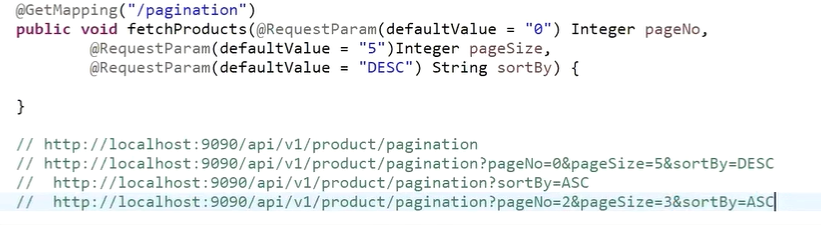
Predefined method of PageRequest class



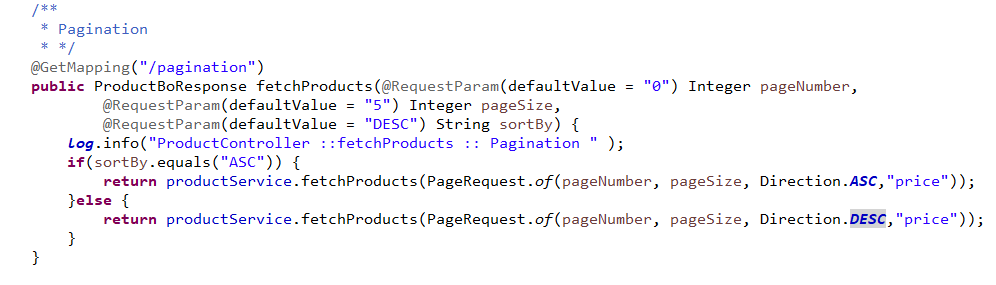
**ProductBoResponse:**



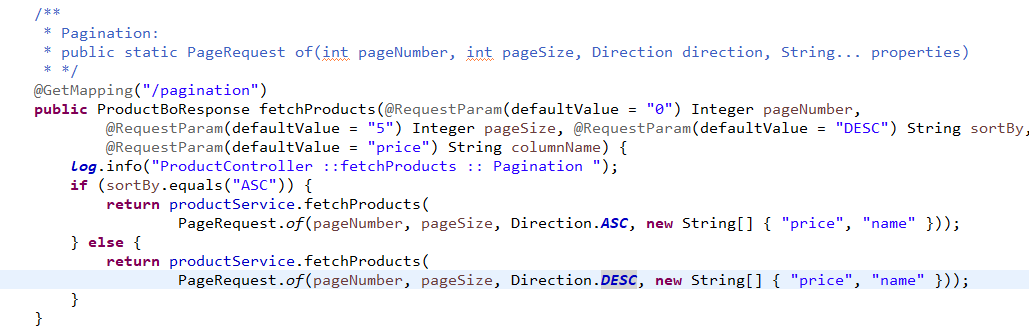
**Controller:**



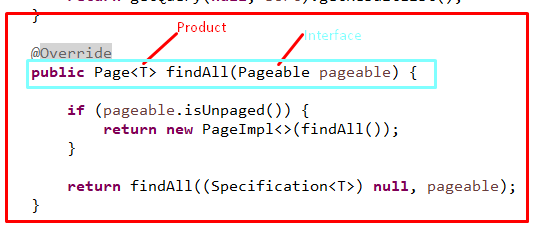
Possible URLs to call this API



**OR**



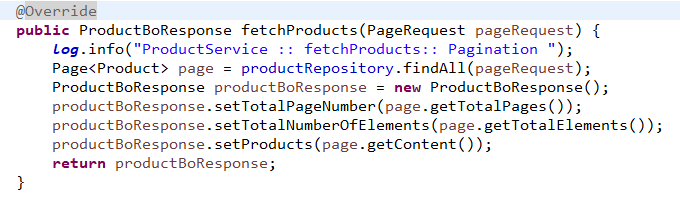
**Predefined Method of “SimpleJpaRepository” class for Pagination**



If a method need an Interface as Method Parameter, instead of that we can pass its Implementation class.

Like: Here **findAll(Pagable pagable)** but we are passing its implementation class **“PageRequest”**

**Service:**



**Possible URLs:**

[**http://localhost:8081/api/v1/product/pagination**](http://localhost:8081/api/v1/product/pagination) **: Default values will set of method parameters**

[**http://localhost:8081/api/v1/product/pagination?pageNumber=0**](http://localhost:8081/api/v1/product/pagination?pageNumber=0)

[**http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6**](http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6)

[**http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6&sortBy=”ASC**](http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6&sortBy=)**”**

[**http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6&sortBy=”DESC**](http://localhost:8081/api/v1/product/pagination?pageNumber=0&pageSize=6&sortBy=)**”**