

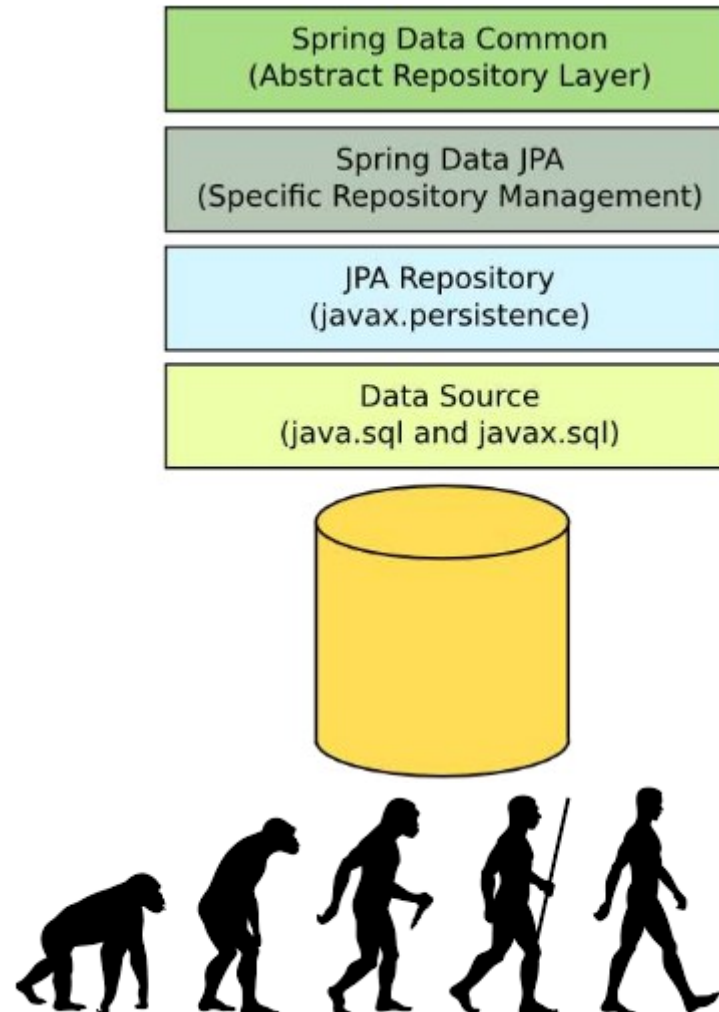
Spring data

Rajeev Gupta

Mtech CS

Java Trainer & Consultant

Spring data evaluation



spring data

spring hib

hib

jdbc

Jdbc

- The JDBC API makes it possible to
- do three things:
 - Establish a connection with a
 - database or access any tabulardata source,Send SQL statements
 - Process the results

```
Connection conn = null;  
Statement stmt = null;  
Class.forName("com.mysql.jdbc.Driver");  
conn = DriverManager.getConnection("jdbc:mysql://localhost/EMP",  
"username", "password");  
stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT id, first FROM Employees");  
while (rs.next()) {  
    System.out.println("ID: " + rs.getInt("id")  
        + ", First: " + rs.getString("first"));  
}  
rs.close();  
stmt.close();  
conn.close();
```

JPA

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private long id;
    @Column(name = "date")
    private Date date;
    @ManyToOne
    @JoinColumn(name = "product_id")
    private Product product;
}
```

