

## Repository Pattern and Query Methods

The Repository design pattern acts as a central hub for managing all Java data access logic, abstracting the details of data storage and retrieval from the rest of the application.

### Real-world example

Imagine a library system where a librarian acts as the repository. Instead of each library patron searching through the entire library for a book (the data), they go to the librarian (the repository) who knows exactly where each book is located, regardless of whether it's on a shelf, in the storeroom, or borrowed by someone else. The librarian abstracts the complexities of book storage, allowing patrons to request books without needing to understand the storage system. This setup simplifies the process for patrons (clients) and centralizes the management of books (data access logic).

Repository is one of the easiest and important design pattern that you can use and see, especially when you need a layer to deal with data access whether this data is in a database or another storage.

### Where to Use;

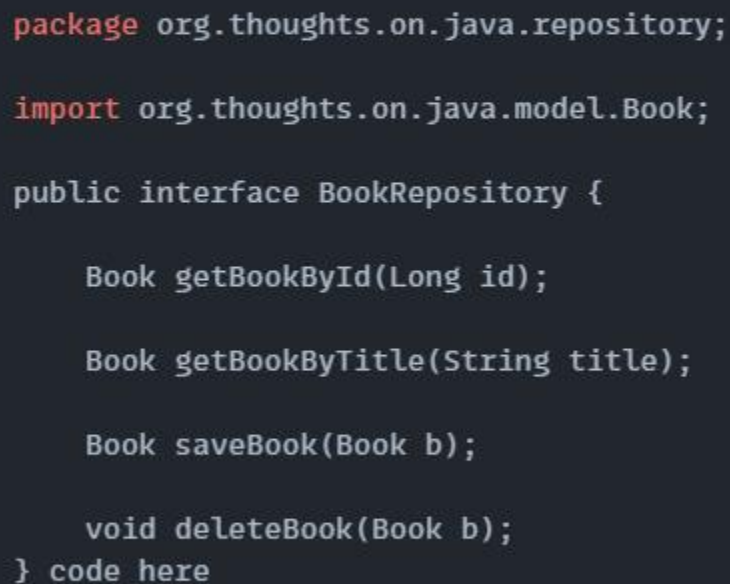
- Between domains(entity) and Data storage
- To prevent duplicate query
- In a system where you have a lot of heavy query
- It is used for search or remove element using specification of the entity that the repository created for.

### Defining the repository interface

Let's implement the same *BookRepository* interface as I showed you in the diagram. It defines 4 methods that you can use to:

- save a new or changed entity (Please keep in mind that Hibernate detects and persists all changes of managed entities automatically. So, you don't need to call the save method after you changed any entity attributes),
- delete an entity,

- find an entity by its primary key and
- find an entity by its title.



```
package org.thoughts.on.java.repository;

import org.thoughts.on.java.model.Book;

public interface BookRepository {

    Book getBookById(Long id);

    Book getBookByTitle(String title);

    Book saveBook(Book b);

    void deleteBook(Book b);
} code here
```

snappify.com

## Implementing the repository with JPA and Hibernate

In the next step, you can implement the *BookRepository* interface. In this example, I only create a simple JPA-based implementation, that doesn't rely on any other frameworks.

```

package org.thoughts.on.java.repository;

import javax.persistence.EntityManager;
import javax.persistence.TypedQuery;

import org.thoughts.on.java.model.Book;

public class BookRepositoryImpl implements BookRepository {

    private EntityManager em;

    public BookRepositoryImpl(EntityManager em) {
        this.em = em;
    }

    @Override
    public Book getBookById(Long id) {
        return em.find(Book.class, id);
    }

    @Override
    public Book getBookByTitle(String title) {
        TypedQuery<Book> q = em.createQuery("SELECT b FROM Book b WHERE b.title = :title", Book.class);
        q.setParameter("title", title);
        return q.getSingleResult();
    }

    @Override
    public Book saveBook(Book b) {
        if (b.getId() == null) {
            em.persist(b);
        } else {
            b = em.merge(b);
        }
        return b;
    }

    @Override
    public void deleteBook(Book b) {
        if (em.contains(b)) {
            em.remove(b);
        } else {
            em.merge(b);
        }
    }
}

```

snappify.com

## CrudRepository

The CrudRepository is a core interface in Spring Data that provides basic Create, Read, Update, and Delete (CRUD) operations for a specific entity. It allows developers to perform fundamental data manipulation tasks, such as:

- **Save** an entity

- **Find** an entity by its ID
- **Find all** entities
- **Delete** an entity by its ID
- **Count** the total number of entities

By extending the `CrudRepository` interface, Spring Data automatically provides implementations of these common data access methods, reducing boilerplate code and simplifying data management.

#### PagingAndSortingRepository

The `PagingAndSortingRepository` extends the `CrudRepository` interface to offer additional functionality for handling large datasets. It provides methods for **pagination** and **sorting** of entities, which are essential for efficiently retrieving and managing subsets of data. This interface adds two key methods:

- **Pagination:** Allows retrieving a specific "page" of results, which is useful when working with large datasets.
- **Sorting:** Enables sorting the data based on specified properties, which can be helpful for ordering results in a particular manner.

#### Query Methods

Query methods in Spring Boot are a powerful way to interact with your database using Spring Data JPA. They allow you to define custom queries directly in your repository interfaces without writing SQL or JPQL. This is made possible by the method naming conventions in Spring Data, which automatically generate the necessary queries based on the method name.

#### Key Concepts of Query Methods

1. **Derived Query Methods:** These methods are automatically generated based on the method name. Spring Data JPA parses the method name and creates a query accordingly.
2. **Query Creation from Method Names:** Spring Data JPA supports various keywords that you can use in your method names to create queries. Some of the commonly used keywords are:

- And, Or: Combines two conditions.
  - Between: For range queries.
  - LessThan, GreaterThan: For comparison queries.
  - Like, NotLike: For pattern matching.
  - In, NotIn: For checking within a collection.
  - OrderBy: For sorting the results.
3. **@Query Annotation:** If the method name becomes too complex or you need a more specific query, you can use the @Query annotation to write JPQL (Java Persistence Query Language) or native SQL queries directly.
- **Example (JPQL):**
4. **Pagination and Sorting:** Spring Data JPA allows you to add pagination and sorting to your queries. You can pass Pageable or Sort as a parameter to your query methods.
5. **Named Queries:** These are queries that are defined in the @Entity class using the @NamedQuery annotation. They can be referenced in repository methods by their name.

### Advantages of Using Query Methods

- **Simplicity:** You don't need to write complex SQL queries; Spring Data JPA does the heavy lifting.
- **Type Safety:** Since query methods are defined in the interface, they are type-safe.
- **Maintainability:** Method names can be more descriptive and easier to understand than complex SQL.
- **Flexibility:** You can still write custom queries using @Query if necessary.