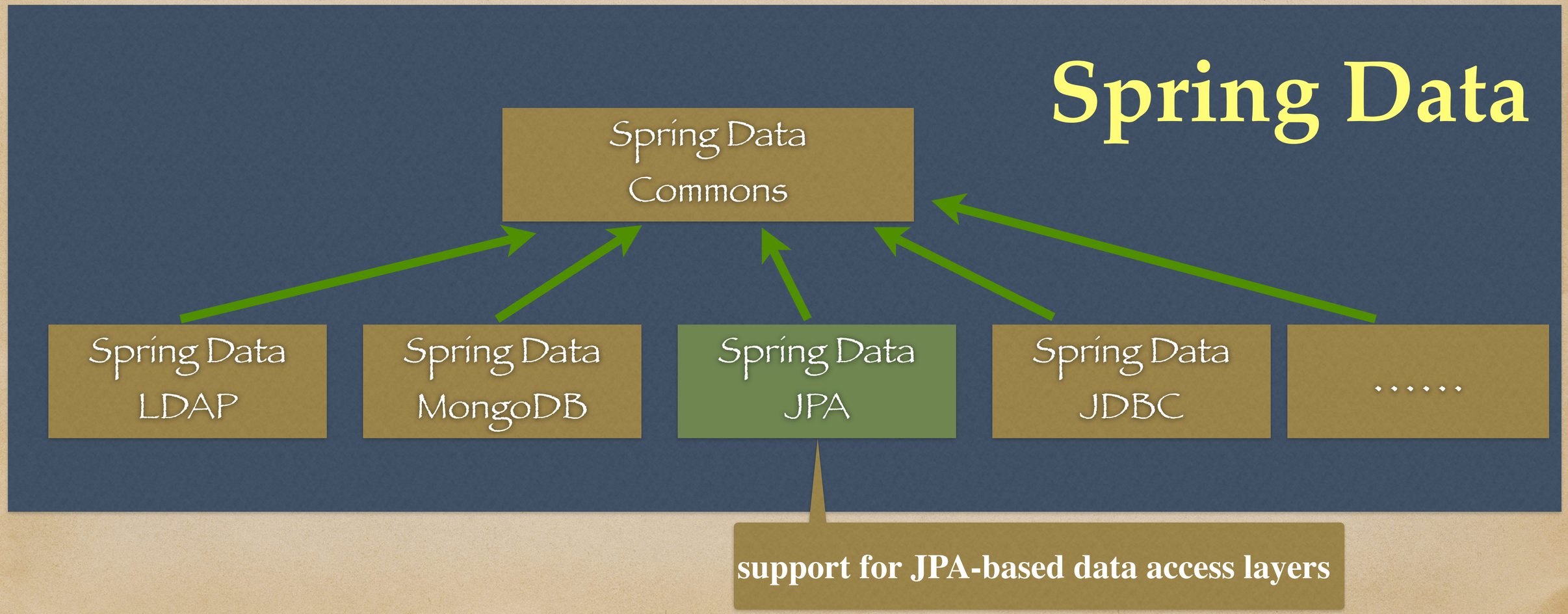
# Spring Data

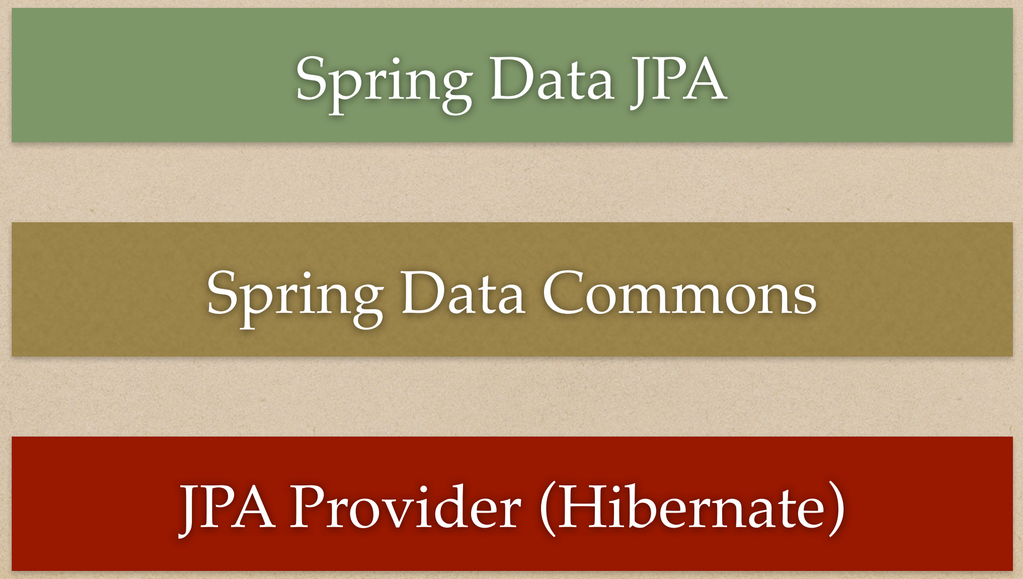
By Ramesh Fadatare (Java Guides)