```
Product product = new Product();
Order order = new Order();
order.setProduct(product);
```

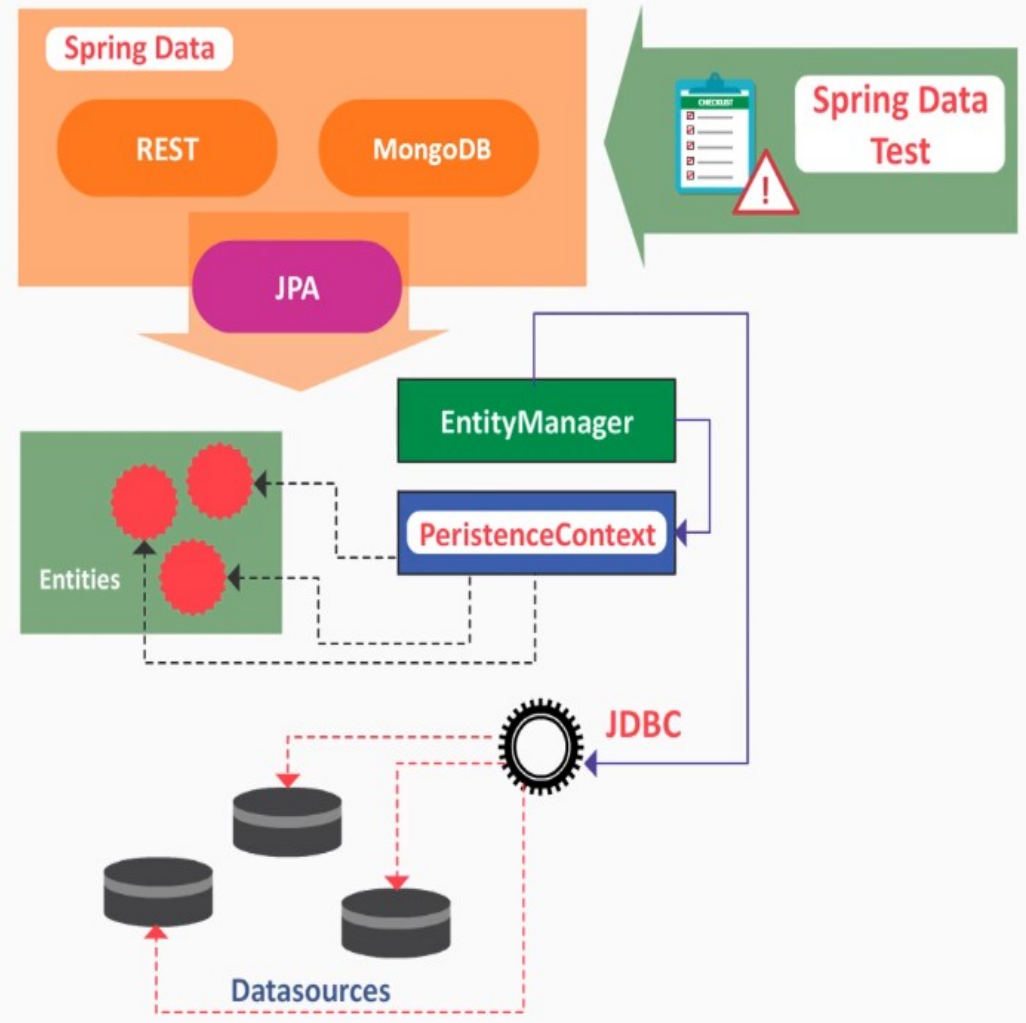
```
entityManager.persist(product);
entityManager.persist(order);
```

