



INDEX

Spring Boot Data

1. Spring Data	<u>04</u>
a. Important sub modules of Spring Data module	<u>08</u>
2. Repositories in Spring Data JPA	<u>13</u>
3. Select operations using CrudRepository	<u>25</u>
4. Delete operations using CrudRepository	<u>29</u>
5. PagingAndSortingRepository	<u>33</u>
6. JpaRepository	<u>39</u>
7. Custom Persistence operations in Spring Data JPA	<u>47</u>
a. Finder methods	<u>47</u>
b. Performing Scalar Operations or Projections using Finder methods	<u>58</u>
c. Working with @Query methods	<u>69</u>
d. Performing non-select Operations using HQL/ JPQL in @Query methods	<u>82</u>
e. Using @Query methods having Native SQL queries	<u>84</u>
f. Calling PL/ SQL Procedure and Functions using Spring Data JPA	<u>86</u>
8. Working with Date values using Java 8 Date and Time API	<u>104</u>
9. Working with Large Objects (LOBS)	<u>108</u>
10. Associations in Spring Data JPA	<u>116</u>
11. Joins in Spring Data JPA using HQL/ JPQL	<u>130</u>
12. Spring Data MongoDB	<u>134</u>
a. MongoDB Software Installation	<u>137</u>
b. Procedure to create Logical DB with collection in MongoDB using Robo3T	<u>146</u>
c. Procedure to Develop First Spring Data MongoDB Application using MongoRepository	<u>154</u>
d. Finder methods in MongoRepository/ Spring Data MongoDB	<u>161</u>
e. Association Mapping in Spring Boot MongoDB	<u>167</u>
f. MongoDB @Query method with Projections	<u>183</u>
g. Spring Boot MongoDB using MongoTemplate	<u>193</u>
13. Interacting with multiple DB s/w	<u>204</u>

Spring Boot Data

Spring Data

✚ We will discuss the following things

- Spring Data JPA
- Spring Data MongoDB

JDBC, O-R mapping

Java App -----> DB s/w (SQL DB s/w)

O-R mapping (Object-Relation Mapping):

- It is all about mapping java classes with DB tables.
- The Process of mapping java classes with DB tables and properties of java classes with columns of DB tables to make the objects of java classes representing the records of DB tables having synchronization between them is called O-R mapping.
- JDBC persistence logic is SQL queries-based DB s/w dependent persistence logic.
- The O-R mapping persistence logic is objects-based Database s/w independent persistence logic (No SQL queries).

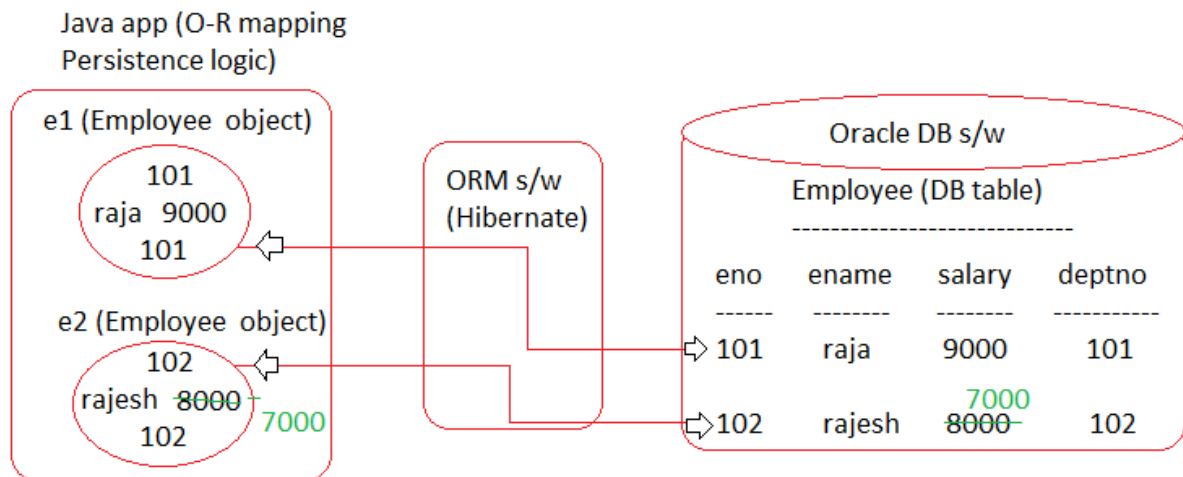
Example:

//Model class or Entity class or BO class

```
public class Employee {  
    private int eno;  
    private String ename;  
    private Float salary;  
    private Integer deptno;  
  
    //setters & getters  
    .....  
}
```

O-R mapping cfgs (XML or Annotations)

com.sahu.model.Employee <-----> EMPLOYEE		
Entity	eno <-----> ENO	DB table columns
clas	ename <-----> ENAME	
props	salary <-----> SALARY	
	deptno <-----> DEPTNO	



Note: ORM s/w internally uses dynamically generated JDBC code + SQL query to complete the persistence operations on the DB s/w.

ORM s/w or Tools or Framework List (Provides abstraction on JDBC):

- Hibernate from SoftTree/ Red hat (1)
- Eclipse Link from Eclipse (2)
- Toplink from Oracle corporation
- iBatis from Apache (3)
- OJB from Apache
- JDO from Apache and etc.

Q. Where should we use JDBC and where should we use O-R mapping (hibernate) for persistence logic development?

Ans.

- If the application is getting huge amount of data batch by batch for processing, then prefer using JDBC rather O-R mapping. If we use O-R mapping here to process huge number of records at a time the application needs to create huge number of objects (like 10,000 objects) at a time in the JVM memory which may lead to application crash or performance issues.
E.g.: Marketing data processing, Census data processing, LIC matured policies detections every month, Sending bank loan EMI remainders, Bank ROI adjustments and etc.
- If the application is getting little amount of (<100 records) for processing at a time then prefer using O-R mapping to take various advantages like DB portability, caching, versioning, timestamping and etc.
E.g.: Nareshit.com enquiries processing, small scale e-commerce apps and etc. organization.

Note:

- ✓ Instead of using plain JDBC directly prefer using Spring JDBC.
- ✓ Instead of using plain ORM (like hibernate) prefer using Spring ORM (old) or Spring Data JPA (new)
- If data the coming for batch processing huge and there in different formats like csv files, XML file, json and etc. then prefer using Spring Batch (another module).
E.g.: Marketing data processing, Census data processing, LIC matured policies detections every month, Bank ROI adjustments and etc.
- If data the coming for batch processing beyond storing and processing capacity of any DB s/w or single computer then prefer using Big Data framework like Hadoop, Spark.
E.g.: Facebook, YouTube, Google, Gmail, FlipKart, Amazon and etc.

Different types DB s/w:

- a. SQL DB s/w
e.g.: Oracle, MySQL, PostgreSQL and etc.
- b. NoSQL DB s/w
e.g.: MongoDB, Cassandra, Couchbase, Neo4j and etc.
- If application is dealing with formatted data having fixed number of attributes, then prefer SQL DB s/w.
E.g.: Employee having max of 30 details, Customer having max of 20 details.
- If one DB table having multiple records with different count of attributes or column values but allocating memory even for unfilled column is meaningless.
E.g.: DB table in oracle for Employee 20 columns
 - 1st record with 4 details
 - 2nd record with 6 details
 - 3rd record with 20 details
 - 4th record with 30 details (X)
- SQL DB s/w contains DB table having fixed structure and schema.
- If application is getting unstructured dynamically growing and not having fixed schema then prefer using NoSQL DB s/w.

Here each record size cannot be increased or decreased dynamically.

Note: NoSQL DB s/w does not maintain DB table and records, in fact they maintain documents, graphs, trees and etc. which do not need any structure any fixed schema.

Records in MongoDB will be stored as documents

- Document 1 with 5 details of customer 1
- Document 2 with 10 details of customer 2
- Document 3 with 100 details of customer 3

Before arrival of Spring data module:

Spring JDBC
Spring App -----> RDBMS DB s/w or SQL DB s/w
(JDBC style)

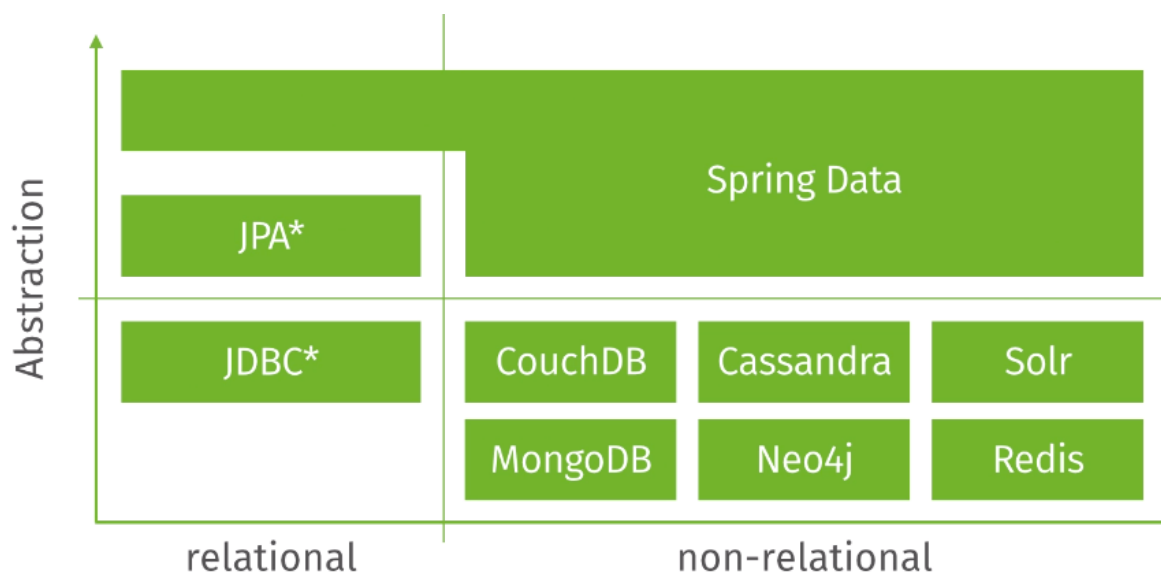
Spring ORM
Spring App -----> RDBMS DB s/w or SQL DB s/w
(O-R mapping style)

MongoDB API + MongoDB Driver
Spring App -----> MongoDB DB s/w (NoSQL DB s/w)

Cassandra API + Cassandra Driver
Spring App -----> Cassandra DB s/w (NoSQL DB s/w)

Note:

- ✓ Spring is not having any module to interact with NoSQL DB s/w before the arrival Spring Data module.
- ✓ Before arrival of Spring Data module there is no single unified mechanism to talk both SQL and NoSQL DB s/w from Spring i.e., we need to use different APIs to interact with different types of SQL or NoSQL DB s/w.



- ✚ Spring Data provides single unified model to interact both SQL and NoSQL DB s/w by providing lots sub modules.
- ✚ Spring Data module provides abstraction on multiple technologies and frameworks to simplify the interaction with both SQL and NoSQL DB s/w in the unified model environment.

Important sub modules of Spring Data module

- Spring Data JPA provides abstraction on ORM framework.
- Spring Data JDBC provides abstraction on JDBC Technology (Different from Spring JDBC)
- Spring data MongoDB provides abstraction on MongoDB API and etc....

Main modules:

- [Spring Data Commons](#) - Core Spring concepts underpinning every Spring Data module.
- [Spring Data JDBC](#) - Spring Data repository support for JDBC.
- [Spring Data JDBC Ext](#) - Support for database specific extensions to standard JDBC including support for Oracle RAC fast connection failover, AQ JMS support and support for using advanced data types.
- [Spring Data JPA](#) - Spring Data repository support for JPA.
- [Spring Data KeyValue](#) - Map based repositories and SPIs to easily build a Spring Data module for key-value stores.
- [Spring Data LDAP](#) - Spring Data repository support for Spring LDAP.
- [Spring Data MongoDB](#) - Spring based, object-document support and repositories for MongoDB.
- [Spring Data Redis](#) - Easy configuration and access to Redis from Spring applications.
- [Spring Data REST](#) - Exports Spring Data repositories as hypermedia-driven RESTful resources.
- [Spring Data for Apache Cassandra](#) - Easy configuration and access to Apache Cassandra or large scale, highly available, data-oriented Spring applications.
- [Spring Data for Apache Geode](#) - Easy configuration and access to Apache Geode for highly consistent, low latency, data-oriented Spring applications.
- [Spring Data for Pivotal GemFire](#) - Easy configuration and access to Pivotal GemFire for your highly consistent, low latency/ high throughput, data-oriented Spring applications.

Community modules:

- [Spring Data Aerospike](#) - Spring Data module for Aerospike.
- [Spring Data ArangoDB](#) - Spring Data module for ArangoDB.
- [Spring Data Couchbase](#) - Spring Data module for Couchbase.
- [Spring Data Azure Cosmos DB](#) - Spring Data module for Microsoft Azure Cosmos DB.
- [Spring Data Cloud Datastore](#) - Spring Data module for Google Datastore.
- [Spring Data Cloud Spanner](#) - Spring Data module for Google Spanner.
- [Spring Data DynamoDB](#) - Spring Data module for DynamoDB.
- [Spring Data Elasticsearch](#) - Spring Data module for Elasticsearch.
- [Spring Data Hazelcast](#) - Provides Spring Data repository support for Hazelcast.
- [Spring Data Jest](#) - Spring Data module for Elasticsearch based on the Jest REST client.
- [Spring Data Neo4j](#) - Spring-based, object-graph support and repositories for Neo4j.
- [Oracle NoSQL Database SDK for Spring Data](#) - Spring Data module for Oracle NoSQL Database and Oracle NoSQL Cloud Service.
- [Spring Data for Apache Solr](#) - Easy configuration and access to Apache Solr for your search-oriented Spring applications.
- [Spring Data Vault](#) - Vault repositories built on top of Spring Data KeyValue.
- [Spring Data YugabyteDB](#) - Spring Data module for YugabyteDB distributed SQL database.

Related modules:

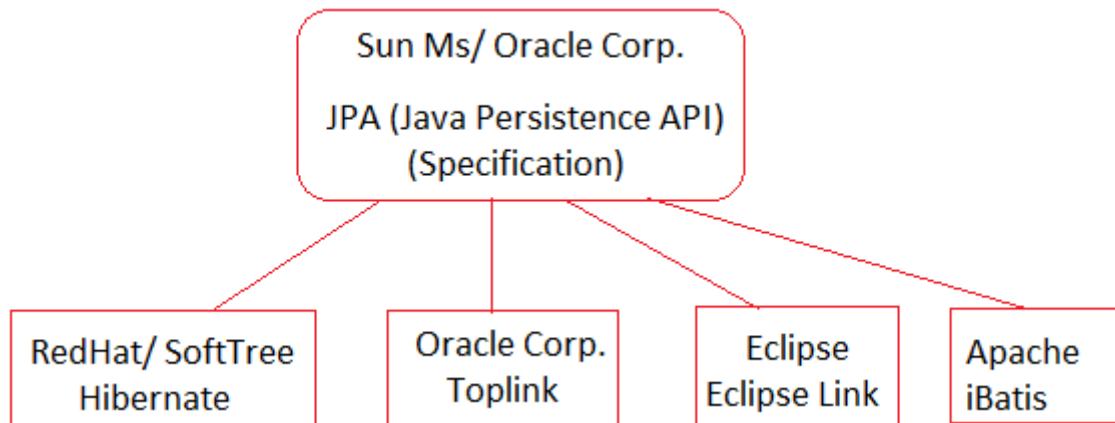
- [Spring Data JDBC Extensions](#) - Provides extensions to the JDBC support provided in the Spring Framework.
- [Spring for Apache Hadoop](#) - Simplifies Apache Hadoop by providing a unified configuration model and easy to use APIs for using HDFS, MapReduce, Pig, and Hive.
- [Spring Content](#) - Associate content with your Spring Data Entities and store it in a number of different stores including the File-system, S3, Database or Mongo's GridFS.

Modules in Incubation:

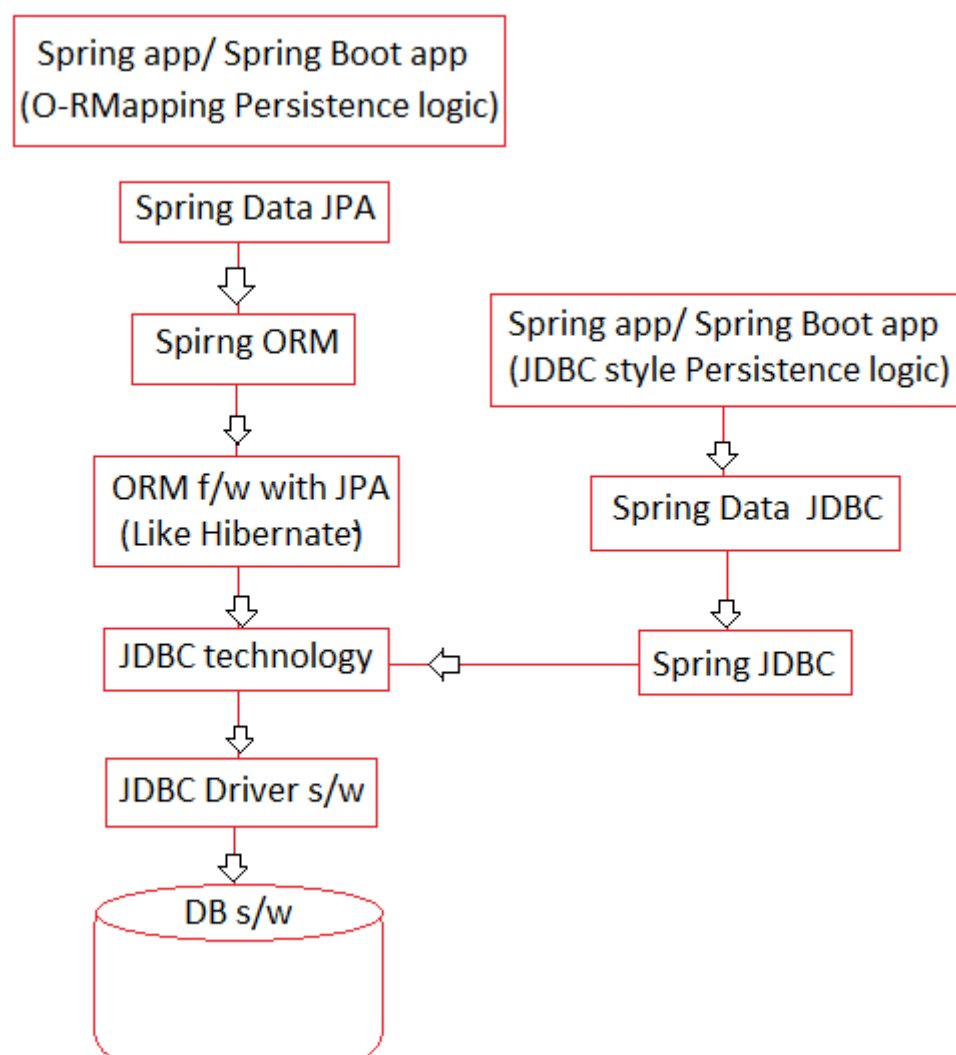
- [Spring Data R2DBC](#) - Spring Data support for R2DBC.

Q. What is JPA?

Ans. It is a software specification providing rules and guidelines to develop ORM s/w like Hibernate, iBatis and etc.