# Spring Data

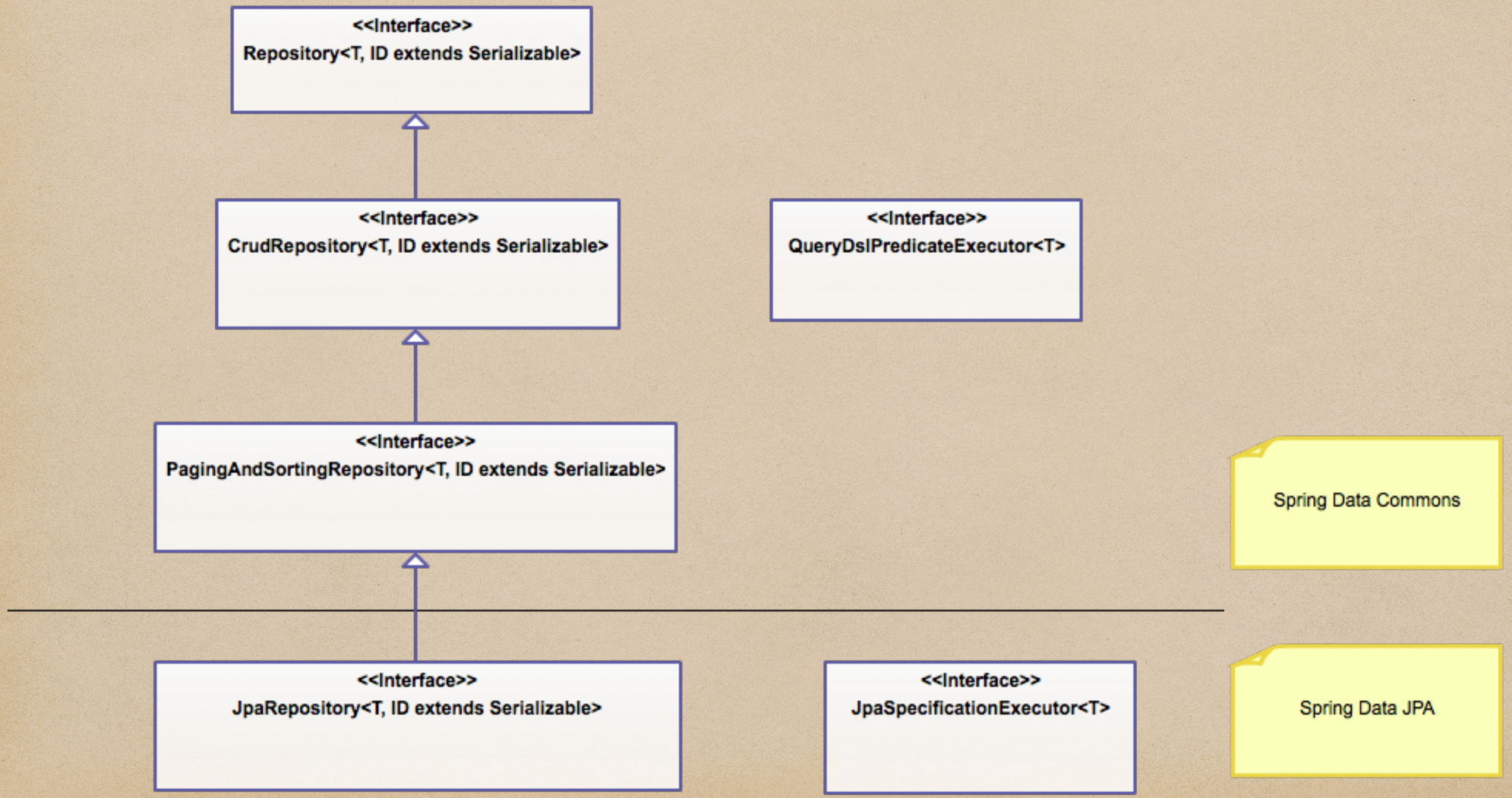

Spring Data
Spring Data
Commons
Spring Data
LDAP
Spring Data
MongoDB
Spring Data
JPA
Spring Data
JDBC
……
support for JPA-based data access layers

# Spring Data JPA?

**Spring Data JPA is not a JPA provider. It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider. If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:**



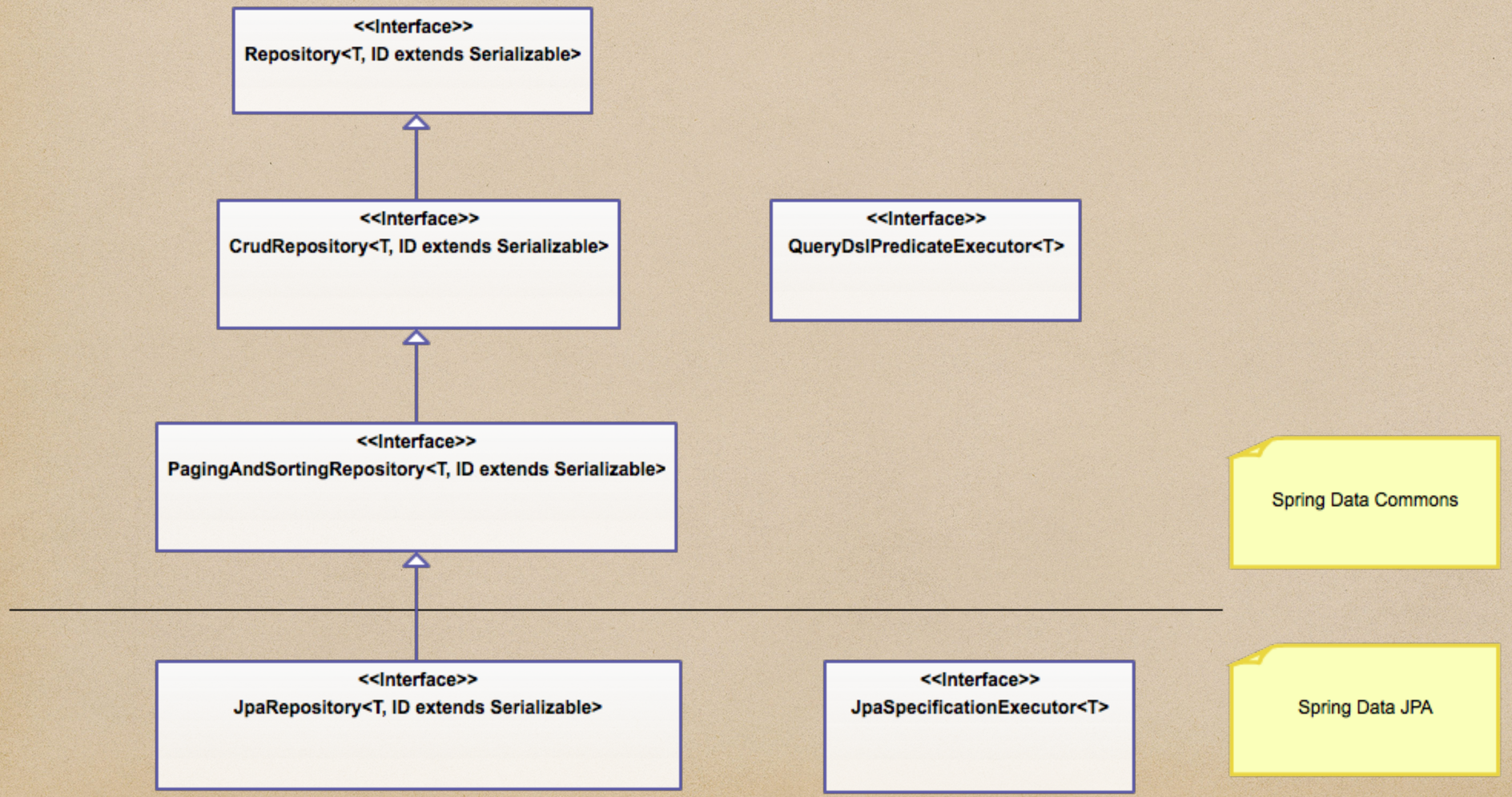
# Spring Data Commons Repository Interfaces