```
Product dbProduct = entityManager.find(Product.class, product.getId());
```

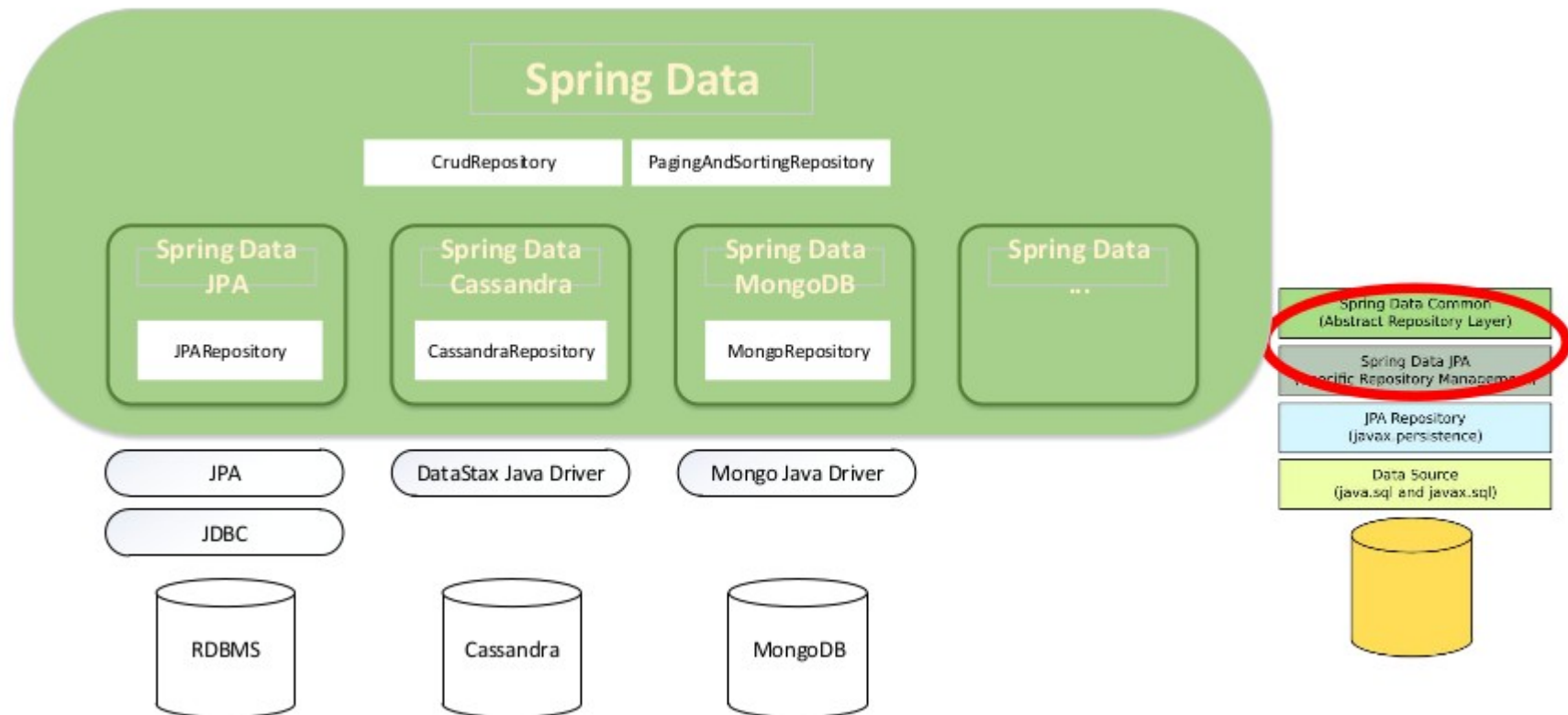
```
String name = "InitialName";
String query = "SELECT p FROM Product p WHERE p.name LIKE :productName";
Product product = entityManager.createQuery(query, Product.class)
    .setParameter("productName", name)
    .getSingleResult();
```

Introduction

- Spring Data is a framework composed of different modules that support easy access to data repository and cloud services
- Spring Data JPA is a module of Spring Data that avoids boilerplate data repository implementation



Spring data



JpaRepository vs CrudRepository

rdbms specific

**Generics concept
supported by rdbms
no-sql**

JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository.

Their main functions are:

- ✓ CrudRepository mainly provides CRUD functions.
- PagingAndSortingRepository provides methods to do pagination and sorting records.
- ✓ JpaRepository provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.

Because of the inheritance mentioned above, JpaRepository will have all the functions of CrudRepository and PagingAndSortingRepository. So if you don't need the repository to have the functions provided by JpaRepository and PagingAndSortingRepository, use CrudRepository.

JpaRepository vs CrudRepository

Basics

The base interface you choose for your repository has two main purposes. First, you allow the Spring Data repository infrastructure to find your interface and trigger the proxy creation so that you inject instances of the interface into clients. The second purpose is to pull in as much functionality as needed into the interface without having to declare extra methods.