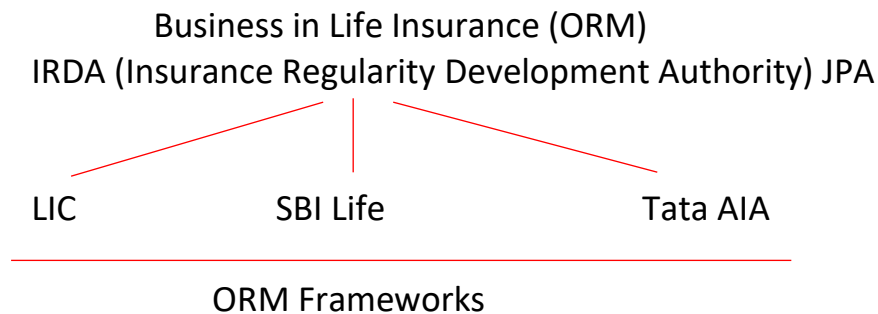
How Spring app or Spring Boot app communicate with DB:



Q. What is difference b/w ORM (O-R mapping) and JPA?

Ans.

- O-R mapping is a style of persistence logic development. For that style of programming JPA provides rules and guidelines to vendor companies to create ORM s/w.



- ORM is concept of Persistence logic development.
- JPA Programming rules and guidelines for ORM persistence logic development
- ORM framework are really s/w or tools that allows to implement ORM persistence logic according JPA rules and guidelines.

Plain JDBC code (JDBC Technology code):

- | | |
|---|-----------------------------|
| ▪ Load JDBC driver class | common logics |
| ▪ Establish the connection | |
| ▪ Create statement | |
| ▪ Send and execute SQL query to DB s/w | Application Specific logics |
| ▪ Gather SQL query results and process them | |
| ▪ Handle exceptions | common logics |
| ▪ Perform TxMgmt | |
| ▪ Close JDBC objects | |

Note:

- ✓ Developer should write both common logics and application specific logics (Boilerplate code problem).
- ✓ The code that repeats across the multiple parts of projects either with no changes or with minor changes is called boiler plate code.

Plain Hibernate code (ORM f/w code):

- Create Configuration class object

- | | |
|---|-----------------------------|
| <ul style="list-style-type: none"> ▪ Create Session factory object ▪ Creates Session object | common logics |
| <ul style="list-style-type: none"> ▪ Write persistence logic using Session object and Entity class objects | Application Specific logics |
| <ul style="list-style-type: none"> ▪ Perform Tx Mgmt ▪ Close Session, SessionFactory objects | common logics |

Note: Same problem boiler plate code, like above.

Spring ORM code (providing abstraction on plain ORM f/w):

- Create HibernateTemplate class obj (takes care of common logics of ORM style persistence logic)
- Develop Persistence logic using the objects of Entity class.

Note: Boilerplate code problem is solved.

Limitation with spring ORM module:

- If project is having 500 DB tables and we are looking perform CRUD operations on all DB tables then we should develop 500 DAO Interfaces, 500 DAO implementation classes and 500* 6 methods having common persistence logic activities + additional methods in each DAO implementation class specific to project requirement. (This is another type of boiler plate code).

Spring Data JPA Code:

- ✓ Just create Repository/ DAO interface extending pre-defined Repository Interface (Different types of Repository interfaces are there).
- ✓ Pre-define repository interface contains declaration of CRUD operation methods (10 to 20).
- ✓ If needed, declare some custom methods by following coding conventions

Advantage with spring data JPA

- ✓ Implementation classes for Custom repository interfaces will be generated dynamically providing persistence logics for common methods inherited from pre-defined Repository Interfaces, and also for custom methods declarations. (All these operations are taken care by Spring Data JPA using InMemory Proxy classes).

Note: Now for 500 DB tables, we just need to take 500 custom Repository interfaces having optional custom method declaration.

Normal Java class:

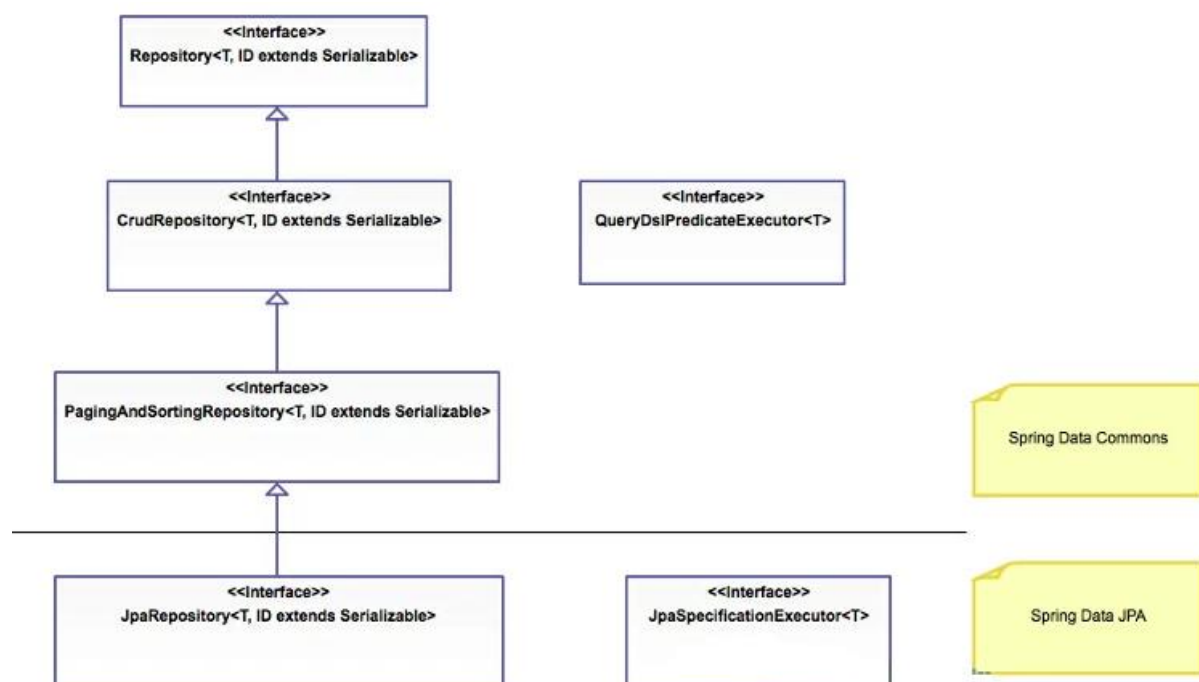
.java (HDD) ----> .class (HDD) ----> JVM Loads .class file (JVM Memory of RAM)
----> execution class file (JVM Memory of RAM)

InMemory Proxy class (Everything happens at JVM memory of RAM):

Run the application ----> source code generation (JVM Memory of RAM) ---->
compilation (JVM Memory of RAM) ----> JVM Loads .class file (JVM
Memory RAM) ----> execution class file (JVM Memory of RAM)

Note: Spring Data JPA uses Proxy Design pattern to generate implementation classes of programmer supplied DAO/ Repository interfaces as InMemory proxy classes dynamically at runtime. i.e., While working with Spring Data JPA the persistence layer just contains DAO/ Repository interfaces having few custom methods declarations.

Repositories in Spring Data JPA



- Repository (I) is a marker interface means no methods.
- CrudRepository having 12 methods.
- PagingAndSortingRepository having 2 methods.
- JpaRepository is same a CrudRepository but contains JPA specific

methods, total 15 methods.

- ✚ Spring Data JPA internally uses hibernate as ORM framework.
- ✚ So strong knowledge in Hibernate programming definitely helps to work with Spring Data JPA effectively.
- ✚ While developing Entity classes/ model classes we need to prefer using annotations in the following order
 - a. JPA annotations (Portable across the multiple ORM s/w)
 - b. Java Config annotations (Given by JDK/ JEE)
 - c. Hibernate Specific Annotations
 - d. Third party Annotations

Procedure to develop our First Spring Data JPA application as standalone application (Partially layered application):

Client app -----> Services class -----> Repository -----> DB s/w
(Presentation logic) (Business logic) (InMemory Implementation class of Repository (I) contains persistence logic)

Step 1: Create Spring Boot starter project adding the following starters as dependencies,

- Lombok
- Spring Data JPA
- Oracle Driver

Note: Spring Data JPA starter gives Hibernate jar files + HikariCP jar files + Spring Data JPA jar files

Step 2: Add the DataSource, Hibernate specific properties in application.properties file.

Note:

- ✓ Dialect is Hibernate internally managed component/ service that is capable of generating SQL queries based on the underlying DB s/w and its version. All dialects are the classes extending from [org.hibernate.dialect.Dialect](#) (c).
- ✓ The dialect value will change based on DB s/w and its version we choose. It is optional property to specify because based on JDBC/ DataSource properties we configure the Dialect component will be picked up automatically.

- ✓ To get all dialects: [\[Click here\]](#)
- ✓ To get all application properties: [\[Click here\]](#)

✚ `spring.jpa.show-sql=true` : To shows the dialect component generated SQL queries as log messages.

✚ `spring.jpa.hibernate.ddl-auto=update` : To use Dynamic schema generation feature of Hibernate.

- `create`: Always creates new DB tables by dropping the DB tables if already existed
- `update`: If DB tables are already available then uses them, if modifications required alters them (only adding of new columns is possible). If DB tables are not there, then creates new DB tables according to the entity classes. (So useful project development and production environment).
- `validate`: Does nothing on its own expect verifying whether DB tables are there according entity classes or not i.e., DB tables must be created by developers/ DB team manually.
- `create-drop`: Creates DB tables at the startup of app uses them throughout app's execution and drops them at the end of the application. So useful in "test", "UAT" environments also in demos of projects and POCs
- `None`: Does nothing, DB tables should create manually, even verification does not take place.

Step 3: Develop the Entity class using the above said annotations priority.

Note:

- ✓ `@Table`, `@Entity`, `@Column`, `@Id`, `@GeneratedValue` and etc. are the JPA annotations. In that `@Entity`, `@Id` are mandatory annotations and remaining all are optional annotations.
- ✓ `@Id` property configuration is mandatory pointing to PK column.
- ✓ We can control length of only string type columns not numeric type columns because based on java data types range the underlying DB s/w decides the lengths of numeric columns.

Step 4: Develop user-defined Repository interface extends from predefined `CrudRepository` (I).

Note: We generally take Entity classes, Repository interfaces on 1 per DB table basis.

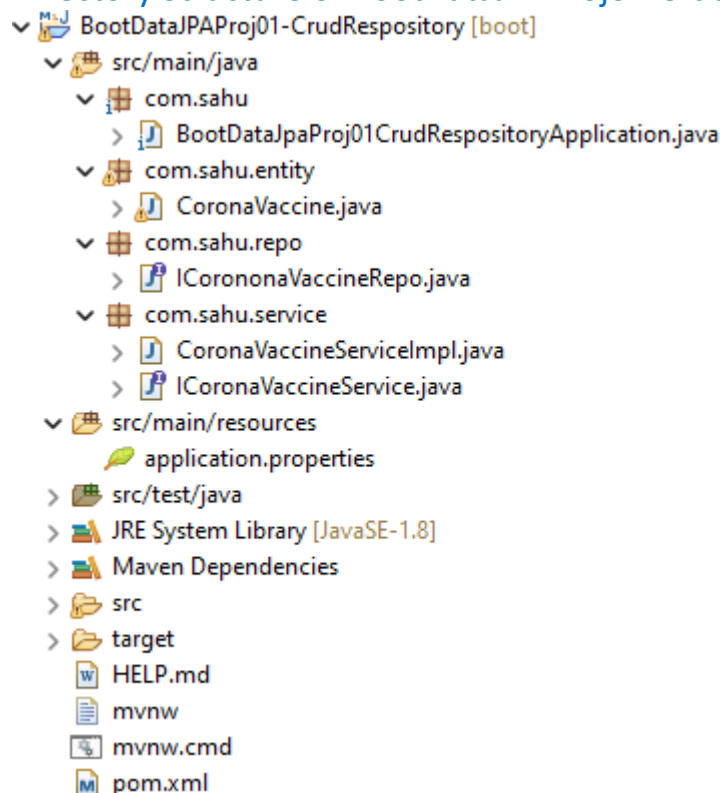
Step 5: Develop service (I) and service implementation class.

Step 6: Develop code in client App (main class) or Runner class for testing

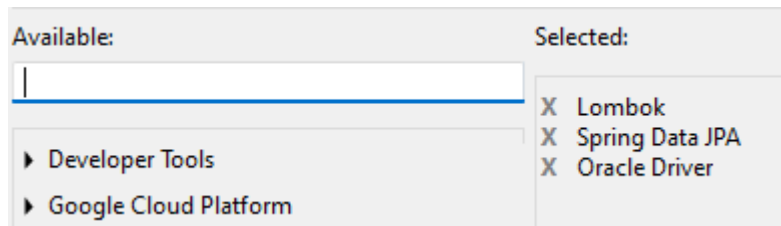
Note:

- ✓ Spring JDBC internally uses plain JDBC and translates SQLException to DataAccessException class hierarchy exceptions.
- ✓ Spring ORM internally uses plain ORM s/w like hibernate and translates ORM s/w specific exception like HibernateException to DataAccessException class hierarchy exceptions
- ✓ Spring Data JPA internally uses plain ORM s/w like hibernate and translates ORM s/w specific exception like HibernateException to DataAccessException class hierarchy exceptions
- ✓ Spring Mongo DB internally uses Mongo DB API and translates MongoDB API specific exceptions to DataAccessException class hierarchy exceptions
- ✓ All these are internally using exception rethrowing concept.

Directory Structure of BootDataJPAProj01-CrudRepository:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Following Staters are needed so choose them during project creation.



- Then place the following code with in their respective files.

application.properties

```
#DataSource Configuration
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
#Other possible values create, validate, create-drop
```

CoronaVaccine.java

```
package com.sahu.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
//@Table(name="CORONA_VACCINE_TAB") optional if you want to DB table
name same as entity class name
--
```

```

@Data
@AllArgsConstructor
@NoArgsConstructor
public class CoronaVaccine implements Serializable {
    @Id //To make property as singular ID property and to map with
singular PK column
    //@Column(name="reg_No") optional if you want to take property
name as same as the column name
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long regNo;
    @Column(length=20)
    private String name;
    @Column(length=20)
    private String company;
    @Column(length=20)
    private String country;
    private Double price;
    private Integer requiredDoseCount;
}

```

ICorononaVaccineRepo.java

```

package com.sahu.repo;

import org.springframework.data.repository.CrudRepository;

import com.sahu.entity.CoronaVaccine;

public interface ICorononaVaccineRepo extends
CrudRepository<CoronaVaccine, Long> {

}

```

ICoronaVaccineService.java

```

package com.sahu.service;

import com.sahu.entity.CoronaVaccine;

public interface ICoronaVaccineService {
    public String registerVaccine(CoronaVaccine coronaVaccine);
}

```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;
import com.sahu.repo.ICoronaVaccineRepo;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICoronaVaccineRepo corononaVaccineRepo;

    @Override
    public String registerVaccine(CoronaVaccine coronaVaccine) {
        CoronaVaccine vaccine = null;
        if (coronaVaccine!=null)
            vaccine = corononaVaccineRepo.save(coronaVaccine);
        return vaccine!=null?"Vaccine Registered Successfully with  
"+vaccine.getRegNo():"Vaccine registration Failed";
    }
}
```

BootDataJpaProj01CrudRespositoryApplication.java

```
package com.sahu;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.dao.DataAccessException;

import com.sahu.entity.CoronaVaccine;
import com.sahu.service.ICoronaVaccineService;

@SpringBootApplication
```

```

public class BootDataJpaProj01CrudRespositoryApplication {

    public static void main(String[] args) {
        //Get Access to IoC container
        ApplicationContext ctx =
SpringApplication.run(BootDataJpaProj01CrudRespositoryApplication.class,
args);

        //Get service class object
        ICoronaVaccineService vaccineService =
ctx.getBean("vaccineService", ICoronaVaccineService.class);
        //Invoke methods
        try {
            CoronaVaccine vaccine = new
CoronaVaccine(null,"COVAXIN", "Bharat-Bio-Tech","India", 780.0, 2);

            System.out.println(vaccineService.registerVaccine(vaccine));
        }
        catch (DataAccessException dae) {
            dae.printStackTrace();
        }
        catch (Exception e) {
            e.printStackTrace();
        }

        //closing container
        ((ConfigurableApplicationContext) ctx).close();
    }
}

```

Note:

- ✓ repo.save(-) internally uses persist(-) method to perform save object operation if record is not already available in the DB table with the generated/ given id value.
- ✓ repo.save(-) internally uses merge(-) method to perform update object operation if record is already available in the DB table with the generated/ given id value.
- ✓ While performing non-select operations we need not worry about Tx Mgmt. @Service annotation-based service class and Repository methods will take care of that object internally

```

@Transactional
@Override
public <S extends T> S save(S entity) {
    Assert.notNull(entity, "Entity must not be null.");

    if (entityInformation.isNew(entity)) {
        em.persist(entity);
        return entity;
    } else {
        return em.merge(entity);
    }
}

```

Q. By enabling generator can we perform update operation with repo.save(-) method?

Ans.

1. Yes, in this scenario - If DB table is already having full of records and the generator generated id value is also already there in PK column.
2. No, in this scenario - If DB table is empty or if the generator generated value is not already there in the PK column. In this situation take the support of custom methods declaration in repository interface for update operation.

Note: AUTO generator internally uses hibernate_sequence (start with 1 and increment by 1) in case of oracle DB s/w and the same generator uses "autoincrement" (identity column) in case of MySQL DB s/w.

Q. What is the difference between JPA persist (-) and hibernate persist (-) method?

Ans. JPA persist (-) supports generators to generate id value, whereas hibernate persist (-) method does not support generators configuration.

Q. What is difference between JPA's persist (-), merge (-) and Spring Data JPA's save (-) method?

Ans.

- JPA's persist (-) method performs save object operation (inserting record).
- JPA's merge (-) method performs update object operation (updating record).
- Spring Data JPA's save (-) method performs either save object or update

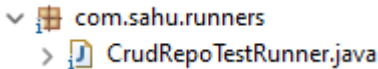
object operation because it internally uses JPA's persist (-) or merge (-) method based availability of the record.

Q. What is the difference between Spring Data JPA's save (-) and hibernate save (-) method?

Ans. Hibernate save (-) performs only save object operation whereas Spring Data JPA's save (-) performs either save object or update object operation.

Note: All Spring Data JPA's non-select single row operation methods takes care of Tx Mgmt automatically.

Developing Client code using the support of Spring Boot Runners:

- Create the following package and class in project directory structure.

- And place the following code with their respective files.

[BootDataJpaProj01CrudRespositoryApplication.java](#)

```
package com.sahu;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BootDataJpaProj01CrudRespositoryApplication {

    public static void main(String[] args) {

        SpringApplication.run(BootDataJpaProj01CrudRespositoryApplication.
class, args);
    }

}
```

[CrudRepoTestRunner.java](#)

```
package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
```

```

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Component;

import com.sahu.entity.CoronaVaccine;
import com.sahu.service.ICoronaVaccineService;

@Component
public class CrudRepoTestRunner implements CommandLineRunner {

    @Autowired
    private ICoronaVaccineService coronaVaccineService;

    @Override
    public void run(String... args) throws Exception {
        // Invoke methods
        try {
            CoronaVaccine vaccine = new CoronaVaccine(null,
"COVAXIN", "Bharat-Bio-Tech", "India", 785.0, 2);

            System.out.println(coronaVaccineService.registerVaccine(vaccine));
        } catch (DataAccessException dae) {
            dae.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

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

- This method is given for batch insertion, i.e., inserts the multiple records by sending batch insert SQL queries to DB s/w from Spring Data JPA application at once and reduces the network round trips between application and DB s/w.
- Use case: Group ticket reservation
- Iterable (I) is the functional interface of java collections API, and it common interface implemented by all the collections directly or indirectly.
- In ORM (O-R mapping) saving object means inserting record.