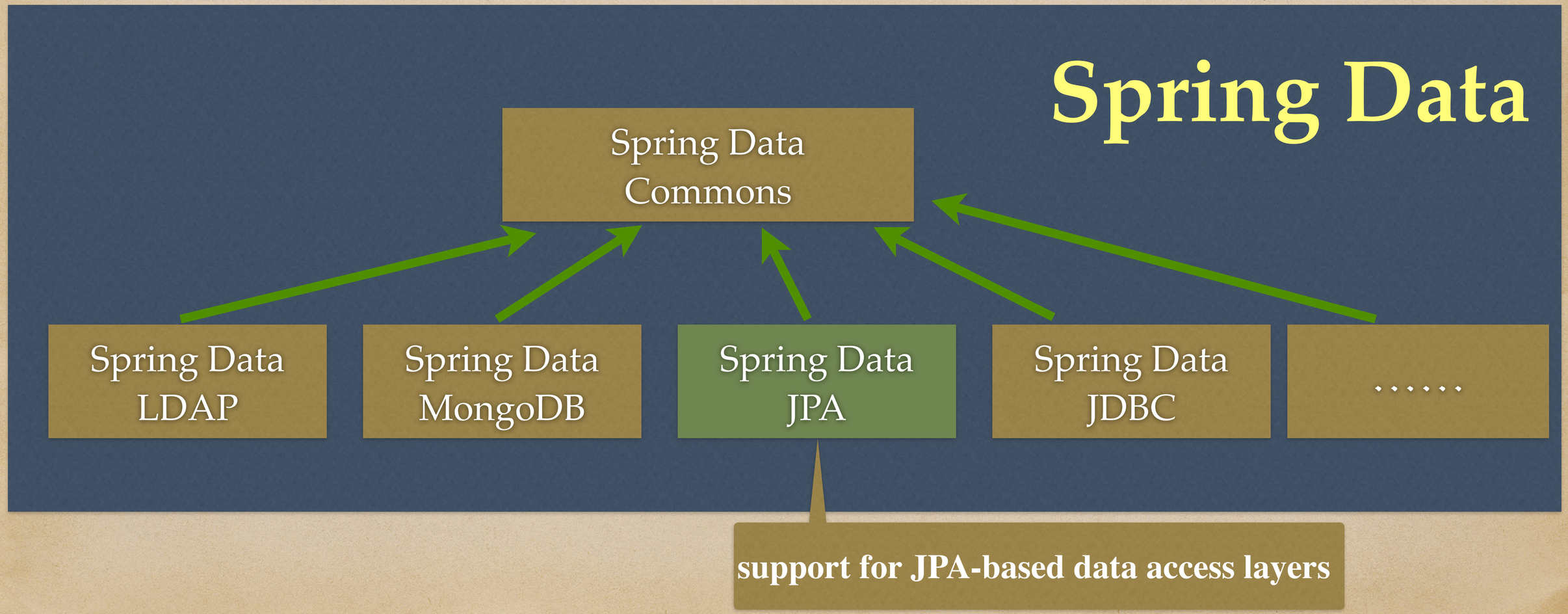
# How to use Spring Data JPA Repository

By Ramesh Fadatare (Java Guides)

# Spring Data Commons and Spring Data JPA Repository Interfaces



# Spring Data


Spring Data
Spring Data
Commons
Spring Data
LDAP
Spring Data
MongoDB
Spring Data
JPA
Spring Data
JDBC
……
support for JPA-based data access layers

# JpaRepository

```
public interface ProductRepository extends JpaRepository<Product,Integer> {

}
```

Entity Type

Primary Key

CRUD Operations

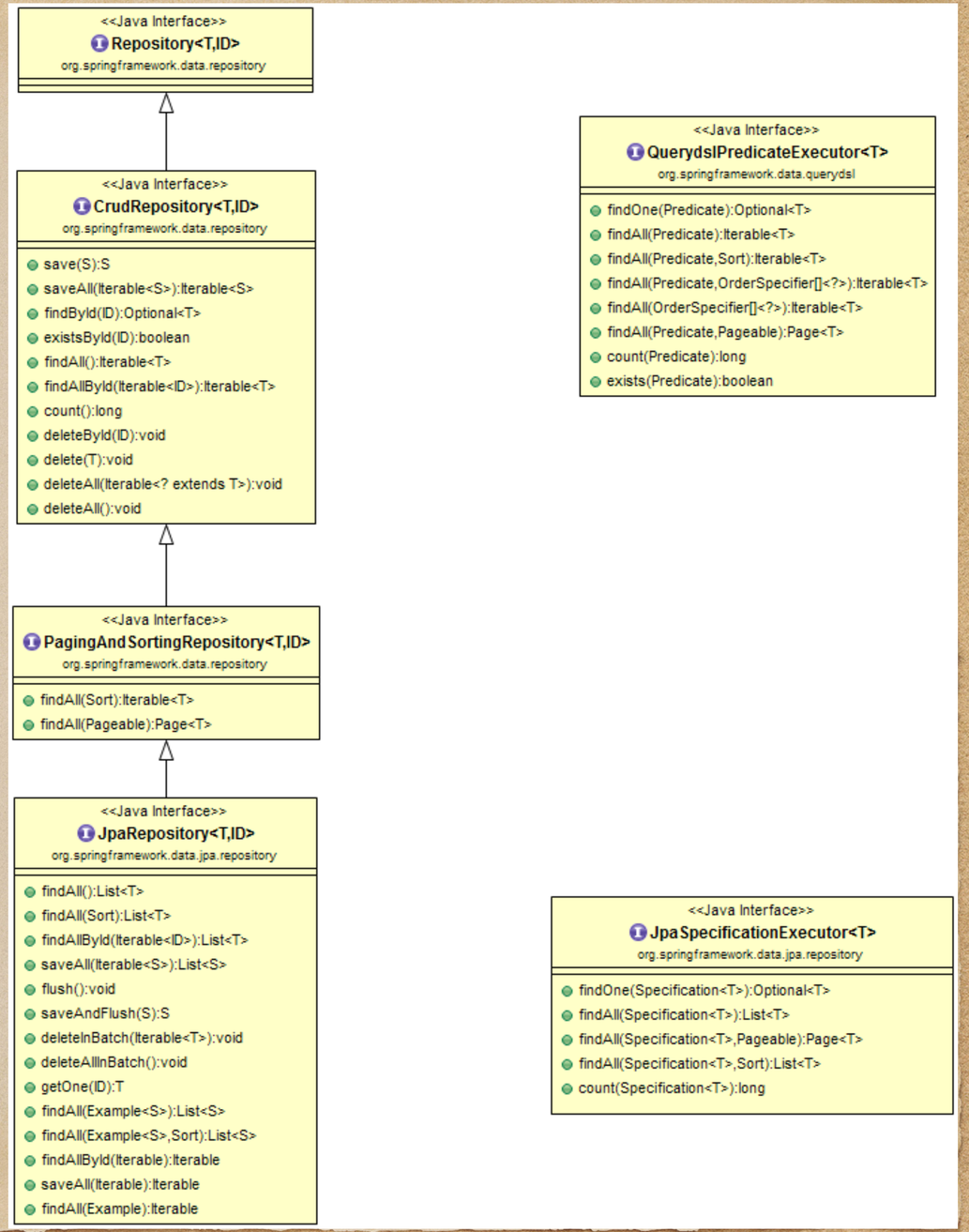
findAll()

findById()

save()

deleteById()

- - - - - -



# JpaRepository Implementation

SimpleJpaRepository implementation class provides implementation for methods

```java
@Repository
@Transactional(
    readOnly = true
)
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {
    private static final String ID_MUST_NOT_BE_NULL = "The given id must not be null!";
    private final JpaEntityInformation<T, ?> entityInformation;
    private final EntityManager em;
    private final PersistenceProvider provider;
    @Nullable
    private CrudMethodMetadata metadata;
    private EscapeCharacter escapeCharacter;
```

JpaRepository Interface

```java
@NoRepositoryBean
public interface JpaRepositoryImplementation<T, ID> extends JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
    void setRepositoryMethodMetadata(CrudMethodMetadata crudMethodMetadata);

    default void setEscapeCharacter(EscapeCharacter escapeCharacter) {
    }
}
```

# Steps to create and use Spring Data JPA Repository

1. Create a repository interface and extend to JpaRepository interface

2. Add custom query methods to the created repository interface (if we need them)

3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring Data Jpa.