The common interfaces

The Spring Data core library ships with two base interfaces that expose a dedicated set of functionalities:

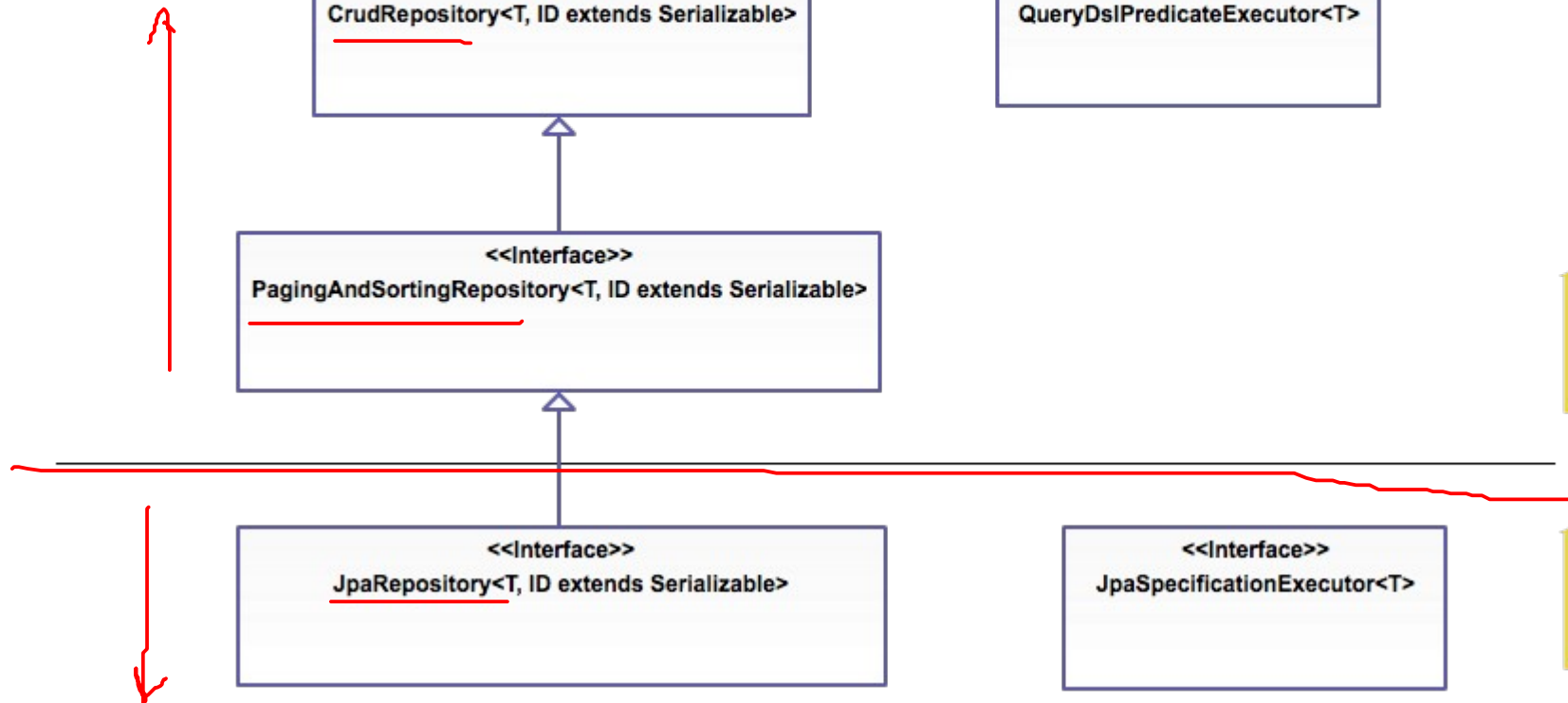
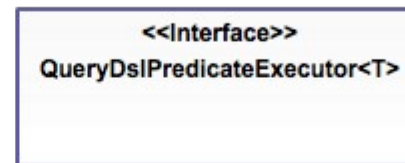
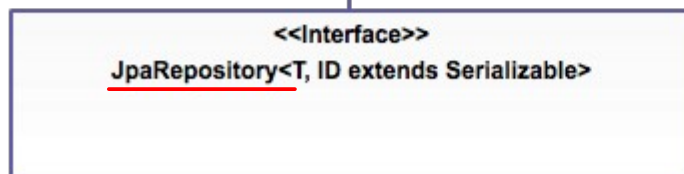
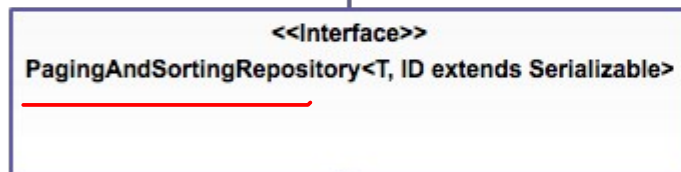
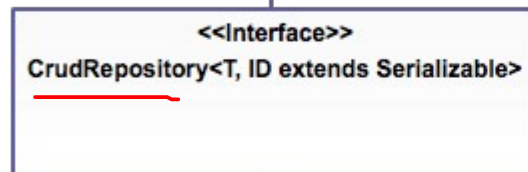
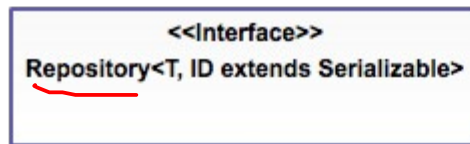
- `CrudRepository` - CRUD methods
- `PagingAndSortingRepository` - methods for pagination and sorting (extends `CrudRepository`)

JpaRepository vs CrudRepository

Store-specific interfaces

The individual store modules (e.g. for JPA or MongoDB) expose store-specific extensions of these base interfaces to allow access to store-specific functionality like flushing or dedicated batching that take some store specifics into account. An example for this is `deleteInBatch(...)` of `JpaRepository` which is different from `delete(...)` as it uses a query to delete the given entities which is more performant but comes with the side effect of not triggering the JPA-defined cascades (as the spec defines it).

We generally recommend *not* to use these base interfaces as they expose the underlying persistence technology to the clients and thus tighten the coupling between them and the repository. Plus, you get a bit away from the original definition of a repository which is basically "a collection of entities". So if you can, stay with `PagingAndSortingRepository`.