- In ORM (O-R mapping) updating object means updating record.
- In ORM (O-R mapping) deleting object means deleting record.
- In ORM (O-R mapping) Load object means retrieving record .

ICoronaVaccineService.java

```
public interface ICoronaVaccineService {
    public String registerVaccine(CoronaVaccine coronaVaccine);
    public Iterable<CoronaVaccine>
registerBatch(Iterable<CoronaVaccine> vaccines);
}
```

CoronaVaccineServiceImpl.java

```
@Override
public Iterable<CoronaVaccine>
registerBatch(Iterable<CoronaVaccine> vaccines) {
    if (vaccines!=null)
        return corononaVaccineRepo.saveAll(vaccines);
    else
        throw new IllegalArgumentException("Batch insertion is
not done.");
}
```

CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        //Bulk insertion/ batch insertion
        Iterable<CoronaVaccine> listCoronaVaccines =

        coronaVaccineService.registerBatch(Arrays.asList(new
CoronaVaccine(null, "Sputnlk", "Russie", "Russia", 568.5, 2),
new CoronaVaccine(null,
"Pyzer", "Pyzer", "USA", 568.5, 2),
new CoronaVaccine(null,
"Moderena", "Moderena", "USA", 568.5, 2)));

        System.out.println("The regNos. are - ");
        listCoronaVaccines.forEach(vaccine->
```



```

        System.out.println(vaccine.getRegNo());
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Note: List.of(...), Map.of(...), Set.of(...) are introduced from Java9 to create and return immutable collection objects having given data as the elements.

E.g., Iterable<CoronaVaccine> listCoronaVaccines =

```

coronaVaccineService.registerBatch(List.of(
    new CoronaVaccine(null, "Sputnik", "Russie", "Russia", 568.5, 2),
    new CoronaVaccine(null, "Pyzer", "Pyzer", "USA", 568.5, 2),
    new CoronaVaccine(null, "Modrna", "Modrna", "USA", 568.5, 2)));

```

We can also use Arrays.asList(...) method to get mutable List collection objects having given data as the elements.

E.g., Iterable<CoronaVaccine> listCoronaVaccines =

```

coronaVaccineService.registerBatch(Arrays.asList(
    new CoronaVaccine(null, "Sputnik", "Russie", "Russia", 568.5, 2),
    new CoronaVaccine(null, "Pyzer", "Pyzer", "USA", 568.5, 2),
    new CoronaVaccine(null, "Modrna", "Modrna", "USA", 568.5, 2)));

```

Select operations using CrudRepository

✚ These are the following select operations.

- Optional<T> findById (ID id); - To get single record
- Iterable<T> findAll (); - To get all records
- Iterable<T> findAllById (Iterable<ID> ids), - To get multiple records based on given id
- boolean existsById (ID id), - To check record is available or not
- long count (); - To get count of records.

ICoronaVaccineService.java

```

public Long getVaccineCount();
public Boolean checVaccineAvailabilityByRegNo(Long regNo);
public Iterable<CoronaVaccine> fetchAllDetails();
public Iterable<CoronaVaccine> fetchAllDetailsByIds(Iterable<Long>
regNos);

```

CoronaVaccineServiceImpl.java

```
@Override
public Long getVaccineCount() {
    return coronaVaccineRepo.count();
}
@Override
public Boolean checVaccineAvailabilityByRegNo(Long regNo) {
    return coronaVaccineRepo.existsById(regNo);
}
@Override
public Iterable<CoronaVaccine> fetchAllDetails() {
    return coronaVaccineRepo.findAll();
}
@Override
public Iterable<CoronaVaccine> fetchAllDetailsByIds(Iterable<Long>
regNos) {
    return coronaVaccineRepo.findAllById(regNos);
}
```

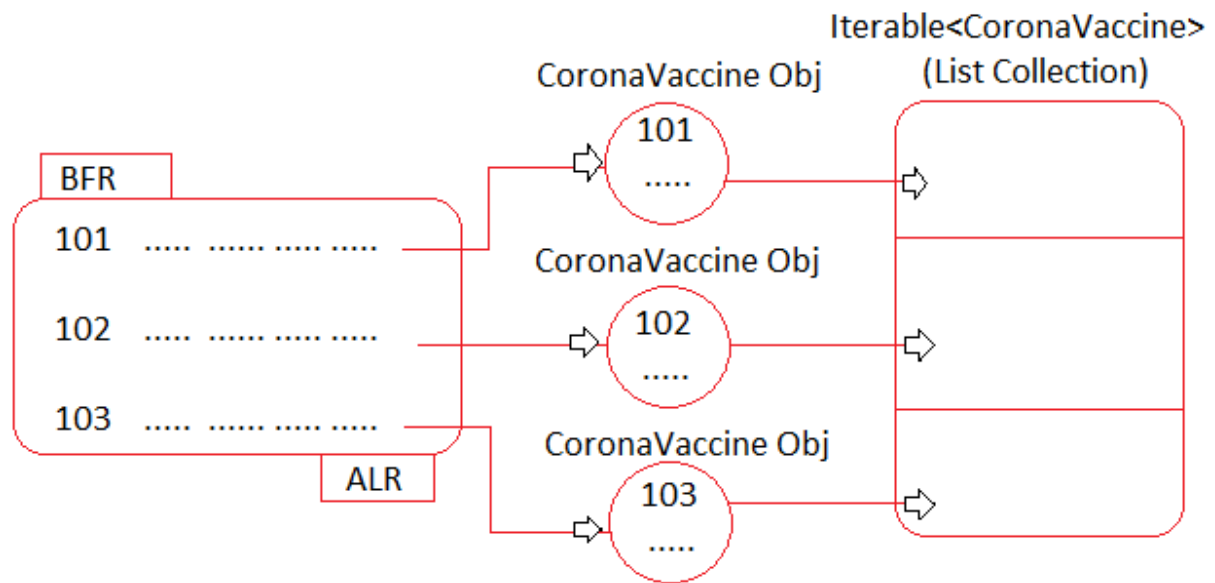
CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {

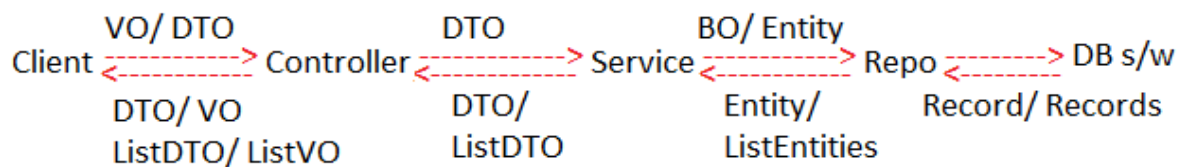
        System.out.println("Records Count -
"+coronaVaccineService.getVaccineCount());

        System.out.println("8 Reg No vaccine available ? -
"+coronaVaccineService.checVaccineAvailabilityByRegNo(8l));

        coronaVaccineService.fetchAllDetails().forEach(System.out::println);
        coronaVaccineService.fetchAllDetailsByIds(Arrays.asList(1l, 2l, 3l, 4l,
5l)).forEach(System.out::println);
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

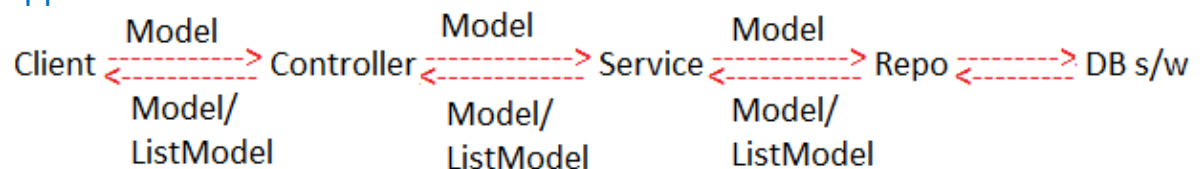


Approach 1:



This approach is fading out because we are using multiple number of java bean classes and objects.

Approach 2:



This approach is trending because using single java bean and its objects we can carry data across the layers.

BO class = Entity class = Model class = Domain class = Persistence class

Note:

- ✓ Model class in approach2 can have extra properties to hold business logic generated data (i.e., not just the properties representing DB table columns).
- ✓ To avoid problems related to Dynamic Schema generation with these extra properties we can place @Transient annotation on the extra properties.
- ✓ @Transient: Makes the underlying ORM framework not using given

property of Entity/ Model class not participating in Persistence activity

E.g.,

@Entity

class Student {

private Integer sno;

private String sname;

private String sadd;

private Float total;

private Float avg;

@Transient

private String remarks;

Does not participate in

Persistence Operation

}

Db table

Student

sno (int) (pk)

sname (vc2)

sadd (vc2)

total (float)

avg (float)

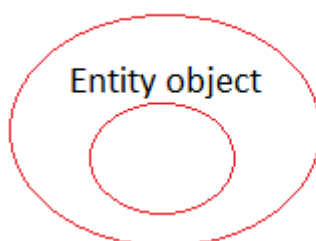
Q. Why we are not using @Repository in Spring Data module?

Ans. @Repository is given to configure java class as Spring bean cum DAO class. In Spring Data module, we do not develop separate DAO class we get InMemory proxy classes implementing given Repository interface having persistence logic. So, there is no need of using @Repository.

Optional<T> findById (ID id):

- Gives Single object of Entity class based on the given id value.
- Returns Optional object (Java 8) having entity class object. So, we can check Entity object is present or not (isPresent ()) inside Optional object before collecting and using it.
- Optional API is very useful to avoid NullPointerException from the application.

Optional object (opt)



if(opt.isPresent())

CoronaVaccine vaccine = opt.get();

ICoronaVaccineService.java

```
public Optional<CoronaVaccine> fetchVaccineById(Long regNo);
```

CoronaVaccineServiceImpl.java

```
@Override
public Optional<CoronaVaccine> fetchVaccineById(Long regNo) {
    return coronaVaccineRepo.findById(regNo);
}
```

CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        Optional<CoronaVaccine> optVaccine =
        coronaVaccineService.fetchVaccineById(111);
        if (optVaccine.isPresent())
            System.out.println(optVaccine.get());
        optVaccine.orElseThrow(() -> new
        IllegalArgumentException("Record not found"));

    } catch (DataAccessException dae) {
        dae.printStackTrace();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Delete operations using CrudRepository

✚ These are the following select operations.

- void deleteById (ID id);
- void delete (T entity);
- void deleteAllById (Iterable<? extends ID> ids);
- void deleteAll (Iterable<? extends T> entities);
- void deleteAll ();

void deleteById (ID id);

- Deleting record by the given id value.
- It is Spring Data JPA supplied method.

ICoronaVaccineService.java

```
public String removeVaccineByRegNo(Long regNo);
```

CoronaVaccineServiceImpl.java

```
@Override
public String removeVaccineByRegNo(Long regNo) {
    Optional<CoronaVaccine> optVaccine =
corononaVaccineRepo.findById(regNo);
    if (optVaccine.isPresent()) {
        corononaVaccineRepo.deleteById(regNo);
        return "Record delete having Reg. No : "+regNo;
    }
    return "Record not found for deletion";
}
```

CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        System.out.println(coronaVaccineService.removeVaccineByRegNo(81)
    );
        } catch (DataAccessException dae) {
            dae.printStackTrace();
        }
    }
}
```

void delete (T entity);

- Deletes the record by taking given entity object id value.
- It is JPA specific method implemented in Spring Data JPA.

ICoronaVaccineService.java

```
public String removeVaccineByObject(CoronaVaccine vaccine);
```

CoronaVaccineServiceImpl.java

```
@Override
public String removeVaccineByObject(CoronaVaccine vaccine) {
    Optional<CoronaVaccine> optVaccine =
corononaVaccineRepo.findById(vaccine.getRegNo());
    if (optVaccine.isPresent()) {
        corononaVaccineRepo.delete(vaccine);
        return "Record delete having Reg. No :
```

```

        "+vaccine.getRegNo();
    }
    return "Record not found for deletion";
}

```

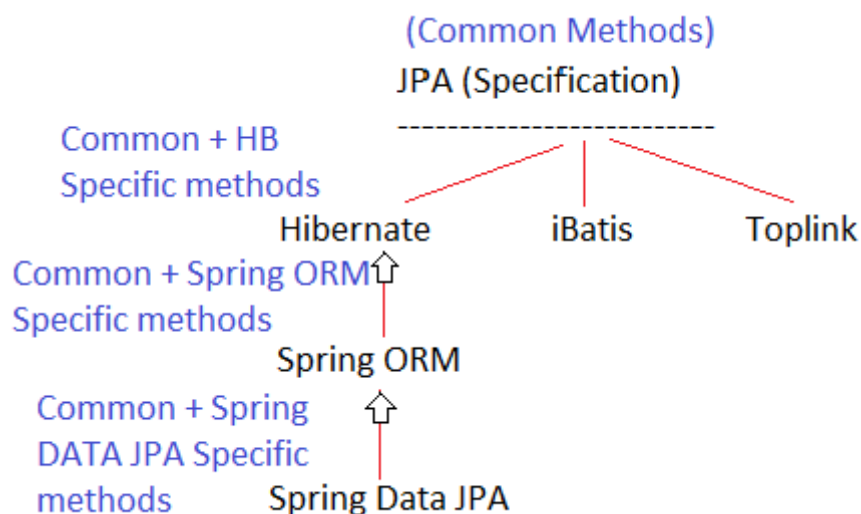
CrudRepoTestRunner.java

```

@Override
public void run(String... args) throws Exception {
    try {
        CoronaVaccine vaccine = new CoronaVaccine();
        vaccine.setRegNo(121);

        System.out.println(coronaVaccineService.removeVaccineByObject(vaccine));
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}

```



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

- Performs bulk deletion based on the given multiple id values.
- It is direct method of Spring Data JPA
- Deletes multiple records based on given ids if all ids-based records are present in DB table otherwise throws exception.
- Does not generate single delete query having condition using where and in clauses.
- It actually generates multiple delete queries taking id value as the

criteria value to delete multiple records by executing delete queries in a batch repo. deleteAllById(-) method.

ICoronaVaccineService.java

```
public String removeVaccineByIds(Iterable<Long> ids);
```

CoronaVaccineServiceImpl.java

```
@Override
public String removeVaccineByIds(Iterable<Long> ids) {
    Iterable<CoronaVaccine> listEntities =
corononaVaccineRepo.findAllById(ids);
    if (((List)listEntities).size() == ((List)ids).size()) {
        corononaVaccineRepo.deleteAllById(ids);
        return ((List)listEntities).size()+" no. of records are
deleted";
    }
    return "Problem in deleting records";
}
```

CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        System.out.println(coronaVaccineService.removeVaccineByIds(Arrays
.asList(21l, 31l, 41l, 6l)));
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
```

void deleteAll ();

- Deletes all the records of the DB table
- First gets all the records from the DB table by executing select query and takes their id values as criteria values to execute multiple delete queries in a batch i.e., it is not just executing single delete query without condition.
- It is Spring Data JPA direct method.

ICoronaVaccineService.java

```
public String removeAllVaccines();
```

CoronaVaccineServiceImpl.java

```
@Override
public String removeAllVaccines() {
    Long count = coronaVaccineRepo.count();
    if (count !=0) {
        coronaVaccineRepo.deleteAll();
        return count+" no. of records are deleted";
    }
    return "Table is empty to delete records";
}
```

CrudRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        System.out.println(coronaVaccineService.removeAllVaccines());
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
```

PagingAndSortingRepository

- ✚ It is given to perform sorting of records and pagination.
- ✚ Pagination is all about displaying huge amount of records page by page.
- ✚ We can sort the records either in Ascending order or in Descending order
- ✚ In Ascending order
 - Special characters (*, ?, _, -, .);
 - Numbers (0-9)
 - Uppercase alphabets (A-Z)
 - Lowercase alphabets (a-z)

@NoRepositoryBean

```
public interface PagingAndSortingRepository<T, ID> extends
```

```

CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort); //For sorting
    Page<T> findAll(Pageable pageable); //Only for Pagination
}

```

`Iterable<T> findAll(Sort sort);`

- Performs the required ascending or descending order of sorting based given single or multiple properties. We need to specify all these inputs in the given sort object.

Note:

- ✓ [String...] - var args param internally array type param.
- ✓ One method can have only one var args param.

Directory Structure of BootDataJPAProj02-PagingAndSortingRepository:

```

v BootDataJPAProj02-PagingAndSortingRepository [boot]
v src/main/java
  v com.sahu
    > BootDataJpaProj02PagingAndSortingRepositoryApplication.java
  v com.sahu.entity
    > CoronaVaccine.java
  v com.sahu.repo
    > ICoronaVaccineRepo.java
  v com.sahu.runners
    > PagingAndSortingRepoTestRunner.java
  v com.sahu.service
    > CoronaVaccineServiceImpl.java
    > ICoronaVaccineService.java
v src/main/resources
  application.properties
> src/test/java
> JRE System Library [JavaSE-1.8]
> Maven Dependencies
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml

```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.
- Place the following code within their respective files.

ICoronaVaccineService.java

```
package com.sahu.service;

import com.sahu.entity.CoronaVaccine;

public interface ICoronaVaccineService {
    public Iterable<CoronaVaccine> fetchDetails(boolean asc, String...
properties);
}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;
import com.sahu.repo.ICorononaVaccineRepo;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICorononaVaccineRepo corononaVaccineRepo;

    @Override
    public Iterable<CoronaVaccine> fetchDetails(boolean asc, String...
properties) {
        Sort sort = Sort.by(asc?Direction.ASC:Direction.DESC,
properties);
        Iterable<CoronaVaccine> listEntities =
corononaVaccineRepo.findAll(sort);
        return listEntities;
    }
}
```

ICorononaVaccineRepo.java

```
package com.sahu.repo;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.entity.CoronaVaccine;

public interface ICorononaVaccineRepo extends
PagingAndSortingRepository<CoronaVaccine, Long> { }
```

PagingAndSortingRepoTestRunner.java

```
package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Component;

import com.sahu.entity.CoronaVaccine;
import com.sahu.service.ICorononaVaccineService;

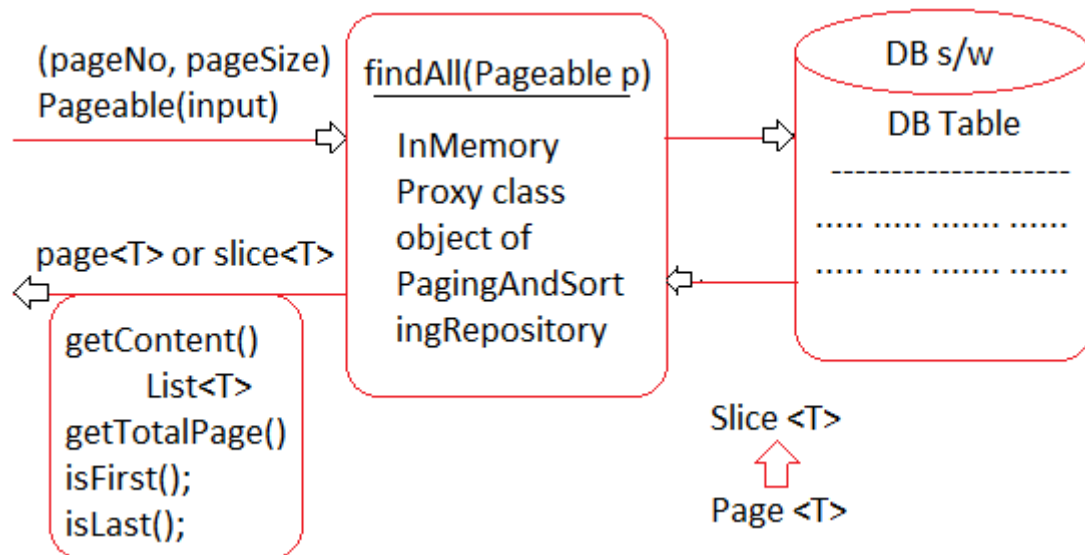
@Component
public class PagingAndSortingRepoTestRunner implements
CommandLineRunner {

    @Autowired
    private ICorononaVaccineService coronaVaccineService;

    @Override
    public void run(String... args) throws Exception {
        try {
            Iterable<CoronaVaccine> listEntities =
coronaVaccineService.fetchDetails(true, "price", "name");
            listEntities.forEach(System.out::println);
        } catch (DataAccessException dae) {
            dae.printStackTrace();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Page<T> findAll(Pageable pageable);

- This method takes pageNo (0 based), pageSize as inputs in the form of Pageable object and returns output as Page<T>/ Slice<T> object having requested page records, pages count, current numbers, total records and etc.