# 1. Create a repository interface and extend to JpaRepository interface

```
public interface ProductRepository extends JpaRepository<Product,Integer> {

}
```

JPA Entity

Primary Key

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "PRODUCT_TBL")
public class Product {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private int quantity;
    private double price;
}
```

# 2. Add custom query methods to the created repository interface (if we need them)

```
public interface ProductRepository extends JpaRepository<Product,Integer> {
    Product findByName(String name);
}
```

Query method or finder method

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "PRODUCT_TBL")
public class Product {

    @Id
    @GeneratedValue
    private int id;
    private String name;
    private int quantity;
    private double price;
}
```

# 3. Using Repository interface in our project

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository repository;

    public Product saveProduct(Product product) {
        return repository.save(product);
    }

    public List<Product> saveProducts
        return repository.saveAll(pro
    }

    public List<Product> getProducts()
        return repository.findAll();
    }

    public Product getProductById(int i
        return repository.findById(id).
    }
```

Our repository

Calling our repository save() method

Calling our repository saveAll() method

Calling our repository findAll() method

# Minimised boilerplate code

## Before Spring Data JPA

```java
public interface EmployeeDAO {

	public List<Employee> findAll();

	public Employee findById(int theId);

	public void sav

	public void del

}
```

```java
@Repository
public class EmployeeDAOJpaImpl implements EmployeeDAO {

	private EntityManager entityManager;

	@Autowired
	public EmployeeDAOJpaImpl(EntityManager theEntityManager) {
		entityManager = theEntityManager;
	}

	@Override
	public List<Employee> findAll() {
		// create a query
		Query theQuery = entityManager.createQuery("from Employee");
		// execute query and get result list
		List<Employee> employees = theQuery.getResultList();
		// return the results
		return employees;
	}

	@Override
	public Employee findById(int theId) {
		// get employee
		Employee theEmployee = entityManager.find(Employee.class, theId);
		// return employee
		return theEmployee;
	}

	@Override
	public void save(Employee theEmployee) {
		// save or update the employee
		Employee dbEmployee = entityManager.merge(theEmployee);
		// update with id from db ... so we can get generated id for save/insert
		theEmployee.setId(dbEmployee.getId());
	}

	@Override
	public void deleteById(int theId) {
		// delete object with primary key
		Query theQuery = entityManager.createQuery("delete from Employee where id=:employeeId");
		theQuery.setParameter("employeeId", theId);
		theQuery.executeUpdate();
	}
}
```

2 Files
30+ lines of code

## After Spring Data JPA

```java
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

	// that's it ... no need to write any code LOL!

}
```

1 File
3 lines of code

No need for implementation Class

# 4 Primary Key Generation Strategies

By Ramesh Fadatare (Java Guides)

# 4 Primary Key Generation Strategies

1. GenerationType.AUTO

2. GenerationType.IDENTITY

3. GenerationType.SEQUENCE

4. GenerationType.TABLE

# 1. GenerationType.AUTO

The GenerationType.AUTO is the default generation type and lets the persistence provider choose the generation strategy.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "id")
private Long id;
```

If you use Hibernate as your persistence provider, it selects a generation strategy based on the database-specific dialect.

For most popular databases, it selects GenerationType.SEQUENCE which I will explain in a further section.

# 2. GenerationType.IDENTITY

```java
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id")
private Long id;
```

**It relies on an auto-incremented database column and lets the database generate a new value with each insert operation.**

**From a database point of view, this is very efficient because the auto-increment columns are highly optimized, and it doesn’t require any additional statements.**

**Not good for JDBC batch operations**

# 3. GenerationType.SEQUENCE

```
@GeneratedValue(strategy = GenerationType.SEQUENCE,
        generator = "product_generator")
@SequenceGenerator(name = "product_generator",
        sequenceName = "product_sequence_name",
        allocationSize = 1)
```

**The GenerationType.SEQUENCE is to generate primary key values and uses a database sequence to generate unique values.**

**It requires additional select statements to get the next value from a database sequence. But this has no performance impact on most applications.**

**The @SequenceGenerator annotation lets you define the name of the generator, the name, and schema of the database sequence and the allocation size of the sequence.**

# 4. GenerationType.TABLE

```java
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
@Column(name = "id")
private Long id;
```

**The *GenerationType.TABLE* gets only rarely used nowadays.**

**It simulates a sequence by storing and updating its current value in a database table which requires the use of pessimistic locks which put all transactions into a sequential order.**

**This slows down your application, and you should, therefore, prefer the *GenerationType.SEQUENCE*, if your database supports sequences, which most popular databases do.**

# Spring Data JPA Methods

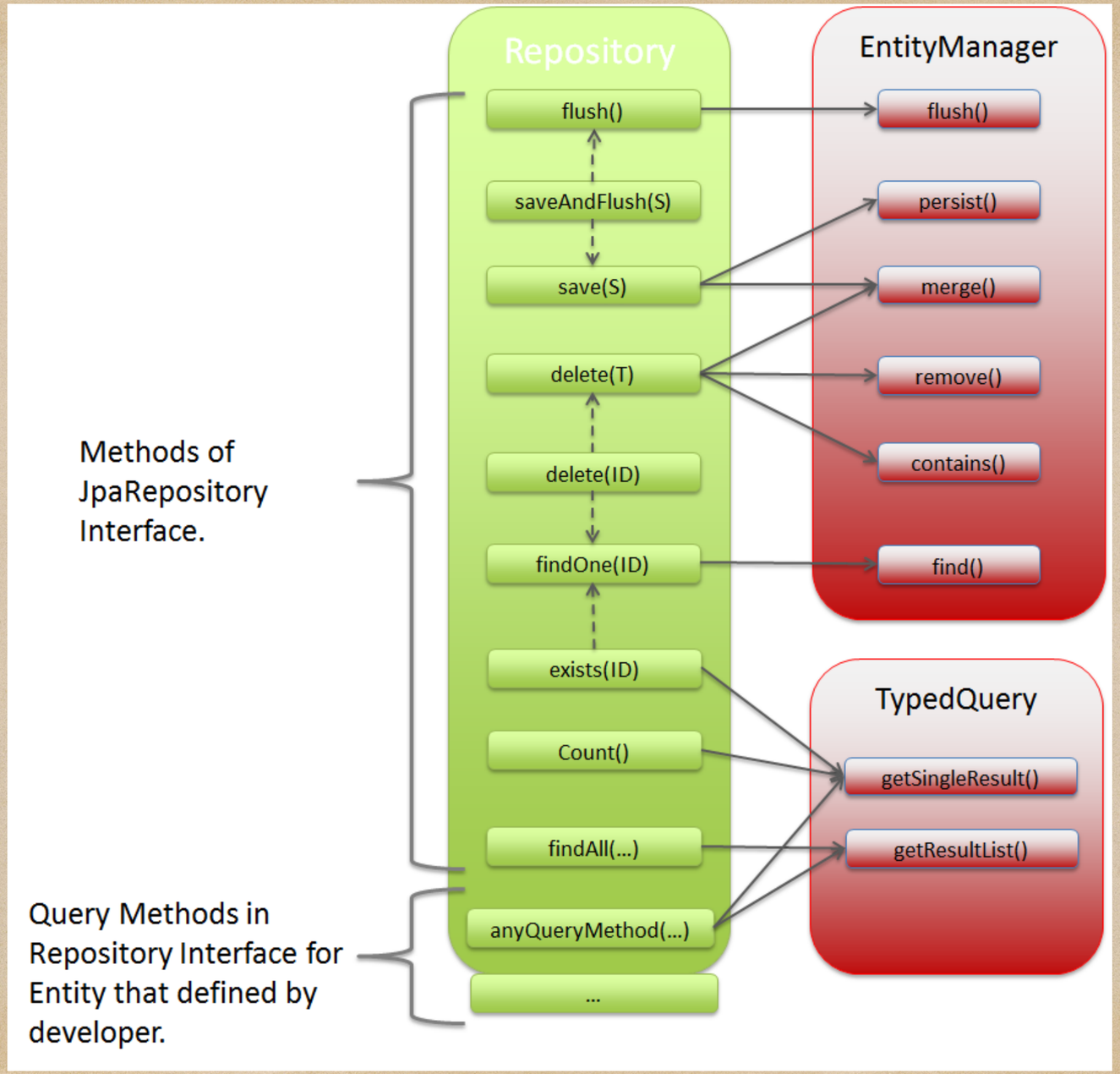
By Ramesh Fadatare (Java Guides)

```java
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```