```
import java.time.LocalDate;

@Entity
@Table(name="book_table")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long bookId;
    private String title;
    private LocalDate publishDate;
    private int pageCount;
    private double price;
}
```

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long>{
}
```

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/boot_demo?useSSL=false
spring.jpa.show-sql=true
logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR
```

crud examples

```
bookRepo.deleteById(1L);

bookRepo.delete(bookRepo.findById(2L).orElseThrow(RuntimeException::new));

bookRepo.deleteAll((bookRepo.findAllById((new ArrayList<Long>()){
    add(3L);
    add(4L);
})))));

bookRepo.deleteInBatch(bookRepo.findAllById(new ArrayList<Long>()){
    add(5L);
    add(6L);
}));

bookRepo.deleteAll();

bookRepo.deleteAllInBatch();
```

//update records

```
Book book = bookRepo.findById(1L).orElseThrow(RuntimeException::new);
book.setTitle("java in action");
bookRepo.save(book);
```

Turning off repo definations

- lets say our requirment is to create an safe repo, only for reading purpose
- use @NoRepositoryBean, Spring jpa will only give implementation for methods that are mentioned in interface!

```
@NoRepositoryBean
public interface ReadOnlyRepository<T, ID extends Serializable> extends
    Repository<T, ID> {

    T findOne(ID id);

    Iterable<T> findAll();
}
```

Derived queries from method names

- Spring Data can derive queries based on method name declared into the repo interface

```
class Person{  
    private firstName;  
}
```

```
public interface PersonRepository extends CrudRepository<Person, String>{  
    public List<Person> findByFirstName(String firstName);
```

- Query generated:
 - select p from Person p where p.firstName=?

String comparison methods:

How derived queries are constructed?

@Repository

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    //Spring data inspect method sign and find findBy keyword  
    //query builder in background remove findBy and inspect next portion  
    //"title" (title of POJO)  
    //using this info for spring data enable to construct queries....  
  
    public Book findByTitle(String title);  
}
```

Examples

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long>{
    |
    public List<Book> findByTitle(String title);
    public List<Book> findByTitleLike(String title);

    public List<Book> findByTitleContaining(String title);
    public List<Book> findByTitleStartingWith(String title);
    public List<Book> findByTitleEndingWith(String title);
    public List<Book> findByTitleIgnoreCase(String title);
}
```

Relational comparison methods in derived queries

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    public List<Book> findByTitle(String title);
    public List<Book> findByPageCountEquals(int pageCount);
    public List<Book> findByPageCountGreaterThan(int pageCount);
    public List<Book> findByPageCountLessThan(int pageCount);
    public List<Book> findByPageCountGreaterThanEqual(int pageCount);
    public List<Book> findByPageCountLessThanEqual(int pageCount);
    public List<Book> findByPageCountBetween(int min, int max);
}

class Book{
    String authorName;
    private Author author; }
}
```

✓ findByBookAuthorAuthorName(String name)

Logical operator in derived queries (or not and)

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    public List<Book> findByTitleContainingOrTitleContaining
        (String title, String title2);

    public List<Book> findByTitleContainingAndPageCountGreaterThan
        (String title, int pageCount);

    public List<Book> findByTitleNot(String title);
}
```

Date comparision in spring data

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    public List<Book> findByPublishDateAfter(LocalDate date);

    public List<Book> findByPublishDateBefore(LocalDate date);

    public List<Book> findByPublishDateBetween(LocalDate date, LocalDate date2);
}
```