- If total records count is 20 and pagesize is 5 then it divides 20 records into 4 pages (5, 5, 5, 5). If we ask for 3rd page (indirectly 4th page) records then it gives 16 to 20 records. If ask for 2nd page (indirectly 3rd page) records then it gives 11 to 15 records.
- Pageable pageable = PageRequest.of(2,5);

Note: Pageable object can have both Paging and Sorting information.

ICoronaVaccineService.java

```

public Iterable<CoronaVaccine> fetchDetailsByPageNo(int pageNo,
int pageSize, boolean asc, String... properties);
  
```

CoronaVaccineServiceImpl.java

```

@Override
public Iterable<CoronaVaccine> fetchDetailsByPageNo(int pageNo,
int pageSize, boolean asc, String... properties) {
    //create pageable object having inputs
    Pageable pageable = PageRequest.of(pageNo, pageSize,
asc?Direction.ASC:Direction.DESC, properties);
  
```

```
Page<CoronaVaccine> page =  
corononaVaccineRepo.findAll(pageable);  
return page.getContent();  
}
```

PagingAndSortingRepoTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
    try {  
        coronaVaccineService.fetchDetailsByPageNo(2, 4, true,  
"price").forEach(System.out::println);  
    } catch (DataAccessException dae) {  
        dae.printStackTrace();  
    }  
}
```

Note:

- ✓ First it performs sorting and then performs pagination.
- ✓ If requested page no, page is not available then it returns Page<T> having empty list collection for getContent() method.

Displaying Huge Number of Page by page or part by part in Console Environment:

ICoronaVaccineService.java

```
public void fetchDetailsByPagination(int pageSize);
```

CoronaVaccineServiceImpl.java

```
@Override  
public void fetchDetailsByPagination(int pageSize) {  
    // get total records count  
    long count = corononaVaccineRepo.count();  
    //Decides the pages count  
    long pageCount = count/pageSize;  
    pageCount =  
(count%pageSize==0)?pageCount:++pageCount;  
  
    for (int i = 0; i < pageCount; i++) {  
        Pageable pageable = PageRequest.of(i, pageSize);
```

```

        Page<CoronaVaccine> page =
corononaVaccineRepo.findAll(pageable);
        page.getContent().forEach(System.out::println);
        System.out.println("----- "+(i+1)+" of
"+page.getTotalPages()+" -----");
    }
}

```

PagingAndSortingRepoTestRunner.java

```

@Override
public void run(String... args) throws Exception {
    try {
        coronaVaccineService.fetchDetailsByPagination(2);
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}

```

JpaRepository

- JpaRepository contains all methods functioning in Sun MS JPA style. Use only additional methods of this Repository to avoid using the similar methods that are already in other repositories.
- Most of the methods in JpaRepository are having similar functionality of CrudRepository, PagingAndSortingRepository methods having different signatures avoid them and use only different functionality methods.

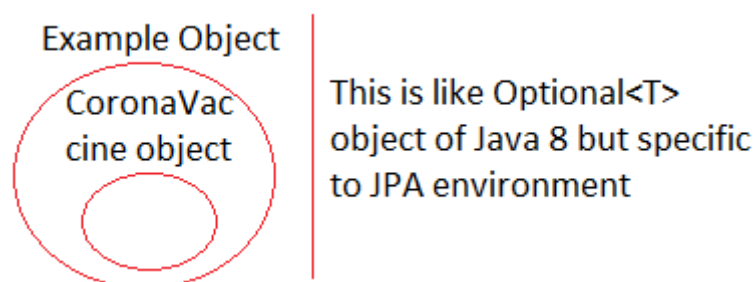
Modifier and Type	Method and Description
void	<code>deleteAllByIdInBatch(Iterable<ID> ids)</code> Deletes the entities identified by the given ids using a single query.
void	<code>deleteAllInBatch()</code> Deletes all entities in a batch call.
void	<code>deleteAllInBatch(Iterable<T> entities)</code> Deletes the given entities in a batch which means it will create a single query.
default void	<code>deleteInBatch(Iterable<T> entities)</code> Deprecated. Use <code>deleteAllInBatch(Iterable)</code> instead.
List<T>	<code>findAll()</code>
<S extends T> List<S>	<code>findAll(Example<S> example)</code>

<code><S extends T> List<S></code>	<code>findAll(Example<S> example, Sort sort)</code>
<code>List<T></code>	<code>findAll(Sort sort)</code>
<code>List<T></code>	<code>findAllById(Iterable<ID> ids)</code>
<code>void</code>	<code>flush()</code> Flushes all pending changes to the database.
<code>T</code>	<code>findById(ID id)</code> Returns a reference to the entity with the given identifier.
<code>T</code>	<code>getOne(ID id)</code> Deprecated. use <code>JpaRepository#findById(ID)</code> instead.
<code><S extends T> List<S></code>	<code>saveAll(Iterable<S> entities)</code>
<code><S extends T> List<S></code>	<code>saveAllAndFlush(Iterable<S> entities)</code> Saves all entities and flushes changes instantly.
<code><S extends T> S</code>	<code>saveAndFlush(S entity)</code> Saves an entity and flushes changes instantly.

Note: `findAll (-)` methods in `CrudRepository` and `PagingAndSortingRepository` returns `Iterable<T>`, whereas the `findAll (-)` methods in `JpaRepository` returns `List<T>` directly. Same thing in case of `saveAll (-)` methods.

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

- Example object is a container object holding given Entity object.
Example `ex = Example.of(new CoronaVaccine());`



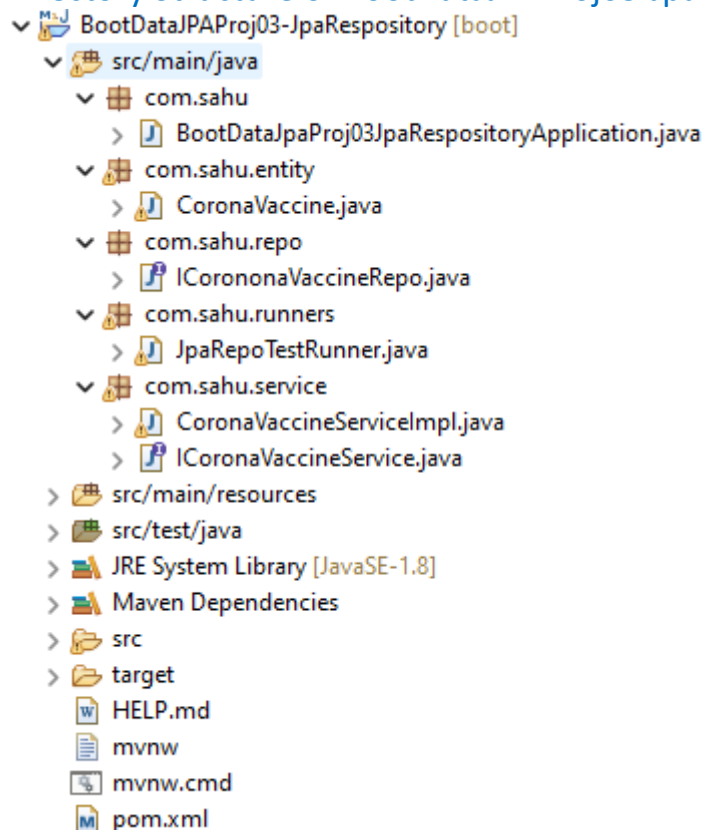
- This method performs select operations in the DB table by matching given object non-null property values with DB table columns applying "and" clause.
- On the retrieved records, it also performs sorting activity.

- ✚ While working with Spring Data JPA take methods for persistence operations in the following order,
 - a. CrudRepository methods
 - b. PagingAndSortingRepository methods
 - c. JpaRepository methods
 - d. Custom methods

Note:

- ✓ If we use more methods of CrudRepository, PagingAndSortingRepository methods then code becomes much portable across the multiple SQL and NoSQL DB s/w.
- ✓ Because most of Repository interfaces of NoSQL DB s/w like MongoRepository (I) extends from PagingAndSortingRepository.

Directory Structure of BootDataJPAProj03-JpaRepository:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.
- Place the following code within their respective files.

ICoronaVaccineService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.CoronaVaccine;

public interface ICoronaVaccineService {
    public List<CoronaVaccine>
searchVaccinesByGivenData(CoronaVaccine vaccine, boolean ascOrder,
String... properties);
}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
import org.springframework.data.domain.Sort.Direction;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;
import com.sahu.repo.ICoronaVaccineRepo;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICoronaVaccineRepo corononaVaccineRepo;

    @Override
    public List<CoronaVaccine>
searchVaccinesByGivenData(CoronaVaccine vaccine, boolean ascOrder,
String... properties) {
    // Prepare Example object
    Example example = Example.of(vaccine);
```

```

        //Prepare Sort Object
        Sort sort = Sort.by(ascOrder?Direction.ASC:Direction.DESC,
properties);
        List<CoronaVaccine> listVaccines =
        corononaVaccineRepo.findAll(example, sort);
        return listVaccines;
    }
}

```

ICorononaVaccineRepo.java

```

package com.sahu.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.CoronaVaccine;

public interface ICorononaVaccineRepo extends
JpaRepository<CoronaVaccine, Long> {

}

```

JpaRepoTestRunner.java

```

package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Component;

import com.sahu.entity.CoronaVaccine;
import com.sahu.service.ICoronaVaccineService;

@Component
public class JpaRepoTestRunner implements CommandLineRunner {

    @Autowired
    private ICoronaVaccineService coronaVaccineService;

    @Override
    public void run(String... args) throws Exception {
        try {

```

```

        CoronaVaccine vaccine = new CoronaVaccine(null,
"Sputnik", "Russie", "Russia", 567.8, 2);
        coronaVaccineService.searchVaccinesByGivenData(vaccine,
true, "price").forEach(System.out::println);
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
}

```

T getByld(ID id);

- Very much similar to findById(-) of CrudRepository.

ICoronaVaccineService.java

```

public CoronaVaccine getVaccineByRegNo(Long regNo);

```

CoronaVaccineServiceImpl.java

```

@Override
public CoronaVaccine getVaccineByRegNo(Long regNo) {
    return corononaVaccineRepo.getByld(regNo);
}

```

JpaRepoTestRunner.java

```

@Override
public void run(String... args) throws Exception {
    try {
        CoronaVaccine vaccine =
coronaVaccineService.getVaccineByRegNo(2l);
        if (vaccine!=null)
            System.out.println("Vaccine Details - "+vaccine);
        else
            System.out.println("Record not found");
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
}

```

application.properties

```
spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true
```

Note:

- ✓ `getOne(-)` and `getById(-)` methods implementation depends on underlying ORM framework used by Spring Data JPA.
- ✓ Since Spring Data JPA by default uses hibernate, to perform this load operation internally through lazy loading we need to place the additional property in `application.properties` file.
- ✓ `spring.jpa.properties.hibernate.enable_lazy_load_no_trans=true`, this property is required while working with `getById(-)` and `getOne()` methods.
- ✓ In older versions of Spring Boot, the above property is not required, rather `@Transactional` we need to place on the top of Service class or service class methods.

void deleteAllByIdInBatch(Iterable<ID> ids);

- Deletes the entities identified by the given ids using a single query. This kind of operation leaves JPAs first level cache and the database out of sync. Consider flushing the `EntityManager` before calling this method.
- Parameters:
ids - the ids of the entities to be deleted. Must not be null.

ICoronaVaccineService.java

```
public String removeVaccineByRegNos(Iterable<Long> regNos);
```

CoronaVaccineServiceImpl.java

```
@Override
public String removeVaccineByRegNos(Iterable<Long> regNos) {
    Iterable<CoronaVaccine> listVaccines =
corononaVaccineRepo.findAllById(regNos);
    if (((List<CoronaVaccine>) listVaccines).size() != 0) {
        corononaVaccineRepo.deleteAllByIdInBatch(regNos);
        return ((List<CoronaVaccine>) listVaccines).size() + " no. of
records are deleted";
    }
    return "Records not found for deletion.";
}
```

JpaRepoTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    try {
        System.out.println(coronaVaccineService.removeVaccineByRegNos(
            Arrays.asList(5l, 6l, 8l)));
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
```

Q. What is the difference between `deleteAllByIdInBatch` (`Iterable<ID> ids`) of `JpaRepository` and `deleteAllById` (`Iterable<ID> ids`) of `CrudRepository`?

Ans.

- `deleteAllByIdInBatch` (`Iterable<ID> ids`), this method generates for single delete query having in clause condition to delete the records of multiple given ids. If any id is not available then it will not generate any exception.
- `deleteAllById` (`Iterable<ID> ids`), this method generates multiple delete SQL queries to delete multiple records of given ids. If anyone id value is not available then exception will be thrown.

Q. What are the differences among the `findAll (-)` methods of different Repositories?

Ans.

<code>findAll ()</code> in Repository	Sorting	Pagination	Passing of Example object	Return Type
<code>JpaRepository</code>	Yes	No	Yes	<code>List<T></code>
<code>CrudRepository</code>	No	No	No	<code>Iterable<T></code>
<code>PagingAndSortingRepository</code>	Yes	Yes	No	<code>Iterable<T></code>

Note:

- ✓ `save(-)` method of `CrudRepository` performs save object or update object operation. In this process to commit or rollback the records it takes the support of `TransactionManager` (like `tx.commit()` or `tx.rollback()`).
- ✓ `saveAndFlush (-)` method of `JpaRepository` performs save object or update object operation. In this process it uses `flush ()` method to write changes to DB s/w without any `TransactionManager` support.

- ✓ New versions of Hibernate are not supporting only flush () call to perform non-select persistence operations, so in new versions the saveAndFlush () internally uses HibernateTransactionManager support.
- ✓ flush () writes the pending changes of different entity objects that are in their in-session object to DB table records.
- ✓ Prefer using CrudRepository and PagingAndSortingRepository a lot because these are common repositories while using Spring Data JDBC, Spring Data JPA and Spring Data for NoSQL DD s/w.

Custom Persistence operations in Spring Data JPA

- ✚ To perform persistence operation with our choice conditions, to execute JPQL (Java Persistence Query Language)/ HQL (Hibernate Query Language), SQL (Native SQL) queries, to call PL/SQL procedure and functions we need to place different types custom methods in Repository interfaces.
- ✚ The following custom methods are possible,
 - a. Finder methods (Only select operations)
 - b. @Query methods (To execute HQL/ JPQL, Native SQL select queries)
 - c. @Query + @Modifying methods (To execute HQL/JPQL, native SQL non-select Queries)

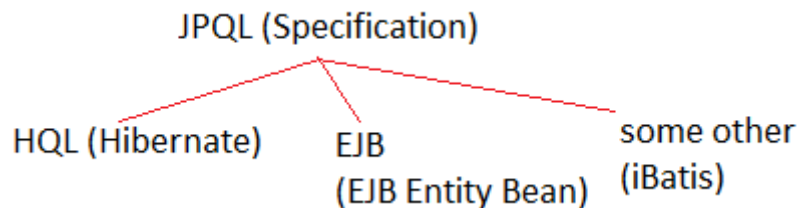
Finder methods

- These are custom abstract methods placed in our repository interface which will be converted into SELECT SQL queries internally.
- Supports both Entity select operations (selecting all column values of records) and Scalar select operations (selecting specific one or more column values of the records).

- E.g.:
 - select * from student (Entity query)
 - select sno, sadd from student (Scalar query)
 - select sadd from student (Scalar query)
 - select count(*) from student (Scalar query)

Scalar select operations can also be called as projection

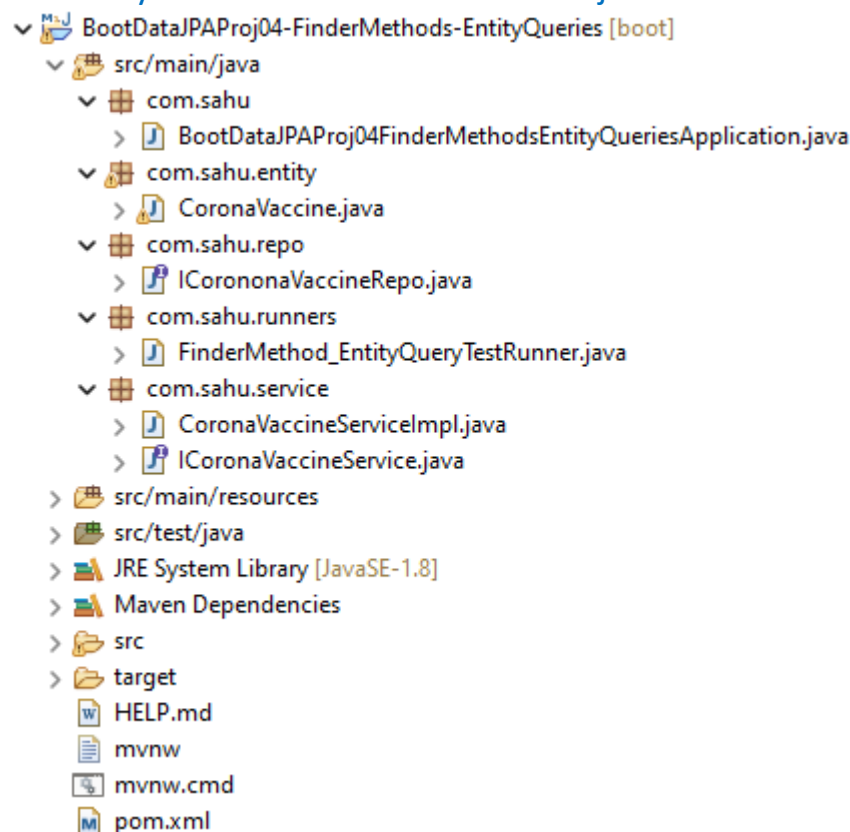
- We can prepare finder methods having one or more conditions with different clauses like and, or, in and etc.
- **Syntax:**
 public <Return type> findBy<Property name(s)><Condition (s)> (Params);



Note:

- ✓ The implementation of finder methods takes place in the Spring Data JPA generated InMemory proxy class
- ✓ If we take finder method without any condition the default condition that will be applied is "equals" (=) on given property/ column.
- ✓ In valid property name in finder methods are not allow. That property name must match as Entity class field name other that will be not taken.

Directory Structure of BootDataJPAProj04-FinderMethods-EntityQueries:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.
- Place the following code within their respective files.

ICoronaVaccineService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.CoronaVaccine;

public interface ICoronaVaccineService {
    public List<CoronaVaccine> fetchVaccinesByCompany(String
company);
}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;
import com.sahu.repo.ICorononaVaccineRepo;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICorononaVaccineRepo corononaVaccineRepo;

    @Override
    public List<CoronaVaccine> fetchVaccinesByCompany(String
company) {
        //return corononaVaccineRepo.findByCompany(company);
        //return
        corononaVaccineRepo.findByCompanyEquals(company);
        return corononaVaccineRepo.findByCompanyIs(company);
    }
}
```