```java
@NoRepositoryBean
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<
    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAllById(Iterable<ID> ids);

    <S extends T> List<S> saveAll(Iterable<S> entities);

    void flush();

    <S extends T> S saveAndFlush(S entity);

    <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);

    Deprecated
    @Deprecated
    default void deleteInBatch(Iterable<T> entities) { this.deleteAllInBa

    void deleteAllInBatch(Iterable<T> entities);

    void deleteAllByIdInBatch(Iterable<ID> ids);

    void deleteAllInBatch();

    Deprecated
    @Deprecated
    T getOne(ID id);

    T getById(ID id);

    <S extends T> List<S> findAll(Example<S> example);

    <S extends T> List<S> findAll(Example<S> example, Sort sort);
}
```

```java
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```


Repository
flush()
saveAndFlush(S)
save(S)
delete(T)
delete(ID)
findOne(ID)
exists(ID)
Count()
findAll(...)
anyQueryMethod(...)
...
EntityManager
flush()
persist()
merge()
remove()
contains()
find()
TypedQuery
getSingleResult()
getResultList()
Methods of JpaRepository Interface.
Query Methods in Repository Interface for Entity that defined by developer.

# save() method

As the name depicts, the save() method allows us to save an entity to the DB.

Saving an entity can be performed with the CrudRepository.save(…) method. It persists or merges the given entity by using the underlying JPA EntityManager. If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the entityManager.persist(…) method. Otherwise, it calls the entityManager.merge(…) method.

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# Update existing entity using save() method

As the name depicts, the save() method allows us to save an entity to the DB.

Saving an entity can be performed with the CrudRepository.save(…) method. It persists or merges the given entity by using the underlying JPA EntityManager. If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the entityManager.persist(…) method. Otherwise, it calls the entityManager.merge(…) method.

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# saveAll() method

As the name depicts, the saveAll() method allows us to save multiple entities to the DB.

It belongs to the CrudRepository interface defined by Spring Data.

saveAll() method returns a list of Iterable objects

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# findById() method

As the name depicts, the findById() method allows us to get or retrieve an entity based on a given id (primary key) from the DB.

It belongs to the CrudRepository interface defined by Spring Data.

findById() method returns Optional of type Entity

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# findAll() method

As the name depicts, the findAll() method allows us to get or retrieve all the entities from the database table.

It belongs to the CrudRepository interface defined by Spring Data.

findById() method returns List of Iterable objects

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# deleteById() method

As the name depicts, the deleteById() method allows us to delete an entity by id from the database table.

It belongs to the CrudRepository interface defined by Spring Data.

deleteById() method returns void (nothing)

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# delete() method

As the name depicts, the delete() method allows us to delete an entity from the database table.

It belongs to the CrudRepository interface defined by Spring Data.

delete() method returns void (nothing)

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# deleteAll() two overloaded methods

As the name depicts, the deleteAll() method allows us to delete all the entities from the database table.

It belongs to the CrudRepository interface defined by Spring Data.

deleteAll() method returns void (nothing)

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# existsById() method

As the name depicts, the existsById() method allows us to check if the entity exists with a given id in a database table.

It belongs to the CrudRepository interface defined by Spring Data.

existsById() method returns boolean (true or false)

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# count() method

As the name depicts, the count() method allows us to count the number of records that exist in a database table.

It belongs to the CrudRepository interface defined by Spring Data.

count() method returns long (numbers of records)

```java
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> S save(S entity);

    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);

    boolean existsById(ID id);

    Iterable<T> findAll();

    Iterable<T> findAllById(Iterable<ID> ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAllById(Iterable<? extends ID> ids);

    void deleteAll(Iterable<? extends T> entities);

    void deleteAll();
}
```

# Creating Database Queries From Method Names

By Ramesh Fadatare (Java Guides)

# Query generation from method names

Spring Data JPA query methods are the most powerful methods, we can create query methods to select records from the database without writing SQL queries. Behind the scenes, Spring Data JPA will create SQL queries based on the query method and execute the query for us.

findByName(String name)

```
select id, name, description, active, image_url, price, sku from products where name="product name"
```

We can create query methods for repository using Entity fields

Creating query methods is also called finder methods ( findBy, findAll …)

# Query creation/generation from method names works

We write Query Method using Spring Data JPA

```
public interface UserRepository extends Repository<User, Long> {

  List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
}
```