Quaries to check containing

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    public List<Book> findByTitleContainingOrderByTitleAsc(String title);

    public List<Book> findByTitleContainingOrderByTitleDesc(String title);

    public List<Book> findTopByOrderByPageCountDesc(); // find top book

    public List<Book> findFirstByOrderByPageCountAsc();

    public List<Book> findTop5ByOrderByPriceDesc(); // top 5 costly books....
}
```

JPQL data with spring data, Sort queries

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    @Query("select b from Book b")
    public List<Book> queryOne();

    @Query("select b from Book b where b.pageCount > ?1")
    public List<Book> queryTwo(int pageCount);

    @Query("select b from Book b where b.title = :title")
    public List<Book> queryThree(@Param("title") String title);
}
```

select

**book0_.book_id as book_id1_0_,
book0_.page_count as page_cou2_0_,
book0_.price as price3_0_,
book0_.publish_date as publish_4_0_,
book0_.title as title5_0_**

from

b_table book0_

where

book0_.price>=?

and book0_.~~page~~_count=?

JPQL data with spring data, Sort queries

- Query name Book.queryOne corresponding to method named queryOne() of Book class

```
@Table(name = "book_table")
@NamedQueries({
    @NamedQuery(name = "Book.queryOne", query = "select b from Book b"),
    @NamedQuery(name = "Book.queryTwo", query = "select b from Book b "
        + "where b.pageCount > ?1"),
    @NamedQuery(name = "Book.queryThree", query = "select b from Book b where"
        + " b.title = :title") })

public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)

    @Repository
    public interface BookRepo extends JpaRepository<Book, Long> {
        public List<Book> queryOne();
    }
}
```

JPQL pagination

- Pagination is imp technique for large result set to displayed to a web page
- Breeding down larger data set to sub set, pagination
- Spring provide paging and sorting repo out of the box....contain several method to support pagination

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    public List<Book> findByPageCountGreaterThan(int pageCount, Pageable pageable);
}
```

```
List<Book>books=bookRepo.findByPageCountGreaterThan(340, new PageRequest(0, 3,
    Direction.DESC, "title"));
books.forEach(book-> System.out.println(book));
```

```
List<Book> booksSortedAsPageCount=bookRepo.findAll(new Sort("pageCount"));
booksSortedAsPageCount.forEach(book-> System.out.println(book));
```

```
bookRepo.findAll(new Sort(Direction.ASC, "pageCount").and(new Sort(Direction.ASC, "price")));
```

JPQL pagination returning page/slice

- Page vs Slice
 - Slice is More efficient then Page, but do not give imp inventory information such as getTotalPages()

```
@Repository
public interface BookRepo extends JpaRepository<Book, Long> {
    //public List<Book> findByPageCountGreaterThan(int pageCount, Pageable pageable);

    //public Page<Book> findByPageCountGreaterThan(int pageCount, Pageable pageable);

    public Slice<Book> findByPageCountGreaterThan(int pageCount, Pageable pageable);
}
```

```
Page<Book> page = bookRepo.findByPageCountGreaterThan
    (200, new PageRequest(1, 2, Direction.ASC, "pageCount"));

page.forEach(p-> System.out.println(p));

//important method of page

System.out.println("no of pages:" + page.getTotalPages());
System.out.println("how many result available with subset:" + page.getTotalElements());
```


JPQL pagination returning page/slice: new syntax

```
//Demo on pagination and sorting  
  
Sort sort=Sort.by(Sort.Direction.DESC,"pageCount") ;  
  
List<Book> books=bookRepo.findAll(sort);  
  
books.forEach(b-> System.out.println(b));
```

```
Pageable pageable=PageRequest.of(0, 2);  
  
Page<Book> books=bookRepo.findAll(pageable);  
  
books.forEach(b-> System.out.println(b));
```

Pagable rest response

```
public List<Book> getAllBooksPagable(Pageable pageable);
```

```
@Override  
public List<Book> getAllBooksPagable(Pageable pageable) {  
    return dao.findAll(pageable).getContent();  
}
```

```
@GetMapping(path = "/bookpage", produces = MediaType.APPLICATION_JSON_VALUE)  
public ResponseEntity<List<Book>> getAllBooksPagination(Pageable pageable) {  
    return new ResponseEntity<List<Book>>(  
        bookService.getAllBooksPagable(pageable), HttpStatus.OK);  
}
```

localhost:8090/bookapp/api/bookpage?page=0&size=2

Apps 11 new netwo... Difference bet... Installing and... Unit and Integ... New folder Code Craftsm...