ICorononaVaccineRepo.java

```
package com.sahu.repo;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.CoronaVaccine;

public interface ICorononaVaccineRepo extends
JpaRepository<CoronaVaccine, Long> {
    //SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
    REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COMPANY=?
    public List<CoronaVaccine> findByCompany(String company);
    public List<CoronaVaccine> findByCompanyEquals(String company);
    public List<CoronaVaccine> findByCompanyIs(String company);
}
```

FinderMethod_EntityQueryTestRunner.java

```
package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Component;

import com.sahu.service.ICoronaVaccineService;

@Component
public class FinderMethod_EntityQueryTestRunner implements
CommandLineRunner {

    @Autowired
    private ICoronaVaccineService coronaVaccineService;

    @Override
    public void run(String... args) throws Exception {
        coronaVaccineService.fetchVaccinesByCompany("Bharat-
Bio-Tech").forEach(System.out::println);
    }
}
```

Working finder methods as Custom method in our Repository (I):

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	findByAgeIsNull	... 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 %)
OrderBy	findByAgeOrderByLastNameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastNameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

ICoronaVaccineRepo.java

```

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE PRICE<?
public List<CoronaVaccine> findByPriceLessThan(Double price);

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE PRICE
BETWEEN ? AND ?
public List<CoronaVaccine> findByPriceBetween(Double startPrice,
Double endPrice);

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME LIKE ?/
'P%'
public List<CoronaVaccine> findByNameLike(String initChars);

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME LIKE ?%
public List<CoronaVaccine> findByNameStartingWith(String
startChars);

```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME LIKE %?  
public List<CoronaVaccine> findByNameEndingWith(String endChars);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME LIKE  
%?%  
public List<CoronaVaccine> findByNameContaining(String letters);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY IN (?,  
?, ?)  
public List<CoronaVaccine> findByCountryIn(List<String> countries);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY NOT  
IN (?, ?, ?)  
public List<CoronaVaccine> findByCountryNotIn(List<String>  
countries);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE PRICE>? ORDER  
BY PRICE ASC;  
public List<CoronaVaccine>  
findByPriceGreaterThanOrderByPriceAsc(Double startPrice);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY <> ?  
public List<CoronaVaccine> findByCountryNot(String country);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY <> ?  
public List<CoronaVaccine> findByCountryIsNot(String country);
```

```
//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,  
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE REG_NO = ?  
public Optional<CoronaVaccine> findByRegNo(Long regNo);
```

ICoronaVaccineService.java

```
public List<CoronaVaccine> fetchVaccinesByPriceLessThan(Double price);  
public List<CoronaVaccine> fetchVaccinesByPriceBetween(Double startPrice, Double endPrice);  
public List<CoronaVaccine> fetchVaccinesByNameLike(String initChars);  
public List<CoronaVaccine> fetchVaccinesByNameStartingWith(String startChars);  
public List<CoronaVaccine> fetchVaccinesByNameEndingWith(String endChars);  
public List<CoronaVaccine> fetchVaccinesByNameContaining(String letters);  
public List<CoronaVaccine> fetchVaccinesByCountryIn(String... countries);  
public List<CoronaVaccine> fetchVaccinesByCountryNotIn(List<String> countries);  
public List<CoronaVaccine> searchVaccinesByPriceAscOrder(Double price);  
public List<CoronaVaccine> searchVaccinesNotFromCountry(String country);  
public Optional<CoronaVaccine> getVaccineByRegNo(Long regNo);
```

CoronaVaccineServiceImpl.java

```
@Override  
public List<CoronaVaccine> fetchVaccinesByPriceLessThan(Double price) {  
    return corononaVaccineRepo.findByPriceLessThan(price);  
}  
  
@Override  
public List<CoronaVaccine> fetchVaccinesByPriceBetween(Double startPrice, Double endPrice) {  
    return corononaVaccineRepo.findByPriceBetween(startPrice, endPrice);  
}  
  
@Override  
public List<CoronaVaccine> fetchVaccinesByNameLike(String
```

```

initChars) {
    return corononaVaccineRepo.findByNameLike(initChars);
}

@Override
public List<CoronaVaccine> fetchVaccinesByNameStartingWith(String
startChars) {
    return
corononaVaccineRepo.findByNameStartingWith(startChars);
}

@Override
public List<CoronaVaccine> fetchVaccinesByNameEndingWith(String
endChars) {
    return
corononaVaccineRepo.findByNameEndingWith(endChars);
}

@Override
public List<CoronaVaccine> fetchVaccinesByNameContaining(String
letters) {
    return corononaVaccineRepo.findByNameContaining(letters);
}

@Override
public List<CoronaVaccine> fetchVaccinesByCountryIn(String...
countries) {
    return
corononaVaccineRepo.findByCountryIn(Arrays.asList(countries));
}

@Override
public List<CoronaVaccine>
fetchVaccinesByCountryNotIn(List<String> countries) {
    return corononaVaccineRepo.findByCountryNotIn(countries);
}

@Override
public List<CoronaVaccine> searchVaccinesByPriceAscOrder(Double
price) {

```

```

        return
        coronaVaccineRepo.findByPriceGreaterThanOrderByPriceAsc(price);
    }

    @Override
    public List<CoronaVaccine> searchVaccinesNotFromCountry(String
country) {
        //return coronaVaccineRepo.findByCountryNot(country);
        return coronaVaccineRepo.findByCountryIsNot(country);
    }

    @Override
    public Optional<CoronaVaccine> getVaccineByRegNo(Long regNo) {
        return coronaVaccineRepo.findByRegNo(regNo);
    }

```

FinderMethod_EntityQueryTestRunner.java

```

@Override
    public void run(String... args) throws Exception {
        coronaVaccineService.fetchVaccinesByPriceLessThan(400.5).forEach(S
ystem.out::println);
        coronaVaccineService.fetchVaccinesByPriceBetween(400.5,
500.0).forEach(System.out::println);

        coronaVaccineService.fetchVaccinesByNameLike("P%").forEach(Syste
m.out::println);

        coronaVaccineService.fetchVaccinesByNameLike("____").forEach(Syst
em.out::println);

        coronaVaccineService.fetchVaccinesByNameStartingWith("P").forEac
h(System.out::println);

        coronaVaccineService.fetchVaccinesByNameEndingWith("na").forEac
h(System.out::println);

        coronaVaccineService.fetchVaccinesByNameContaining("e").forEach(
System.out::println);
        coronaVaccineService.fetchVaccinesByCountryIn("India",

```

```

"USA").forEach(System.out::println);

    coronaVaccineService.fetchVaccinesByCountryNotIn(Arrays.asList("India", "USA")).forEach(System.out::println);

    coronaVaccineService.searchVaccinesByPriceAscOrder(400.0).forEach(
        System.out::println);

    coronaVaccineService.searchVaccinesNotFromCountry("USA").forEach(
        System.out::println);

    Optional<CoronaVaccine> optional =
        coronaVaccineService.getVaccineByRegNo(21);
    if (optional.isPresent())
        System.out.println(optional.get());
    else
        System.out.println("Record is not found");
}

```

Using Multiple properties-based conditions:

[ICorononaVaccineRepo.java](#)

```

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME=? AND
COUNTRY=?
    public List<CoronaVaccine> findByNameAndCountry(String name,
String country);
    public List<CoronaVaccine>
findByNameEqualsAndCountryEquals(String name, String country);

//SELECT REG_NO, NAME,COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE NAME LIKE 'p%'
OR PRICE BETWEEN ? AND ?
    public List<CoronaVaccine> findByNameLikeOrPriceBetween(String
initChar, Double startPrice, Double endPrice);

//SELECT REG_NO, NAME, COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY IN (?,
?, ?, ..) AND PRICE BETWEEN ? AND ?
    public List<CoronaVaccine>

```



```
findByCountryInAndPriceBetween(Collection<String> countries, Double
startPrice, Double endPrice);
```

```
//SELECT REG_NO, NAME, COMPANY, COUNTRY, PRICE,
REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE COUNTRY IN (?,
?, ?, ..) AND NAME IN (?, ?, ?) OR PRICE>=?
```

```
public List<CoronaVaccine>
findByCountryInAndNameInOrPriceGreaterThanOrEqual(Collection<String>
countries, Collection<String> names, Double price);
```

ICoronaVaccineService.java

```
public List<CoronaVaccine>
searchVaccinesByNameAndCountry(String name, String country);
public List<CoronaVaccine>
searchVaccinesByNameInitCharOrPriceRang(String initChars, Double
startRang, Double endRange);
public List<CoronaVaccine>
searchVaccinesByCountriesAndPriceRang(List<String> countries, Double
startRang, Double endRange);
public List<CoronaVaccine>
searchVaccinesByCountriesAndNamesOrByPrice(List<String> countries,
List<String> names, Double price);
```

CoronaVaccineServiceImpl.java

```
@Override
public List<CoronaVaccine>
searchVaccinesByNameAndCountry(String name, String country) {
    //return corononaVaccineRepo.findByNameAndCountry(name,
country);
    return
corononaVaccineRepo.findByNameEqualsAndCountryEquals(name,
country);
}
@Override
public List<CoronaVaccine>
searchVaccinesByNameInitCharOrPriceRang(String initChars, Double
startRang,
    Double endRange) {
    return
```

```

corononaVaccineRepo.findByNameLikeOrPriceBetween(initChars,
startRange, endRange);
    }

    @Override
    public List<CoronaVaccine>
searchVaccinesByCountriesAndPriceRang(List<String> countries, Double
startRange,
        Double endRange) {
        return
corononaVaccineRepo.findByCountryInAndPriceBetween(countries,
startRange, endRange);
    }

    @Override
    public List<CoronaVaccine>
searchVaccinesByCountriesAndNamesOrByPrice(List<String> countries,
List<String> names,
        Double price) {
        return
corononaVaccineRepo.findByCountryInAndNameInOrPriceGreaterThanOrEqual(countries, names, price);
    }

```

Note: Writing finder methods by involving multiple properties-based conditions is increasing the length of the method names and killing the readability. To overcome this problem, use @Query based custom methods of Repository (I) where we can write JPQL/ HQL or Native SQL queries directly linking with Custom methods declaration.

Performing Scalar Operations or Projections using Finder methods

- Selecting specific single column or multiple column values from DB table records is called Scalar operations or working with Projections.
- The finder methods of Spring Data JPA supports two types of Projections
 - a. Static Projections
[Allows to select specific fixed column values (single or multiple)]
 - b. Dynamic Projections
[Allows to select specific varying column values (single or multiple)]

Working Static Projections

- While dealing with Entity queries (which all column values from records) we can store each record into an object of Entity class.
- But while working with Scalar queries we need to define certain type to hold specific single or multiple column values, i.e., custom interface having declaration of getter methods for our choice single or multiple properties.

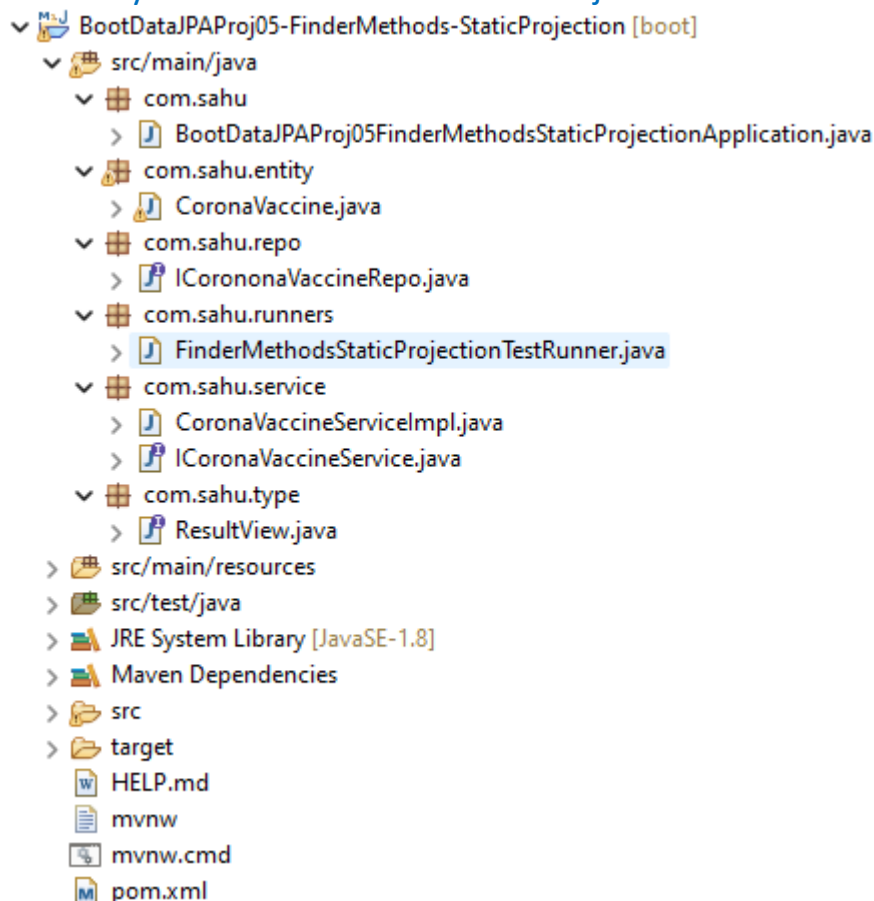
Step 1: Define one Type Interface having getter methods of specific single or multiple properties.

Step 2: Define method in our Repository (I) by involving the above type interface in the return type.

Step 3: Develop Service layer code for the finder method accordingly.

Step 4: Develop the runner class then Run the application.

Directory Structure of BootDataJPAProj05-FinderMethods-StaticProjection:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.

- Place the following code within their respective files.

ResultView.java

```
package com.sahu.type;

public interface ResultView {
    public String getName();
    public String getCountry();
}
```

ICorononaVaccineRepo.java

```
package com.sahu.repo;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.CoronaVaccine;
import com.sahu.type.ResultView;

public interface ICorononaVaccineRepo extends
JpaRepository<CoronaVaccine, Long> {
    public List<ResultView>
findByPriceGreaterThanOrEqualOrderByPrice(Double price);
}
```

ICoronaVaccineService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.type.ResultView;

public interface ICoronaVaccineService {

    public List<ResultView> searchVaccinesByPrice(Double price);

}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.repo.ICoronaVaccineRepo;
import com.sahu.type.ResultView;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICoronaVaccineRepo coronaVaccineRepo;

    @Override
    public List<ResultView> searchVaccinesByPrice(Double price) {
        return
        coronaVaccineRepo.findByPriceGreaterThanOrEqualOrderByPrice(price);
    }
}
```

FinderMethodsStaticProjectionTestRunner.java

```
package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.ICoronaVaccineService;

@Component
public class FinderMethodsStaticProjectionTestRunner implements
CommandLineRunner {

    @Autowired
```

```

private ICoronaVaccineService coronaVaccineService;

@Override
public void run(String... args) throws Exception {

    coronaVaccineService.searchVaccinesByPrice(400.0).forEach(vaccine -
> {
        System.out.println(vaccine.getName() + " " +
vaccine.getCountry());
    });

}

}

```

Note:

- ✓ Here Two InMemory Proxy classes will be generated
 1. For ResultView (I)
 2. For CoronaVaccineRepo (I)
- ✓ Here ResultView (I) act as custom type interfaces whose implementation class objects becomes capable holding specific multiple properties/ columns values whose getter methods are declared in the Type interface.

Core Java Recap

class Person {	class Student extends Person {	class Employee extends Person {
.....	Person {
.....
}	}
	
		}

Option 1:

```

public Object showDetails (Object object) {
    return object;
}

```

- We can pass and get not only Person class hierarchy object in fact any class object and code are not safe.
- E.g.
 Student stud = (Student) ref.showDetails(new Student());
 Employee emp = (Employee) ref.showDetails(new Employee());

- Here typecasting is required i.e., code is not type safe code.

Option 2:

```
public Person showDetails(Person person) {
    return object;
}
```

- Here we can pass and get only Person class hierarchy class objects and code is not type safe.
- E.g.
Student stud = (Student) ref.showDetails(new Student());
Employee emp = (Employee) ref.showDetails(new Employee());
- Here also typecasting is required i.e., code is not type safe code.

Option 3:

```
public <T> T showDetails(Class<T> clazz) {
    return object;
}
```

- Here we can pass and get any class object and code is type safe because of generics.
- E.g.
Student stud = ref.showDetails(Student.class);
Employee emp = ref.showDetails(Employee.class);
Date sysDate = ref.showDetails(Date.class);
- Here typecasting is not possible, so code is type safe code.

Option 4:

```
public <T extends Person> T showDetails(Class<T> clazz){
    return obj;
}
```

- Here we can pass and get only Person hierarchy class object and code is type safe code.
- E.g.
Student stud = ref.showDetails(Student.class);
Employee emp = ref.showDetails(Employee.class);
Date sysDate = ref.showDetails(Date.class); (X) Not possible
- Here type casting is not required. So, code is type safe code.

Dynamic Projections

- Here we can get varying specific single or multiple column values from DB table using the support of finder methods. For this we need to take

multiple type interfaces having hierarchy as show below.

- E.g.
interface view {

}
interface ResultView1 extends View {
 public String getName();
 public String getCompany();
}
interface ResultView2 extends View {
 public Long regNo();
 public Double getPrice();
 public String getCompany();
}
interface ResultView3 extends ResultView1 {
 public Double getPrice();
}

Example application on Dynamic Projections:

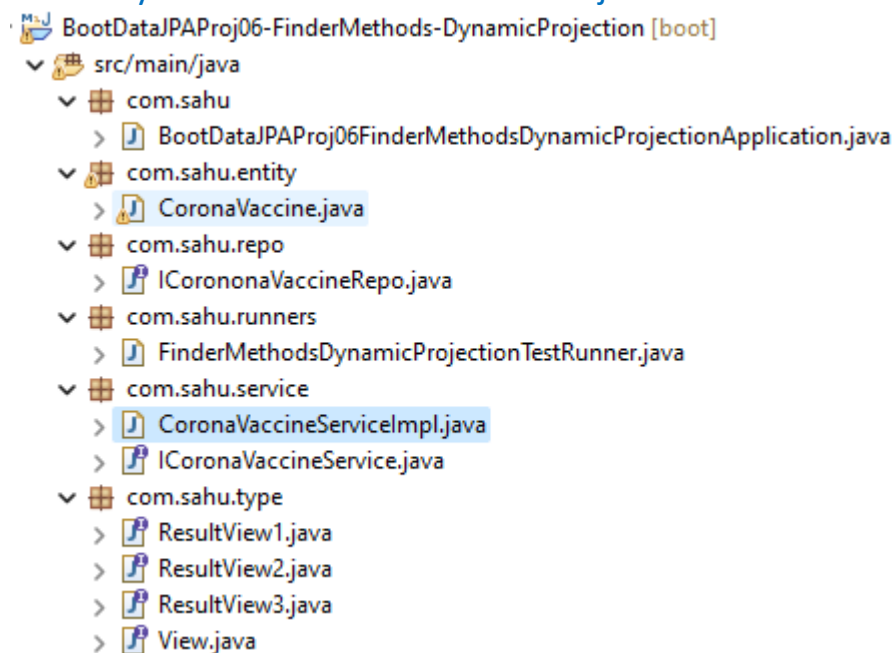
Step 1: Create multiple Type interface having hierarchy as shown above.