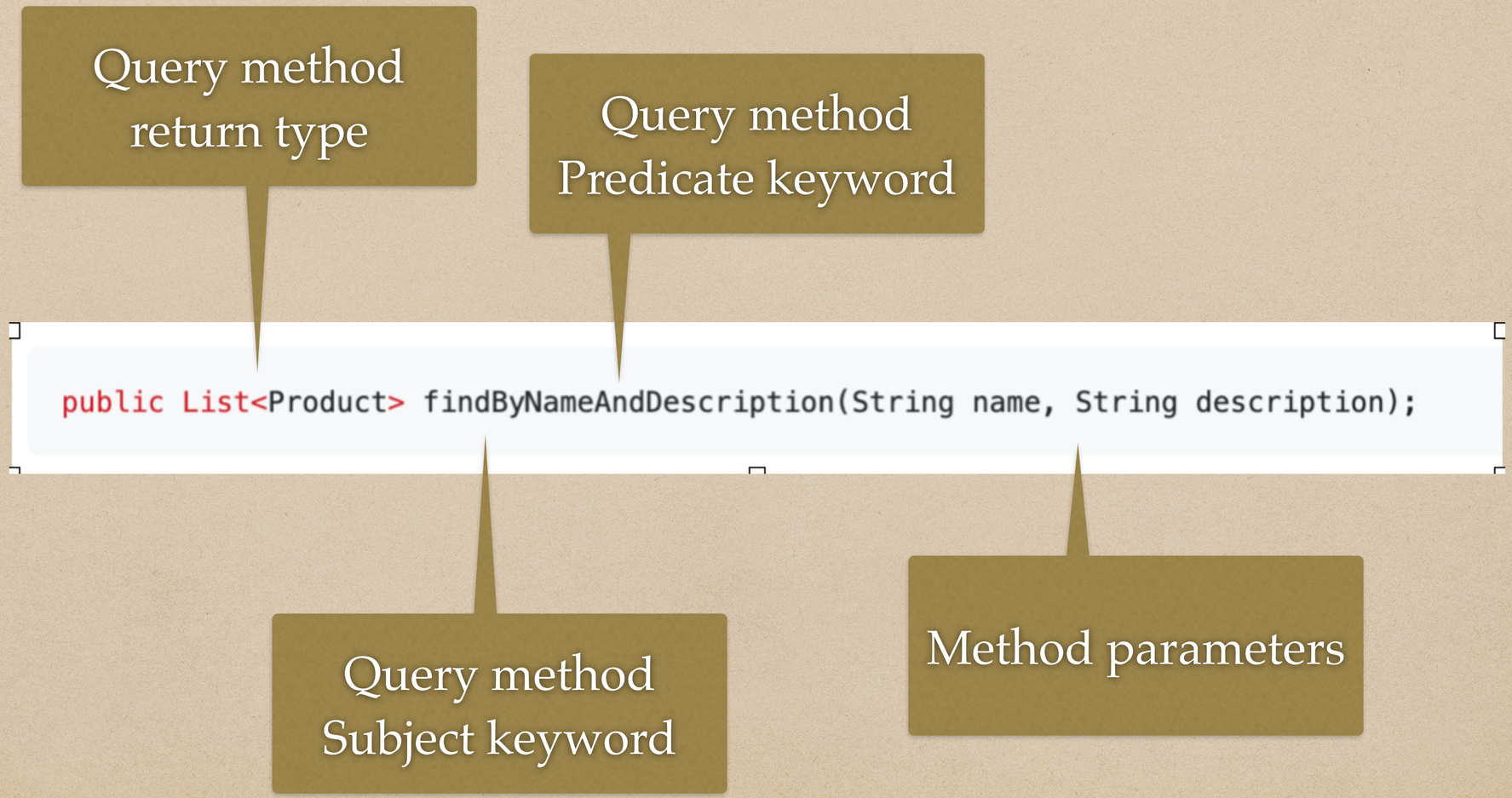
Spring Data JPA parse Query method and creates JPA Criteria

Parsing query method names is divided into subject and predicate. The first part (find…By, exists…By) defines the subject of the query, the second part forms the predicate.

JPA criteria creates JPQL query and executes it

```
select u from User u where u.emailAddress = ?1 and u.lastname = ?2
```

# Query Method Structure



# Rules to create query methods

1. The name of our query method must start with one of the following prefixes: find…By, read…By, query…By, count…By, and get…By.

Examples: findByName, readByName, queryByName, getByName

2. If we want to limit the number of returned query results, we can add the First or the Top keyword before the first By word.

Examples: findFirstByName, readFirst2ByName, findTop10ByName

3. If we want to select unique results, we have to add the Distinct keyword before the first By word.

Examples: findDistinctByName or findNameDistinctBy

4. Combine property expression with AND and OR.

Examples: findByNameOrDescription, findByNameAndDescription

# Returning Values From Query Methods

**A query method can return only one result or more than one result.**

**1. if we are writing a query that should return only one result, we can return the following types:**

- ***Basic type*. Our query method will return the found basic type or *null*.**
- ***Entity*. Our query method will return an entity object or *null*.**
- **Guava / Java 8 *Optional<T>*. Our query method will return an *Optional* that contains the found object or an empty *Optional*.**

```
public Product findByName(String name);
public Optional<Product> findById(long id);
```

**2. if we are writing a query method that should return more than one result, we can return the following types:**

- ***List<T>*. Our query method will return a list that contains the query results or an empty list.**
- ***Stream<T>*. Our query method will return a *Stream* that can be used to access the query results or an empty *Stream*.**

```
List<Product> findByPriceGreaterThan(BigDecimal price);
Stream<Product> findByPriceLessThan(BigDecimal price);
```

# Query or finder methods examples

```java
@Entity
@Table(name="products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
            generator = "product_generator")
    @SequenceGenerator(name = "product_generator",
            sequenceName = "product_sequence_name",
            allocationSize = 1)

    @Column(name = "id")
    private Long id;

    @Column(name = "sku")
    private String sku;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private BigDecimal price;
```

```java
public interface ProductRepository extends JpaRepository<Product, Long> {
    public Optional<Product> findById(long id);
    public List<Product> findByNameOrDescription(String name, String description);
    public Product findDistinctByName(String title);
    List<Product> findByPriceGreaterThan(BigDecimal price);
    List<Product> findByPriceLessThan(BigDecimal price);
    List<Product> findByNameContaining(String name);
    List<Product> findByDateCreatedBetween(LocalDateTime startDate, LocalDateTime endDate);
    List<Product> findByPriceBetween(BigDecimal startPrice, BigDecimal endPrice);
    List<Product> findByNameLike(String name);
    List<Product> findByNameIn(List<String> names);
    List<Product> findFirst2ByName(String name);
}
```

# Query method - Find by single field name

**Write a query method to find or retrieve a product by name**

```
select * from products where name = "product 1";
```

```
public Product findByName(String name);
```

**Write a query method to find or retrieve a product by id**

```
select * from products where id = 1;
```

```
public Optional<Product> findById(long id);
```

# Query method - Find by multiple field names

**Write a query method to find or retrieve a product by name or description**

```
select * from products where name = "product 1" or description = "product 1 desc";
```

```
public List<Product> findByNameOrDescription(String name, String description);
```

**Write a query method to find or retrieve a product by name and description**

```
select * from products where name = "product 1" and description = "product 1 desc";
```

```
public List<Product> findByNameAndDescription(String name, String description);
```

# Query method - Find by Distinct

**Write a query method to find or retrieve a unique product by name**

```
select distinct id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
   name="product 1";
```

```
public Product findDistinctByName(String title);
```

# Query method - Find by GreaterThan

**Write a query method to find or retrieve products whose price is greater than given price as method parameter**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
  price > 100;
```

```
List<Product> findByPriceGreaterThan(BigDecimal price);
```

# Query method - Find by LessThan

**Write a query method to find or retrieve products whose price is less than given price as method parameter**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
  price < 200;
```

```
List<Product> findByPriceLessThan(BigDecimal price);
```

# Query method - Find by Containing

**Write a query method to find or retrieve filtered products that match the given text ( contains check)**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
  name like '%product%';
```

```
List<Product> findByNameContaining(String name);
```

# Query method - Find by Like

**Write a query method to find or retrieve products for a specified pattern in a column ( SQL LIKE condition)**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
  name like '%product%';
```

```
List<Product> findByNameLike(String name);
```