```
[{"bookId":9,"isbn":"XM108","title":"python advance","authorName":"gunika","publisher":"abc","price":900.0,"pubDate":"2013-11-11"},  
{"bookId":3,"isbn":"AM108","title":"python advance","authorName":"gunika","publisher":"abc","price":900.0,"pubDate":"2018-04-11"}]
```

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is , Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	... where x.age >= ?1
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1

IsNull, Null	findByAge(Is)Null	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parameter bound wrapped in %)

EndingWith	findByFirstnameEndingWith	<code>... where x.firstname like ?1</code> (parameter bound with prepended <code>%</code>)
Containing	findByFirstnameContaining	<code>... where x.firstname like ?1</code> (parameter bound wrapped in <code>%</code>)
OrderBy	findByAgeOrderByLastnameDesc	<code>... where x.age = ?1 order by x.lastname desc</code>
Not	findByLastnameNot	<code>... where x.lastname <> ?1</code>
In	findByAgeIn(Collection<Age> ages)	<code>... where x.age in ?1</code>
NotIn	findByAgeNotIn(Collection<Age> ages)	<code>... where x.age not in ?1</code>
True	findByActiveTrue()	<code>... where x.active = true</code>
False	findByActiveFalse()	<code>... where x.active = false</code>
IgnoreCase	findByFirstnameIgnoreCase	<code>... where UPPER(x.firstname) = UPPER(?1)</code>

Creating the @Repository Methods

Spring Boot 2.2 automatically runs `@EnableJpaRepositories` (`queryLookupStrategy=QueryLookupStrategy.Key.CREATE_IF_NOT_FOUND`), which generates the query methods if they do not exist

The query methods uses prefixes such as `findBy***`, `readBy***`, `countBy***`, `getBy***`, and `queryBy***`

These query methods are generated using some keywords, such as `Is`, `Equals`, `And`, `Or`, `Between`, `LessThan`, `GreaterThan`, `Like`, `True`, `False`, and so on

```
public interface ProfileRepository extends JpaRepository<Profile, Long>{
```

```
    public List<Profile> findByNameIgnoreCase(String name);
```

```
    public List<Profile> findByUsernameAndPassword(String username, String password);
```

```
    public List<Profile> findByUsernameStartingWith(String prefix);
```

```
    public List<Profile> findByApprovedTrue();
```

```
    @Query(value = "select * from signup", nativeQuery = true)
```

```
    public List<Profile> findProfiles();
```

```
    @Query(value = "from Profile")
```

```
    public Stream<Profile> findStreamProfiles();
```

```
    @Query(value = "from Profile p WHERE p.id = :id")
```

```
    public Profile findProfile(@Param("id") Long id);
```

```
    @Query(value = "from Profile p WHERE p.username = ?1 AND p.password = ?2")
```

```
    public List<Profile> findProfiles (String username, String password);
```

```
    @Query(value = "Select * from signup WHERE username LIKE ?1 AND password LIKE ?2", nativeQuery = true)
```

```
    public List<Profile> findNativeProfiles (String username, String password);
```



```
@Query(value = "from Profile")  
public Stream<Profile> findStreamProfiles();
```

```
@Query(value = "from Profile p WHERE p.id = :id")  
public Profile findProfile(@Param("id") Long id);
```

```
@Query(value = "from Profile p WHERE p.username = ?1 AND p.password = ?2")  
public List<Profile> findProfiles (String username, String password);
```

```
@Query(value = "Select * from signup WHERE username LIKE ?1 AND password LIKE ?2", nativeQuery = true)  
public List<Profile> findNativeProfiles (String username, String password);
```

```
@Modifying  
@Query(value = "delete from signup where username = :username ", nativeQuery = true)  
public void deleteByUsername(@Param("username") String username);
```

```
@Modifying(clearAutomatically = true)  
@Query(value = "update signup set name = ?1 where username = ?2", nativeQuery = true)  
public void updateByUsername(String name, String username);
```