Step 2: Create finder method(s) in our Repository (I) involving the above Type interfaces.

Step 3: Develop service interface and service implementation class.

Step 4: Develop the Runner class and run the application.

Directory Structure of BootDataJPAProj06-FinderMethods-DynamicProjection:




```
> src/main/resources
> src/test/java
> JRE System Library [JavaSE-1.8]
> Maven Dependencies
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.
- Place the following code within their respective files.

View.java

```
package com.sahu.type;

public interface View {

}
```

ResultView1.java

```
package com.sahu.type;

public interface ResultView1 extends View {
    public String getName();
    public String getCountry();
}
```

ResultView2.java

```
package com.sahu.type;

public interface ResultView2 extends View {
    public Long getRegNo();
    public Double getPrice();
}
```

ResultView3.java

```
package com.sahu.type;

public interface ResultView3 extends ResultView1 {
    public Double getPrice();
}
```

ICorononaVaccineRepo.java

```
package com.sahu.repo;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.CoronaVaccine;
import com.sahu.type.View;

public interface ICorononaVaccineRepo extends
JpaRepository<CoronaVaccine, Long> {
    public <T extends View> List<T>
findByCompanyOrderByCompanyDesc(String company, Class<T> clazz);
}
```

ICoronaVaccineService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.type.View;

public interface ICoronaVaccineService {
    public <T extends View> List<T> searchVaccinesByCompany(String
company, Class<T> clazz);
}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;
```

```

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.repo.ICorononaVaccineRepo;
import com.sahu.type.View;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICorononaVaccineRepo corononaVaccineRepo;

    @Override
    public <T extends View> List<T> searchVaccinesByCompany(String
company, Class<T> clazz) {
        return
corononaVaccineRepo.findByCompanyOrderByCompanyDesc(company,
clazz);
    }
}

```

[FinderMethodsDynamicProjectionTestRunner.java](#)

```

package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.ICoronaVaccineService;
import com.sahu.type.ResultView1;
import com.sahu.type.ResultView2;
import com.sahu.type.ResultView3;

@Component
public class FinderMethodsDynamicProjectionTestRunner implements
CommandLineRunner {

```

```

@Autowired
private ICoronaVaccineService coronaVaccineService;

@Override
public void run(String... args) throws Exception {
    coronaVaccineService.searchVaccinesByCompany("Pyzer",
ResultView1.class)
        .forEach(vaccine ->
System.out.println(vaccine.getName() + " " + vaccine.getCountry()));
    System.out.println("-----");
    coronaVaccineService.searchVaccinesByCompany("Russie",
ResultView2.class)
        .forEach(vaccine ->
System.out.println(vaccine.getRegNo() + " " + vaccine.getPrice()));
    System.out.println("-----");
    coronaVaccineService.searchVaccinesByCompany("Russie",
ResultView3.class)
        .forEach(vaccine ->
System.out.println(vaccine.getName() + " " + vaccine.getCountry()+
"+vaccine.getPrice()));
    }
}

```

Note:

- ✓ We can't use Lombok API in interface to get the declaration of getter methods in the above Type interfaces.
- ✓ Lombok API gives setter and getter methods definitions not the declarations only when properties/ fields are available.

Limitations of finder methods in Spring Data JPA

- a. Only select operations are possible i.e., non-select operations are not possible.
- b. Aggregate operations are not possible.
- c. Group by operations are not possible.
- d. Working with scalar operations/ Projections is bit complex.
- e. While selecting data by using multiple properties and multiple conditions the method names becoming really very lengthy.
- f. Method names must be taken by following some conventions. That process kills the readability.

g. We can't call PL/ SQL procedure and functions.

Note:

- ✓ To overcome all the above problems, take the support of @Query methods using JPQL/ HQL or native SQL queries
- ✓ Use finder methods to perform single property/ column condition based select operations.

Working with @Query methods

- ✚ On the top custom methods declared in our Repository interface, we need to add @Query annotation either having HQL/ JPQL or Native SQL query.
- ✚ Method can have flexible signature and no need of following any naming conventions.
- ✚ **Syntax:**
@Query("<HQL/JPQL> or <native SQL>")
<return type> <method name> (params);

Note:

- ✓ HQL/ JPQL queries are DB s/w independent queries.
- ✓ Native SQL queries are DB s/w dependent queries.
- ✓ HQL/ JPQL queries are written using Entity class and its properties, whereas Native SQL queries are written using DB table name and its column name.

@Query method advantages:

- Support both select and non-select operations (except insert operation).
- Can work with either HQL/ JPQL or Native SQL queries ().
- Method names and signatures can be taken having flexibility.
- Can be used to call PL/ SQL Procedures and functions.
- Supports aggregate select operations.
- Supports group by, order by clauses.
- Supports to work with joins (To get data from two or multiple DB tables having implicit conditions).
and etc.

Q. Why @Query methods do not support INSERT Queries?

Ans. HQL/ JPQL INSERT query can't work with generators configured in the ID property of Entity class but in Spring Data JPA the id value must be generated using one or another generator. So, use repo.save(-) or repo.saveXxx(-)

methods for insert operation which can internally work with generators.

Q. How to perform Record insertion with partial values?

Ans. Use repo.save(-) method having Entity object with partial values (some properties will have NULL values) or keep @Transient on the top certain properties in the Entity class.

HQL/ JPQL

- It is objects-based DB s/w independent Query language.
- These Queries based persistence is portable across the multiple DB s/w.
- Supports both single row and bulk operations with our choice conditions.
- Supports both Entity and scalar (Projections) select operations.
- Supports both :<named> and positional params (?1, ?2, ?3, ...).
- Supports all where clause conditions.
- Supports joins.
- HQL/ JPQL queries based DDL operations are not possible.
- HQL/ JPQL does not support insert queries (There is no HQL/ JPQL insert query).
- HQL/ JPQL based PL/SQL programming is not possible.
- HQL/ JPQL keywords are not case-sensitive but the Entity class names and property names used the same queries are case-sensitive.

Note:

- ✓ Every HQL/JPQL will be converted in to underlying DB s/w specific SQL query.
- ✓ Learning curve of HQL/JPQL is very less because it is very much similar to SQL.

SQL> SELECT * FROM CORONA_VACCINE;

HQL/JPQL> FROM CoronaVaccine;

(or)

HQL/JPQL> FROM com.sahu.entity.CoronaVaccine;

(or)

HQL/JPQL> FROM CoronaVaccine cv;

(or)

HQL/JPQL>SELECT cv FROM com.sahu.entity.CoronaVaccine cv;

Writing SELECT Keyword in HQL/JPQ Queries is optional if you are selecting all column/ property values from DB table.

```
SQL> UPDATE CORONA_VACCINE SET PRICE=PRICE+? WHERE COMPANY=?;
(SQL supports JDBC type positional params)
HQL/JPQL> UPDATE CoronaVaccine SET price=price+?1 WHERE company=?2;
(or)
HQL/JPQL> UPDATE CoronaVaccine SET price=price+:addOnPrice WHERE
company=:manufacture;
```

Note:

- ✓ `?, ?, ?,` are called positional params.
- ✓ `?1, ?2, ?3,` are called ordinal positional params.
- ✓ `:addOnPrice, :manufacture` are called named params.
- ✓ From Hibernate 5.2 there is no support for positional params. So, we can use only ordinal positional params given by JPQL or named params (recommended).

```
SQL> SELECT REG_NO, NAME, COMPANY FROM CORONA_VACCINE WHERE  
COMPANY IN (?, ?);
```

```
HQL/JPQL> SELECT regNo, name, company FROM CoronaVaccine WHERE  
company IN (?1, ?2);
```

(or)

```
HQL/JPQL> SELECT regNo, name, company FROM CoronaVaccine WHERE  
company IN (:comp1, :comp2);
```

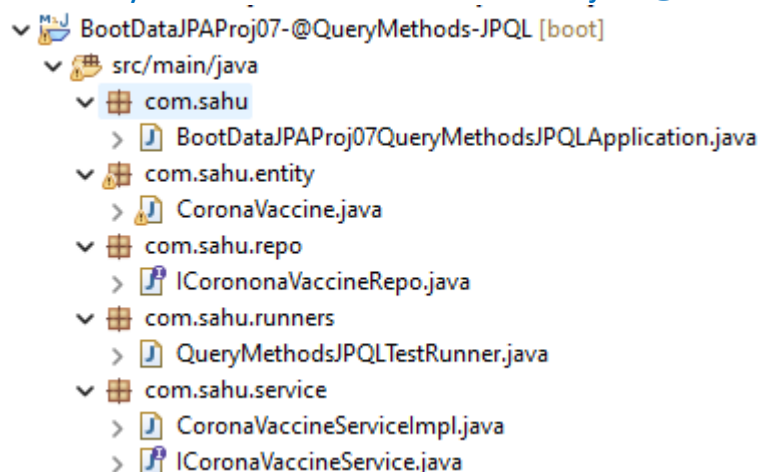
(or)

```
HQL/JPQL> SELECT cv.regNo, cv.name, cv.company FROM CoronaVaccine as cv
WHERE cv.company IN (:comp1,:comp2);
```

(or)

```
HQL/JPQL> SELECT cv.regNo, cv.name, cv.company FROM CoronaVaccine as cv
WHERE cv.company IN (?1,?2);
```

Directory Structure of BootDataJPAProj07-@QueryMethods-JPQL:



```

> src/main/resources
> src/test/java
> JRE System Library [JavaSE-1.8]
> Maven Dependencies
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml

```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties and CoronaVaccine.java file.
- Place the following code within their respective files.

ICorononaVaccineRepo.java

```

package com.sahu.repo;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.sahu.entity.CoronaVaccine;

public interface ICorononaVaccineRepo extends
JpaRepository<CoronaVaccine, Long> {
    // @Query("FROM com.sahu.entity.CoronaVaccine WHERE
company=?1")
    // @Query("FROM CoronaVaccine WHERE company=?1")
    // @Query("FROM CoronaVaccine as cv WHERE cv.company=?1")
    // @Query("SELECT cv FROM CoronaVaccine as cv WHERE
cv.company=?1")
    // @Query("SELECT cv FROM CoronaVaccine as cv WHERE
cv.company=?") throws java.lang.IllegalArgumentException: JDBC style
parameters (?) are not supported for JPA queries.
    @Query("FROM CoronaVaccine WHERE company=:vendor")
    public List<CoronaVaccine> searchVaccinesByCompany(String
vendor);
}

```


ICoronaVaccineService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.CoronaVaccine;

public interface ICoronaVaccineService {

    public List<CoronaVaccine> fetchVaccinesByCompany(String
company);

}
```

CoronaVaccineServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;
import com.sahu.repo.ICorononaVaccineRepo;

@Service("vaccineService")
public class CoronaVaccineServiceImpl implements ICoronaVaccineService {

    @Autowired
    private ICorononaVaccineRepo corononaVaccineRepo;

    @Override
    public List<CoronaVaccine> fetchVaccinesByCompany(String
company) {
        return
corononaVaccineRepo.searchVaccinesByCompany(company);
    }

}
```

QueryMethodsJPQLTestRunner.java

```
package com.sahu.runners;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.ICoronaVaccineService;

@Component
public class QueryMethodsJPQLTestRunner implements
CommandLineRunner {

    @Autowired
    private ICoronaVaccineService coronaVaccineService;

    @Override
    public void run(String... args) throws Exception {

        coronaVaccineService.fetchVaccinesByCompany("Pyzer").forEach(System.out::println);
    }

}
```

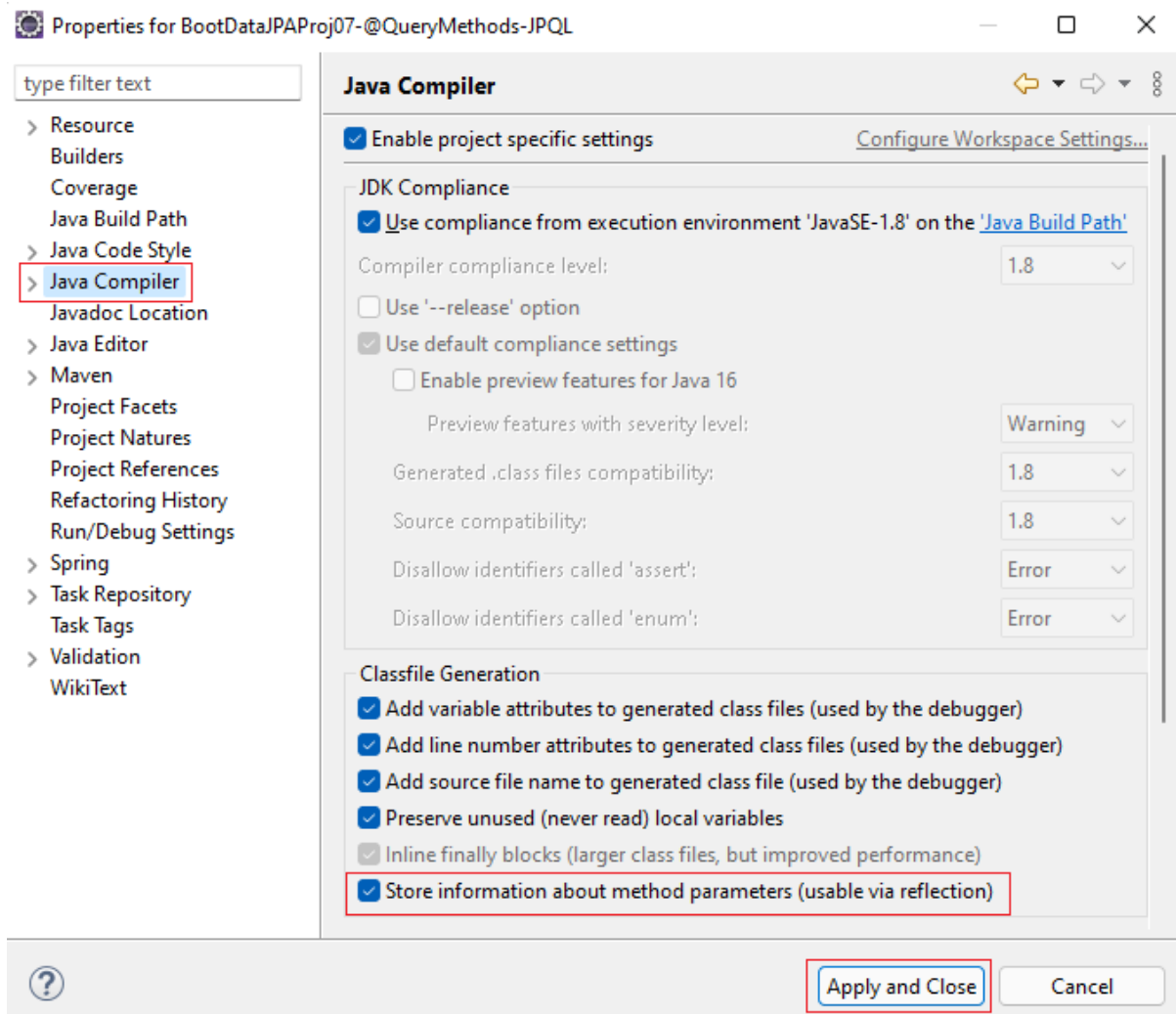
Note:

- ✓ The method parameter values will be bound with named param values automatically if their names are matching otherwise, we need to use @Param explicitly.

ICorononaVaccineRepo.java

```
@Query("FROM CoronaVaccine WHERE company=:comp")
public List<CoronaVaccine>
searchVaccinesByCompany(@Param("comp")String vendor);
```

- ✓ If you are getting exception though names are matching then either use @Param or enable following settings in the Eclipse Project. Go to Project Properties -> Java Compiler then -> check the following option there [Store information about method parameters].

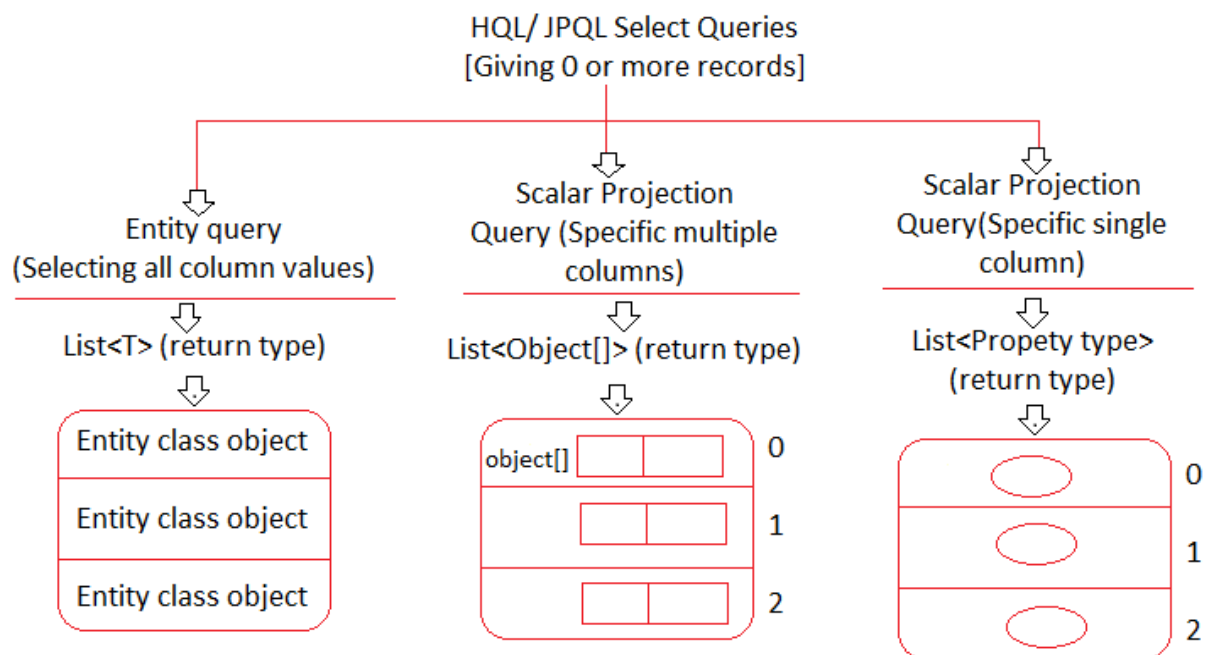


Important observations about HQL/ JPQL Query parameters:

- JDBC style plain positional parameters (?, ?, ..) are not allowed.
- Only JPA style ordinal positional parameters (?1, ?2, ?3, ...) are allowed.
- Ordinal Positional parameter index must start with 1 and should continue in increment sequence without any gap.
- We can take parameters in the HQL/ JPQL Query only representing input values i.e., we should not take representing HQL/ JPQL keywords, Entity class name and entity property names.
- If HQL/ JPQL query's named parameters (:<name>) names and custom method parameter names are matching then no need of placing @Param annotations in the before java method parameters otherwise placing them is mandatory.
- Prefer working with named parameters more compare to the ordinal positional parameters. More positional params makes to give sequence index for the parameters.
- We can't use both positional params and named params together at once in an HQL/ JPQL query.

E.g.