# Query method - Between

**Write a query method to find or retrieve products based on the price range ( start price and end price)**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
   price between 100 and 300
```

```
List<Product> findByPriceBetween(BigDecimal startPrice, BigDecimal endPrice);
```

# Query method - Between

**Write a query method to find or retrieve products based on the start date and end date**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
   date_created between '2022-01-28 22:54:15' and '2022-01-28 22:59:23'
```

```
List<Product> findByDateCreatedBetween(LocalDateTime startDate, LocalDateTime endDate);
```

# Query method - In

**Write a query method to find or retrieve products based on multiple values ( specify multiple values in a SQL where clause)**

```
select id, active, date_created, description,image_url,last_updated,name, price,sku
from
   products
where
   name in ('product 1', 'product 2')
```

```
List<Product> findByNameIn(List<String> names);
```

# Query method - Limiting Query Results

Spring Data JPA supports keywords 'first' or 'top' to limit the query results.
Example: findFirstByName(), findTop5BySku()

An optional numeric value can be appended after 'top' or 'first' to limit the maximum number of results to be returned (e.g. findTop3By....). If this number is not used then only one entity is returned.

There's no difference between the keywords 'first' and 'top'.

# When Should We Use Query Methods

**This query generation strategy has the following benefits:**

- **Creating simple queries is fast.**
- **The method name of our query method describes the selected value(s) and the used search condition(s).**

**This query generation strategy has the following weaknesses:**

- **The features of the method name parser determine what kind of queries we can create. If the method name parser doesn't support the required keyword, we cannot use this strategy.**
- **The method names of complex query methods are long and ugly.**
- **There is no support for dynamic queries.**

# Creating Database Queries With Named Queries

By Ramesh Fadatare (Java Guides)

# Named Queries

- Named queries are one of the core concepts in JPA. They enable you to declare a query in your persistence layer and reference it in your business code. That makes it easy to reuse an existing query.

```
@Entity
@NamedQuery(name = "Product.findBySku",
            query = "SELECT p from Product p WHERE p.sku =:sku"
)
public class Product {
}
```

- When you define a named query, you can provide a JPQL query or a native SQL query in very similar ways.
- If we want to create a JPQL query, we have to annotate our entity with the @NamedQuery annotation. If we want to create a SQL query, we have to annotate our entity with the @NamedNativeQuery annotation.

# Steps to Define Named JPQL Query

If we want to create a JPQL query, we must follow these steps:

- Annotate the entity with the @NamedQuery annotation from JPA.
- Use @NamedQuery annotation's name attribute to set name of the named query (Product.findBySku)
- Use @NamedQuery annotation's query attribute to set the JPQL query (SELECT p from Product p WHERE p.sku =:sku) as the value

```
@Entity
@NamedQuery(name = "Product.findBySku",
            query = "SELECT p from Product p WHERE p.sku =:sku"
)
public class Product {
}
```

- Use named query name in a Repository

```
// named query method
Product findBySku(@Param("sku") String sku);
```

# Define Multiple Named JPQL Queries

**If we want to create a multipleJPQL query, we must follow these steps:**

- **Annotate the entity with the @NamedQueries annotation from JPA/Hibernate.**
- **Use multiple @NamedQuery annotations from JPA/Hibernate to define a named queries**
- **Set the name of the named query as the value of the @NamedQuery annotation's name attribute.**
- **Set the JPQL query as the value of the @NamedQuery annotation's query attribute.**

```
@NamedQueries(
        {
                @NamedQuery(name = "Product.findAllOrderByNameDesc",
                        query = "SELECT p from Product p ORDER BY p.name DESC"
                ),
                @NamedQuery(name = "Product.findByPrice",
                        query = "SELECT p from Product p WHERE p.price =:price"
                )
        }
)
```

# Steps to Define Named SQL Query

**If we want to create a SQL query, we must follow these steps:**

- **Annotate the entity with the @NamedNativeQuery annotation from JPA.**
- **Set the name of the named query as the value of the @NamedNativeQuery annotation's name attribute.**
- **Set the SQL query as the value of the @NamedNativeQuery annotation's query attribute.**
- **Set the returned entity class (Product.class) as the value of the of the @NamedNativeQuery annotation's resultClass attribute.**

```
@NamedNativeQuery(
        name = "Product.findBySku",
        query = "SELECT * from PRODUCTS WHERE sku =:sku",
        resultClass = Product.class
)
```

Using named native query in Repository

```
// named native query method
@Query(nativeQuery = true)
Product findBySku(@Param("sku") String sku);
```

# Define Multiple Named SQL Queries

**If we want to create a multipleJPQL query, we must follow these steps:**

- **Annotate the entity with the @NamedNativeQueries annotation from JPA/Hibernate.**
- **Use multiple @NamedNativeQuery annotations from JPA/Hibernate to define a named native queries**
- **Set the values to name, query and resultClass attributes of @NamedNativeQuery annotation**

# Creating Database Queries With the @Query Annotation

By Ramesh Fadatare (Java Guides)

# Problem with Query Methods

- **Keyword support - If the method name parser doesn’t support the required keyword, we cannot use this strategy.**
- **The method names of complex query methods are long and ugly.**

```
List<Product> findByDescriptionContainsOrNameContainsAllIgnoreCase(String description, String name);
```

- **And this is for *just two* parameters. What happens when you want to create a query for 5 parameters?**
- **This is the point when you'll most likely want to prefer to write your own queries. This is doable via the @Query annotation.**

# Understanding @Query Annotation

- We can configure the invoked database query by annotating the query method with the @Query annotation.

```
// Define JPQL query with index parameters
@Query("select p from Product p where p.name = ?1 or p.description = ?2")
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

- No need to follow query method naming conventions.

- We can use the @Query annotation in Spring Data JPA to execute both JPQL and native SQL queries

# Spring Data JPA @Query annotation works

Define JPQL or Native SQL query using @Query annotation

```
// Define JPQL query with index parameters
@Query("select p from Product p where p.name = ?1 or p.description = ?2")
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

Spring Data JPA provides required JPA code to execute the  statement as a JPQL or native SQL query

Reduces boilerplate code

Hibernate will execute the query and map the result.

Your preferred JPA implementation, e.g., Hibernate or EclipseLink, will then execute the query and map the result.

# Quick Overview of JPQL

JPQL stands for the Java Persistence Query Language. It is defined in the JPA specification and is an object-oriented query language used to perform database operations on persistent entities.

Hibernate, or any other JPA implementation, has to transform the JPQL query into SQL.

The syntax of a JPQL FROM clause is similar to SQL but uses the entity model instead of table or column names.

```
// Define JPQL query with index parameters
@Query("select p from Product p where p.name = ?1 or p.description = ?2")
Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
```

```
@Entity
@Table(name="products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE,
            generator = "product_generator")
    @SequenceGenerator(name = "product_generator",
            sequenceName = "product_sequence_name",
            allocationSize = 1)

    @Column(name = "id")
    private Long id;

    @Column(name = "sku")
    private String sku;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @Column(name = "price")
    private BigDecimal price;
```

# JPQL Features

- It is a platform-independent query language.
- It is simple and robust.
- It can be used with any type of relation database.
- It can be declared statically into metadata or can also be dynamically built in code.
- It is case insensitive.