- FROM CoronaVaccine WHERE price>=?1 AND price<=?2 (valid)
- FROM CoronaVaccine WHERE price>=?1 AND price<=?3 (invalid)
- FROM CoronaVaccine WHERE price>=?0 AND price<=?1 (invalid)
- FROM CoronaVaccine WHERE price>=?2 AND price<=?1 (valid) Java method argument values goes to query params in reverse order
- FROM CoronaVaccine WHERE price>=?1 AND price<=:max (invalid)
- FROM CoronaVaccine WHERE price>=:min AND price<=?2 (invalid)
- FROM CoronaVaccine WHERE price>=? AND price<=? (invalid)
- FROM ?1 WHERE price>=?2 AND price<=?2 (invalid)
- FROM CoronaVaccine WHERE price>=?1 AND price<=?1 (technically valid) But writing this kind query is meaningless
- FROM CoronaVaccine ?1 price>=?2 AND price<=?3 (invalid)
- FROM CoronaVaccine WHERE :prop>=:min AND :prop<=:max (invalid)



Note: No need to take separate type interface for scalar/ projection operations.

ICoronaVaccineRepo.java

```

//Entity Query selecting all column values
@Query("FROM CoronaVaccine WHERE company IN(:company1,
:company2, :company3) ORDER BY company")
public List<CoronaVaccine> searchVaccinesByCompanies(String
company1, String company2, String company3);
//Here we can not take List<String> companies, String... companies
  
```

(var args)/ String[] companies as the param type

```
//Scalar Query selecting multiple column values
@Query("SELECT name, company, price FROM CoronaVaccine
WHERE name IN(:name1, :name2)")
    public List<Object[]> searchVaccineDetailsByNames(String name1,
String name2);

//Scalar Query selecting One column values
@Query("SELECT name FROM CoronaVaccine WHERE price BETWEEN
:min AND :max")
    public List<String> searchVaccineNamesByPriceRange(Double min,
Double max);
```

ICoronaVaccineService.java

```
    public List<CoronaVaccine> fetchVaccinesByCompanies(String
company1, String company2, String company3);
    public List<Object[]> fetchVaccinesByNames(String name1, String
name2);
    public List<String> fetchVaccineNamesByPriceRange(Double min,
Double max);
```

CoronaVaccineServiceImpl.java

```
@Override
    public List<CoronaVaccine> fetchVaccinesByCompanies(String
company1, String company2, String company3) {
        return
corononaVaccineRepo.searchVaccinesByCompanies(company1, company2,
company3);
    }

@Override
    public List<Object[]> fetchVaccinesByNames(String name1, String
name2) {
        return
corononaVaccineRepo.searchVaccineDetailsByNames(name1, name2);
    }

@Override
```

```

    public List<String> fetchVaccineNamesByPriceRange(Double min,
Double max) {
        return
corononaVaccineRepo.searchVaccineNamesByPriceRange(min, max);
    }

```

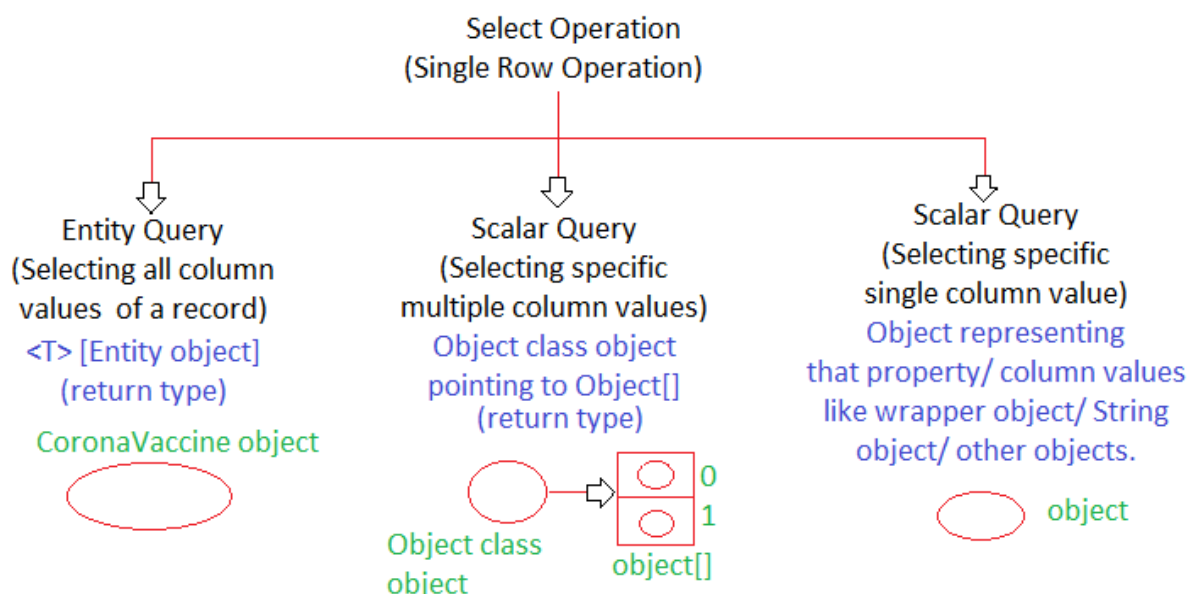
QueryMethodsJPQLTestRunner.java

```

@Override
    public void run(String... args) throws Exception {
        coronaVaccineService.fetchVaccinesByCompanies("Pyzer",
"Russie", "Pyzer").forEach(System.out::println);
        coronaVaccineService.fetchVaccinesByNames("Pyzer",
"Pyzer").forEach(vaccine-> System.out.println(vaccine[0]+" "+vaccine[1]+"
"+vaccine[2]));
        coronaVaccineService.fetchVaccinesByNames("Pyzer",
"Pyzer").forEach(vaccine-> {
            for (Object val : vaccine)
                System.out.println(val+" ");
            System.out.println();
        });
        coronaVaccineService.fetchVaccineNamesByPriceRange(400.0,
600.0).forEach(System.out::println);
    }

```

@Query methods for Single Row Operations



Q. Why should we go for @Query method that give single row when we have direct findById(-) or getById(-) or getOne(-) methods in pre-defined Repository interfaces?

Ans. The pre-defined methods findById(-) or getById(-) or getOne(-) methods in pre-defined Repository interfaces take id value (PK column value) as the criteria value/ condition value to get that single row, but we can design our own single row Entity query by taking other unique column value of DB table as the criteria/ condition value.

Note:

- ✓ If these methods found more than 1 record then we get Caused by: [javax.persistence.NonUniqueResultException](#): query did not return a unique result: 2.
- ✓ Arrays are objects in Java, so the reference variable of java.lang.Object class can refer any array.
- ✓ List<Object[]> - specific multiple column values of multiple records.
- ✓ Object[] - specific multiple column values of single record.
- ✓ List<T> - for multiple records.
- ✓ <T> or Optional<T> - for single record.
- ✓ List<Property type object> - for getting single column values from multiple records.
- ✓ <Property type object> - for getting single column value of single record.

ICorononaVaccineRepo.java

```
// Entity Query giving Single row
@Query("FROM CoronaVaccine WHERE name=:vaccineName")
public Optional<CoronaVaccine> searchVaccineByName(String
vaccineName);

// Scalar Query giving multiple column values of Single row
@Query("SELECT name, comapny, country FROM CoronaVaccine
WHERE name=:vaccineName")
public Object searchVaccineDataByName(String vaccineName);

// Scalar Query giving single column values of Single row
@Query("SELECT country FROM CoronaVaccine WHERE
name=:vaccineName")
public String searchVaccineCountryByName(String vaccineName);
```

ICoronaVaccineService.java

```
public Optional<CoronaVaccine> fetchVaccineByName(String vaccineName);  
public Object fetchVaccineDataByName(String vaccineName);  
public String fetchVaccineCountryByName(String vaccineName);
```

CoronaVaccineServiceImpl.java

```
@Override  
public Optional<CoronaVaccine> fetchVaccineByName(String vaccineName) {  
    return  
    coronaVaccineRepo.searchVaccineByName(vaccineName);  
}  
  
@Override  
public Object fetchVaccineDataByName(String vaccineName) {  
    return  
    coronaVaccineRepo.searchVaccineDataByName(vaccineName);  
}  
  
@Override  
public String fetchVaccineCountryByName(String vaccineName) {  
    return  
    coronaVaccineRepo.searchVaccineCountryByName(vaccineName);  
}
```

QueryMethodsJPQLTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
    Optional<CoronaVaccine> vaccine =  
    coronaVaccineService.fetchVaccineByName("sputnik");  
    if(vaccine.isPresent())  
        System.out.println(vaccine.get());  
    else  
        System.out.println("Record not found");  
  
    Object objVaccine =  
    coronaVaccineService.fetchVaccineDataByName("sputnik");  
    Object[] vaccineDetails = (Object[]) objVaccine;
```



```

        for(Object val : vaccineDetails)
            System.out.println(val+" ");
        System.out.println();

        System.out.println(coronaVaccineService.fetchVaccineCountryByName("sputnik"));
    }

```

HQL/ JPQL supports Aggregate operations

ICorononaVaccineRepo.java

```

@Query("SELECT COUNT(*) FROM CoronaVaccine")
public Long getVaccinesCount();

@Query("SELECT COUNT(*), MAX(price), MIN(price), sum(price), AVG(price) FROM CoronaVaccine WHERE price>=:min AND price<=:max")
public Object getVaccinesAggregateDataByPriceRange(Double min, Double max);

```

ICoronaVaccineService.java

```

public Long fetchVaccinesCount();
public Object fetchVaccineAggregateDetailsByPriceRange(Double min, Double max);

```

CoronaVaccineServiceImpl.java

```

@Override
public Long fetchVaccinesCount() {
    return corononaVaccineRepo.getVaccinesCount();
}

@Override
public Object fetchVaccineAggregateDetailsByPriceRange(Double min, Double max) {
    return corononaVaccineRepo.getVaccinesAggregateDataByPriceRange(min, max);
}

```

QueryMethodsJPQLTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    System.out.println(coronaVaccineService.fetchVaccinesCount());

    Object result[] = (Object[])
coronaVaccineService.fetchVaccineAggregateDetailsByPriceRange(400.0,
600.);

    System.out.println("Vaccines Count : "+result[0]);
    System.out.println("Max Price : "+result[1]);
    System.out.println("Min Price : "+result[2]);
    System.out.println("Sum Price : "+result[3]);
    System.out.println("Avg Price : "+result[4]);
}
```

Performing non-select Operations using HQL/ JPQL in @Query methods

- ✚ INSERT HQL/ JPQL is not supported, so use ses.save(-) method for it
- ✚ For other non-select Operations like DELETE, UPDATE we need to place @Query + @Modifying Annotations.
- ✚ @ Modifying, to indicate the given HQL/ JPQL query is non-select HQL/ JPQL query.
- ✚ If you are not taking separate service class then we need to place @Transactional on the top of Repository (I) or @Query methods + @Modifying methods otherwise we need to place on the top of Service class methods or Service class itself.

ICoronaVaccineRepo.java

```
@Modifying
@Query("UPDATE CoronaVaccine SET price=:newPrice WHERE
country=:countryName")
@Transactional
public int updatePriceByCountry(Double newPrice, String
countryName);

@Modifying
@Query("DELETE FROM CoronaVaccine WHERE price BETWEEN
```

```
:startPrice AND :endPrice")
    @Transactional
    public int deleteVaccineByPriceRange(Double startPrice, Double
endPrice);
```

ICoronaVaccineService.java

```
    public int ModifiyVaccinePriceByCountry(Double newPrice, String
countryName);
    public int removeVaccinesByPriceRange(Double startPrice, Double
endPrice);
```

CoronaVaccineServiceImpl.java

```
    @Override
    public int ModifiyVaccinePriceByCountry(Double newPrice, String
countryName) {
        return corononaVaccineRepo.updatePriceByCountry(newPrice,
countryName);
    }

    @Override
    public int removeVaccinesByPriceRange(Double startPrice, Double
endPrice) {
        return
corononaVaccineRepo.deleteVaccineByPriceRange(startPrice, endPrice);
    }
```

QueryMethodsJPQLTestRunner.java

```
    @Override
    public void run(String... args) throws Exception {
        int count =
coronaVaccineService.ModifiyVaccinePriceByCountry(560.0, "Russia");
        System.out.println("Number of vaccines updated : "+count);

        System.out.println("Number of vaccines deleted :
"+coronaVaccineService.removeVaccinesByPriceRange(400.0, 600.0));
    }
```

Using @Query methods having Native SQL queries

- ✚ Native SQL queries means the underlying DB s/w specific SQL queries i.e., if the underlying DB s/w is Oracle then Oracle specific SQL queries, and etc.
- ✚ Use this Native SQL queries to perform those operations which are not possible with HQL/ JPQL queries like,
 - Insert query
 - Date operations (sysdate in Oracle, now () in MySQL)
 - DDL queries
 - To call PL/ SQL procedure and functions
- ✚ Use @Query annotation with "nativeQuery=true" param, to work with Native SQL queries.
- ✚ While working with Native SQL queries we can place 3 types of parameters
 - a. JDBC style positional parameters (?, ?, ?, ..)
 - b. JPA style ordinal positional parameters (?1, ?2, ?3,)
 - c. Named parameters (:<name1>, :<name2>,)
- ✚ Native SQL queries will be written using DB table names and its column names.

ICorononaVaccineRepo.java

```
@Modifying
@Query(value="INSERT INTO CORONA_VACCINE VALUES(?, ?, ?, ?, ?,
?)", nativeQuery = true)
@Transactional
public int insertVaccine(Long regNo, String company, String country,
String name, Double price, Integer dosesCount);

@Query(value="SELECT SYSDATE FROM DUAL", nativeQuery = true)
public Date getSystemDate();

@Modifying
@Query(value="CREATE TABLE TEMP(COL1 NUMBER(5), COL2
VARCHAR2(20))", nativeQuery = true)
@Transactional
public int createTempTable();
```

ICoronaVaccineService.java

```
public int registerVaccine(Long regNo, String company, String country, String name, Double price, Integer dosesCount);  
public Date fetchSystemDate();  
public int createTempTable();
```

CoronaVaccineServiceImpl.java

```
@Override  
public int registerVaccine(Long regNo, String company, String country, String name, Double price, Integer dosesCount) {  
    return corononaVaccineRepo.insertVaccine(regNo, company, country, name, price, dosesCount);  
}  
  
@Override  
public Date fetchSystemDate() {  
    return corononaVaccineRepo.getSystemDate();  
}  
  
@Override  
public int createTempTable() {  
    return corononaVaccineRepo.createTempTable();  
}
```

QueryMethodsJPQLTestRunner.java

```
public void run(String... args) throws Exception {  
    int isInserted = coronaVaccineService.registerVaccine(858L, "Serim", "India", "Covaxin", 458.0, 2);  
    System.out.println(isInserted==0?"Record not inserted":"Record inserted");  
  
    System.out.println("System date and time : "+coronaVaccineService.fetchSystemDate());  
  
    int tableCreated = coronaVaccineService.createTempTable();  
    System.out.println(tableCreated);  
    System.out.println(tableCreated==0?"Table has created":"Table has not created");  
}
```

Calling PL/ SQL Procedure and Functions using Spring Data JPA

- ✚ Instead of writing same persistence logic or business logic in every module of the Project, it is recommended to write them as PL/ SQL procedure or function (stored procedures or functions) only for one time in DB s/w and call them in multiple modules of the Project.
- ✚ In order to avoid important logics for authentication and authorization from developers of the project, they will be developed as PL/ SQL procedures or functions only their signature details will be exposed to the developers.

Use case:

1. Authentication, Authorization logics will be developed as PL/ SQL procedure or function and will be called from multiple modules.
2. Attendance calculation logics will be developed as PL/ SQL procedure or function and will be called from multiple modules.
3. Billing logics, settling claim amount logics, calculating student CGPA, SGPA logics, some kind of batch processing operations and etc. logics will be developed as PL/ SQL procedures or functions.

Note: A typical Java project contains,

- 60% to 70% persistence logics using JPQL/ HQL + native SQL + O-R mapping logics.
- 30% to 40% persistence logics using PL/ SQL procedures or functions.
- ✚ PL/ SQL programming is specific to each DB s/w because we use SQL queries in PL/ SQL programming.
- ✚ PL/ SQL procedure does not return a value but we can get multiple results/ outputs using OUT params
- ✚ PL/ SQL functions return a value but we can get multiple results/ outputs using return value + OUT params.
- ✚ 5 results from PL/ SQL procedure then take 5 OUT params.
- ✚ 5 results from PL/ SQL function then take 4 OUT params and 1 return value.
- ✚ PL/ SQL procedure or function parameters not only contains type they also maintain mode.
- ✚ The modes are IN (default), OUT, INOUT.
E.g.,
PL/ SQL logic
z: = x + y; x, y - IN parameters and z is OUT parameter.

PL/SQL logic

$x := x * x;$ x - INOUT parameter.

- ✚ A Cursor (collection) is InMemory variable of Oracle PL/ SQL programming that is capable of holding bunch of records (0 or more records) given by SELECT SQL query execution.
- ✚ SYS_REFCURSOR is pre-defined cursor data type of Oracle PL/ SQL programming and it can be used like this

details SYS REFCURSOR; - cursor variable declaration
open details for

`SELECT * FROM STUDENT;`

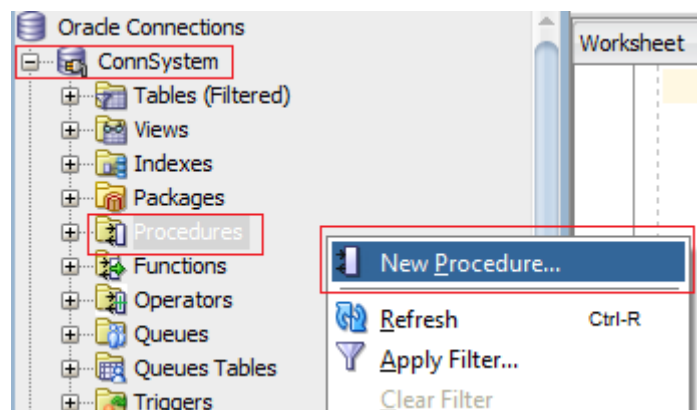
The records given by Select SQL query will be stored into cursor variable (details)

- Oracle PL/ SQL programming's cursor is similar to JDBC ResultSet object. While receiving outputs from Cursor type OUT parameters, we can take the support of JDBC ResultSet object.

Creating PL/ SQL Procedure in Oracle DB s/w using SQL Developer

Step 1: Open SQL Developer and go to your one of connection and expand it.

Step 2: Right click on Procedures and click on New Procedure...



Step 3: Give the Procedure name and required parameters then click on OK. Then you will get below code

```
CREATE OR REPLACE PROCEDURE P_GET_VACCINE_BY_PRICERANGE  
(  
    STARTPRICE IN FLOAT  
    , ENDPRICE IN FLOAT
```

```
, DETAILS OUT SYS_REFCURSOR
) AS
BEGIN
    NULL;
END P_GET_VACCINE_BY_PRICERANGE;
```

Schema: SYSTEM

Name: P_GET_VACCINE_BY_PRICERANGE

☐ Add New Source In Lowercase

Name	Mode	No Copy	Data Type	Default Value
STARTPRICE	IN	<input type="checkbox"/>	FLOAT	
ENDPRI	IN	<input type="checkbox"/>	FLOAT	
DETAILS	OUT	<input type="checkbox"/>	SYS_REFCURSOR	

Buttons: Help, OK, Cancel

Step 4: Write the below code and save it and compiled it.

```
CREATE OR REPLACE PROCEDURE P_GET_VACCINE_BY_PRICERANGE
(
    STARTPRICE IN FLOAT
, ENDPRI IN FLOAT
, DETAILS OUT SYS_REFCURSOR
) AS
BEGIN
    OPEN DETAILS FOR
        SELECT REG_NO, NAME, COMPANY, COUNTRY, PRICE,
            REQUIRED_DOSE_COUNT FROM CORONA_VACCINE WHERE
                PRICE>=STARTPRICE AND PRICE<=ENDPRI;

END P_GET_VACCINE_BY_PRICERANGE;
```


Note: In Spring Data JPA we can use EntityManager object (like HB Session object) to call PL/ SQL procedure. If we add Spring Data JPA starter this EntityManager object will created through AutoConfiguration, so it can be injected to Service class.