If your database can change or varies from development to production, as long as they're both relational - JPQL works wonders and you can write JPQL queries to create generic logic that can be used over and over again.

# Creating JPQL Queries

**Steps to create JPQL query with the @Query annotation:**

Step 2: Annotate the query method with the @Query annotation, and specify the invoked query by setting it as the value of the @Query annotation.

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define JPQL query with index parameters
    @Query("select p from Product p where p.name = ?1 or p.description = ?2")
    Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
}
```

Step 1: Add a query method to our repository interface.

# JPQL Query with Index (Position) Parameters

When using position-based parameters, you have to keep track of the order in which you supply the parameters in:

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define JPQL query with index parameters
    @Query("select p from Product p where p.name = ?1 or p.description = ?2")
    Product findByNameOrDescriptionJPQLIndexParam(String name, String description);
}
```

The first parameter passed to the method is mapped to ?1, the second is mapped to ?2, etc. If you accidentally switch these up - your query will likely throw an exception, or silently produce wrong results.

# JPQL Query with Named Parameters

**Named parameters can be referenced by name, no matter their position:**

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define JPQL query with named parameters
    @Query("select p from Product p where p.name =:name or p.description =:description")
    Product findByNameOrDescriptionJPQLNamedParam(@Param("name") String name,
                                                  @Param("description") String description);
}
```

**The name within the @Param annotation is matched to the named parameters in the @Query annotation, so you're free to call your variables however you'd like - but for consistency's sake - it's advised to use the same name.**

# Creating Native SQL Queries

**Steps to create Native SQL query with the @Query annotation:**

Step 2: Annotate the query method with the @Query annotation, and specify the invoked query by setting it as the value of the @Query annotation.

Step 3: Set the value of the @Query annotation's nativeQuery attribute to true

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define Native SQL query with index parameters
    @Query(value = "select * from products p where p.name =?1 " +
            "or p.description =?2", nativeQuery = true)
    Product findByNameOrDescriptionSQLIndexParam(String name, String description);
}
```

Step 1: Add a query method to our repository interface.

# Native SQL Query with Index (Position) Parameters

**When using position-based parameters, you have to keep track of the order in which you supply the parameters in:**

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define Native SQL query with index parameters
    @Query(value = "select * from products p where p.name =?1 " +
            "or p.description =?2", nativeQuery = true)
    Product findByNameOrDescriptionSQLIndexParam(String name, String description);
}
```

**The first parameter passed to the method is mapped to ?1, the second is mapped to ?2, etc. If you accidentally switch these up - your query will likely throw an exception, or silently produce wrong results.**

# Native SQL Query with Named Parameters

**Named parameters can be referenced by name, no matter their position:**

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    // Define Native SQL query with named parameters
    @Query(value = "select * from products p where p.name =:name " +
            "or p.description =:description", nativeQuery = true)
    Product findByNameOrDescriptionSQLNamedParam(@Param("name") String name,
                                                 @Param("description") String description);
}
```

**The name within the @Param annotation is matched to the named parameters in the @Query annotation, so you're free to call your variables however you'd like - but for consistency's sake - it's advised to use the same name.**

# Spring Data JPA Pagination and Sorting

By Ramesh Fadatare (Java Guides)

# Pagination and Sorting Overview

**As you know, pagination allows the users to see a small portion of data at a time (a page), and sorting allows the users to view the data in a more organized way.**

**Paging and sorting is mostly required when we are displaying domain data in tabular format in UI.**

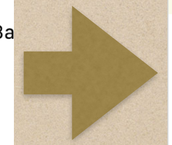
**Pagination consist of two fields – page size and page number.**

**Sorting consist of two fields - sortBy (single or multiple fields) and sortDir ( sort direction can be ASC or DESC)**

# Spring Data JPA Pagination and Sorting

```java
public interface ProductRepository extends JpaRepository<Product, Long> {

}
```

```java
@NoRepositoryBean
public interface JpaRepository<T, ID> extends PagingAndSortingRepository<
    List<T> findAll();

    List<T> findAll(Sort sort);

    List<T> findAllById(Iterable<ID> ids);

    <S extends T> List<S> saveAll(Iterable<S> entities);

    void flush();

    <S extends T> S saveAndFlush(S entity);

    <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);

    Deprecated
    @Deprecated
    default void deleteInBatch(Iterable<T> entities) { this.deleteAllInBa

    void deleteAllInBatch(Iterable<T> entities);

    void deleteAllByIdInBatch(Iterable<ID> ids);

    void deleteAllInBatch();

    Deprecated
    @Deprecated
    T getOne(ID id);

    T getById(ID id);

    <S extends T> List<S> findAll(Example<S> example);

    <S extends T> List<S> findAll(Example<S> example, Sort sort);
}
```



```java
@NoRepositoryBean
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```

# Spring Data JPA Pagination

**To apply only pagination in result set, we need to create Pageable object without any Sort information and pass Pageable object to findAll() method:**

```
int pageNo = 0;
int pageSize = 3;
Pageable pageable = PageRequest.of(pageNo, pageSize);
Page<Product> page = productRepository.findAll(pageable);

List<Product> productList = page.getContent();
productList.forEach((p) ->{
    System.out.println(p);
});

int totalPages = page.getTotalPages();
long totalItems = page.getTotalElements();
int size = page.getSize();
int numberOfElements = page.getNumberOfElements();
boolean isLast = page.isLast();
boolean isfirst = page.isFirst();
```

# Spring Data JPA Sorting

**To apply only sorting in result set, we need create Sort object and pass it to findAll() method**

```
String sortBy = "name";
String sortDir = "desc";
Sort sort = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortBy).ascending()
        : Sort.by(sortBy).descending();

List<Product> sortedProducts = productRepository.findAll(sort);
```

# Spring Data JPA Sorting By Multiple Fields

**If we wish to apply sorting on multiple columns or group by sort, then that is also possible by creating Sort using simple builder pattern steps.**

```
String sortBy = "name";
String sortByDesc = "description";
// default sorting ascending order: DEFAULT_DIRECTION = Sort.Direction.ASC;
String sortDir = "desc";
Sort sortName = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortBy).ascending()
        : Sort.by(sortBy).descending();

Sort sortDesc = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortBy).ascending()
        : Sort.by(sortBy).descending();

// sorting on multiple columns or group by sort,
Sort groupBySort = sortName.and(sortDesc);

List<Product> sortedProducts = productRepository.findAll(groupBySort);
```

# Spring Data JPA
# Pagination and Sorting Together

```
String sortBy = "name";
// default sorting ascending order: DEFAULT_DIRECTION = Sort.Direction.ASC;
String sortDir = "asc";
int pageNo = 0;
int pageSize = 3;

Sort sort = sortDir.equalsIgnoreCase(Sort.Direction.ASC.name()) ? Sort.by(sortBy).ascending()
        : Sort.by(sortBy).descending();

Pageable pageable1 = PageRequest.of(pageNo, pageSize, sort);

Page<Product> listOfProducts = productRepository.findAll(pageable1);
```