Process to develop Spring Data JPA application as Spring Boot application using Gradle

Step 1: Make sure that Build Ship plugin (Gradle plugin) is installed in Eclipse IDE.

Note: In new versions of Eclipse, it is built-in plugin.

Step 2: Create Spring Boot starter project using Gradle as the build tool. File - New - Spring Starter Project then give the information and must choose Type is Gradle Project then click in Next.

New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

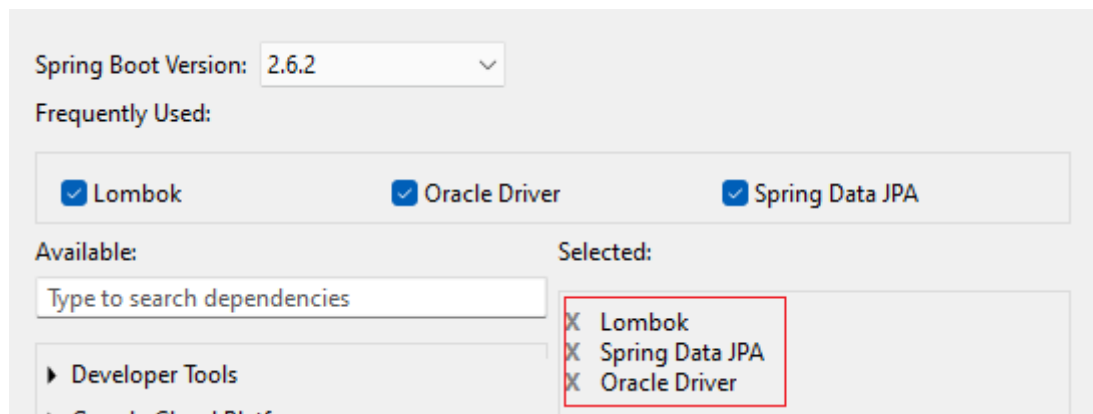
Package:

Working sets

☐ Add project to working sets

Working sets:

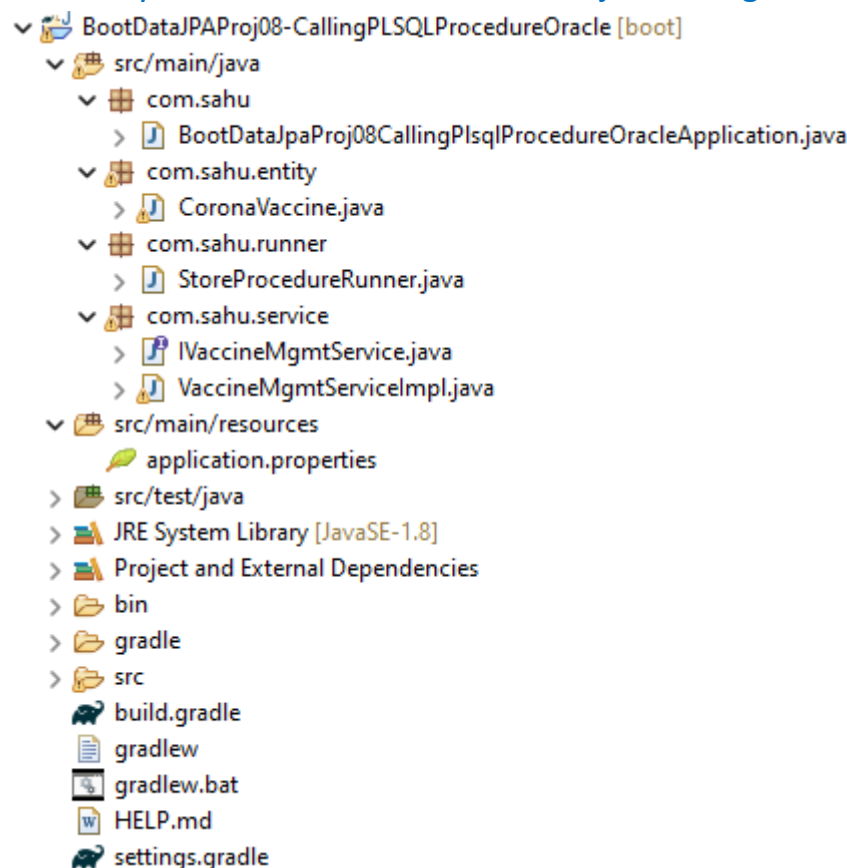
Step 3: Then choose the following starter then click on Next then Finish.



Step 4: Add more dependencies in build.gradle using dependencies {...} (enclosure), you can collect the other jar dependencies from [\[mvnrepository\]](https://mvnrepository.com).

Step 5: Develop the project and run it.

Directory Structure of BootDataJPAProj08-CallingPLSQLProcedureOracle:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starter is enough and copy the application.properties

and CoronaVaccine.java file.

- Place the following code within their respective files.

IVaccineMgmtService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.CoronaVaccine;

public interface IVaccineMgmtService {
    public List<CoronaVaccine> searchVaccineByPriceRange(Double min,
        Double max);
}
```

VaccineMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.ParameterMode;
import javax.persistence.StoredProcedureQuery;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.CoronaVaccine;

@Service
public class VaccineMgmtServiceImpl implements IVaccineMgmtService {

    @Autowired
    private EntityManager manager;

    @Override
    public List<CoronaVaccine> searchVaccineByPriceRange(Double min,
        Double max) {
        //Create Stored Procedure object representing PL/ SQL
    }
}
```

```

procedure
    StoredProcedureQuery query =
manager.createStoredProcedureQuery("P_GET_VACCINE_BY_PRICERANGE"
, CoronaVaccine.class);
    //register Parameters of PL/ SQL procedure
    query.registerStoredProcedureParameter(1, Double.class,
ParameterMode.IN);
    query.registerStoredProcedureParameter(2, Double.class,
ParameterMode.IN);
    query.registerStoredProcedureParameter(3,
CoronaVaccine.class, ParameterMode.REF_CURSOR);
    //Set values to parameter
    query.setParameter(1, min);
    query.setParameter(2, max);
    //Call PL/ SQL procedure
    List<CoronaVaccine> listVaccine = query.getResultList();
    return listVaccine;
}

```

StoreProcedureRunner.java

```

package com.sahu.runner;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.IVaccineMgmtService;

@Component
public class StoreProcedureRunner implements CommandLineRunner {

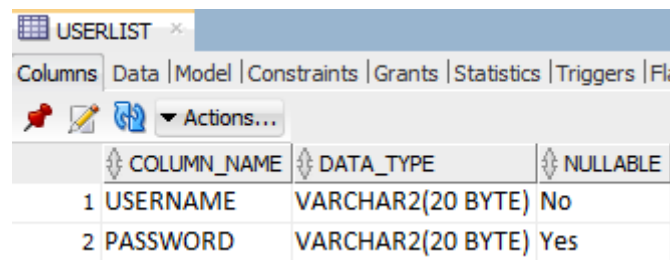
    @Autowired
    private IVaccineMgmtService vaccineMgmtService;

    @Override
    public void run(String... args) throws Exception {
        vaccineMgmtService.searchVaccineByPriceRange(400.0,
900.0).forEach(System.out::println);
    }
}

```

Calling PL/ SQL Procedure that performs Authentication from Spring Data JPA application

Step 1: Create a table in Oracle DB s/w and fill some information.



The screenshot shows the 'USERLIST' table structure in Oracle SQL Developer. The table has two columns: 'USERNAME' and 'PASSWORD', both of type 'VARCHAR2(20 BYTE)'. The 'USERNAME' column is not nullable, while the 'PASSWORD' column is nullable.

	COLUMN_NAME	DATA_TYPE	NULLABLE
1	USERNAME	VARCHAR2(20 BYTE)	No
2	PASSWORD	VARCHAR2(20 BYTE)	Yes

Step 2: Create the following procedure.

```
CREATE OR REPLACE PROCEDURE P_AUTHENTICATION
(
  UNAME IN VARCHAR2
, PWD IN VARCHAR2
, RESULT OUT VARCHAR2
) AS
  CNT NUMBER (5);
BEGIN
  SELECT COUNT(*) INTO CNT FROM USERLIST WHERE
  USERNAME=UNAME AND PASSWORD=PWD;
  IF (CNT<>0) THEN
    RESULT:='Valid Credentials';
  ELSE
    RESULT:='Invalid Credentials';
  END IF;
END P_AUTHENTICATION;
```

Step 3: Develop the code for calling the PL/ SQL Procedure and run the application.

[IVaccineMgmtService.java](#)

```
public String authenticate(String userName,String password);
```

[VaccineMgmtServiceImpl.java](#)

```
@Override
public String authenticate(String userName, String password) {
    //Create Stored Procedure object representing PL/ SQL
    procedure
```

```

        StoredProcedureQuery query =
manager.createStoredProcedureQuery("P_AUTHENTICATION");
        //register Parameters of PL/ SQL procedure
        query.registerStoredProcedureParameter(1, String.class,
ParameterMode.IN);
        query.registerStoredProcedureParameter(2, String.class,
ParameterMode.IN);
        query.registerStoredProcedureParameter(3, String.class,
ParameterMode.OUT);
        //Set values to parameter
        query.setParameter(1, userName);
        query.setParameter(2, password);
        //Call PL/ SQL procedure
        query.execute();
        //Gather results out parameter value
        String result = (String) query.getOutputParameterValue(3);
        return result;
    }

```

StoreProcedureRunner.java

```

@Override
public void run(String... args) throws Exception {
    System.out.println(vaccineMgmtService.authenticate("Raja",
"Rani"));
}

```

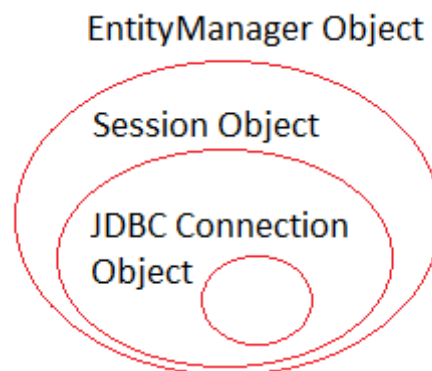
Different methods according to OUT parameter type:

Procedure OUT Parameter type	Method to be invoked on Query Object
Cursor returning bunch of records	query.getResultList();
Cursor returning single record	query.getSingleResult ();
Other than cursor like number, varchar2 (If multiple OUT params are there)	query.getOutputParameterValue(-) (Call this method for multiple times)

Note:

- ✓ Only in standalone Spring applications, we need to depend on the above kind of PL/ SQL procedure and EntityManager object for Authentication (Security).

- ✓ In case of Spring MVC, Spring Rest applications we use Spring Security or Spring OAuth or JWT tokens concepts.
- ✓ Spring Data JPA can be used in standalone Spring applications, Spring MVC Web applications, Spring Rest applications to make them taking with DB s/w.
- ✓ As of now there is no direct support to call PL/ SQL function of DB s/w (stored function) from Spring Data JPA applications. But we can do it by writing plain JDBC style CallableStatement object code by unwrapping Session, Connection objects from EntityManager object.



Callback:

- The method that will be called by underlying server or container or framework is called Callback method.
- The interface that contains the declaration of callback method is called Callback interface.
- Callback will be called by underlying server/ container automatically having container supplied objects and we use those objects in the methods implementation.
- The Servlet container created request, response objects are exposed the programmer as the params of service (-, -) method by calling service (-, -) as callback method by servlet container.

Note: In Hibernate API we have `RetruningWork<T> (I)` as callback interface cum functional interface.

Method:

Modifier and Type	Method and Description
T	<code>execute(Connection connection)</code> Execute the discrete work encapsulated by this work instance using the supplied connection.

I `execute(Connection connection)` throws [SQLException](#)

Execute the discrete work encapsulated by this work instance using the supplied connection.

// Gets Session object

Session ses = manger.unwrap(Session.class);

// LAMDA Expression based Anonymous inner class object as the argument value of ses.doReturningWork(-) method

ses.doReturningWork(conn -> {

..... //JDBC code using CallableStatement object to call PL/ SQL function

.....

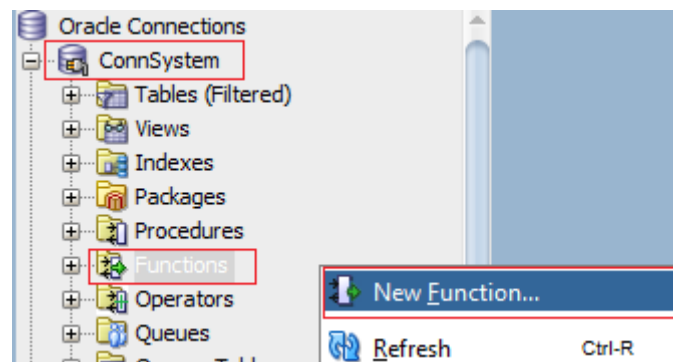
});

Q. Write a PL/SQL function that gives employee name, job, sal, deptno from emp DB table based on given empno?

Ans.

Step 1: Open SQL Developer and go to your one of connection and expand it.

Step 2: Right click on Functions and click on New Function...



Step 3: Give the Function name, return type and required parameters then click on OK.

Then you will get below code

```
CREATE OR REPLACE FUNCTION FX_GET_EMP_DETAILS
(
  ENO IN NUMBER
, NAME OUT VARCHAR2
, DESG OUT VARCHAR2
, SALARY OUT NUMBER
) RETURN NUMBER AS
BEGIN
  RETURN NULL;
END FX_GET_EMP_DETAILS;
```


Create Function

Schema: SYSTEM

Name: FX_GET_EMP_DETAILS

☐ Add New Source In Lowercase

Return Type: NUMBER

Parameters: name

Name	Mode	No Copy	Data Type	Default Value
ENO	IN	<input type="checkbox"/>	NUMBER	
NAME	OUT	<input type="checkbox"/>	VARCHAR2	
DESG	OUT	<input type="checkbox"/>	VARCHAR2	
SALARY	OUT	<input type="checkbox"/>	NUMBER	

Help OK Cancel

Step 4: Write the below code and save it and compiled it.

```
CREATE OR REPLACE FUNCTION FX_GET_EMP_DETAILS
(
  ENO IN NUMBER
, NAME OUT VARCHAR2
, DESG OUT VARCHAR2
, SALARY OUT NUMBER
) RETURN NUMBER AS
  DNO NUMBER(3);
BEGIN
  SELECT ENAME, JOB, SAL, DEPTNO INTO NAME, DESG, SALARY, DNO
FROM EMP WHERE EMPNO=ENO;
  RETURN DNO;
END FX_GET_EMP_DETAILS;
```

For calling the function place the below code with their in respective files

[IVaccineMgmtService.java](#)

```
public Object[] getEmpDetailsByEno(int eno);
```

VaccineMgmtServiceImpl.java

```
@Override
    public Object[] getEmpDetailsByEno(int eno) {
        //Unwrap session object from EntityManager
        Session ses = manager.unwrap(Session.class);
        //Work with ReturningWork<T> interface to write plain JDBC
code to call PL/ SQL fuction
        Object obj[] = ses.doReturningWork(conn->{
            //Create CallableStatement object
            CallableStatement cs = conn.prepareCall("{?=call
FX_GET_EMP_DETAILS(?,?,?,?)}");
            //Register OUT parameters
            cs.registerOutParameter(1, Types.INTEGER);
            cs.registerOutParameter(3, Types.VARCHAR);
            cs.registerOutParameter(4, Types.VARCHAR);
            cs.registerOutParameter(5, Types.FLOAT);
            //Set value to IN parameters
            cs.setInt(2, eno);
            //Call PL/ SQL Function
            cs.execute();
            //Gater Results from return OUT parameter
            Object result[] = new Object[4];
            result[0] = cs.getInt(1);   result[1] = cs.getInt(3);
            result[2] = cs.getInt(4);   result[4] = cs.getInt(5);
            return result;
        });
        return obj;
    }
```

StoreProcedureRunner.java

```
@Override
    public void run(String... args) throws Exception {
        Object result[] =
vaccineMgmtService.getEmpDetailsByEno(7499);
        System.out.println("EMP Name : "+result[1]);
        System.out.println("EMP Job : "+result[2]);
        System.out.println("EMP Name : "+result[3]);
        System.out.println("EMP Dept No : "+result[0]);
    }
```

Calling PL/ SQL Procedure of MySQL DB s/w from Spring Data JPA application

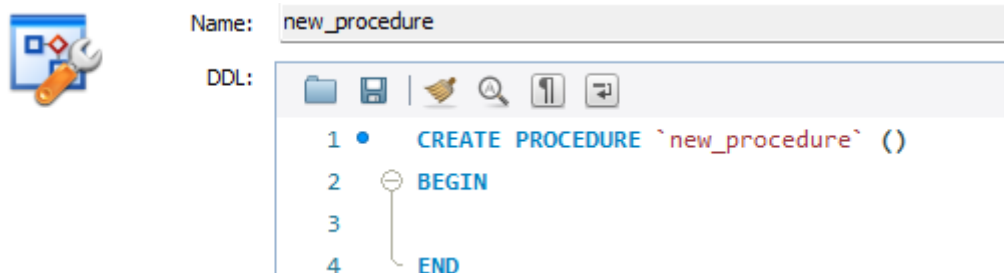
- In MySQL there is no support for Cursors. Actually, Cursors are not required in MySQL PL/ SQL programming.
- To get bunch of records given by select SQL query there is no need of taking any kind of Cursor in PL/ SQL programming of MySQL.

Step 1: Launch MySQL Workbench, select your connection.

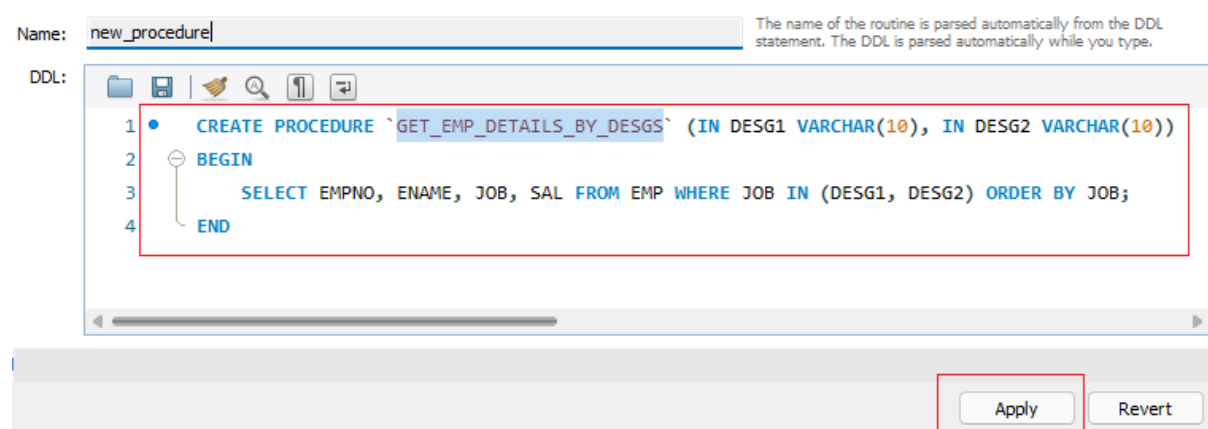
Step 2: Expand your Logical Schema and click on Stored Procedures then click on Create Stored Procedure...



Step 3: Then you will get the following dummy Procedure



Step 4: Write the below code in the body part of procedure then click on Apply, and again Apply then click on Finish .



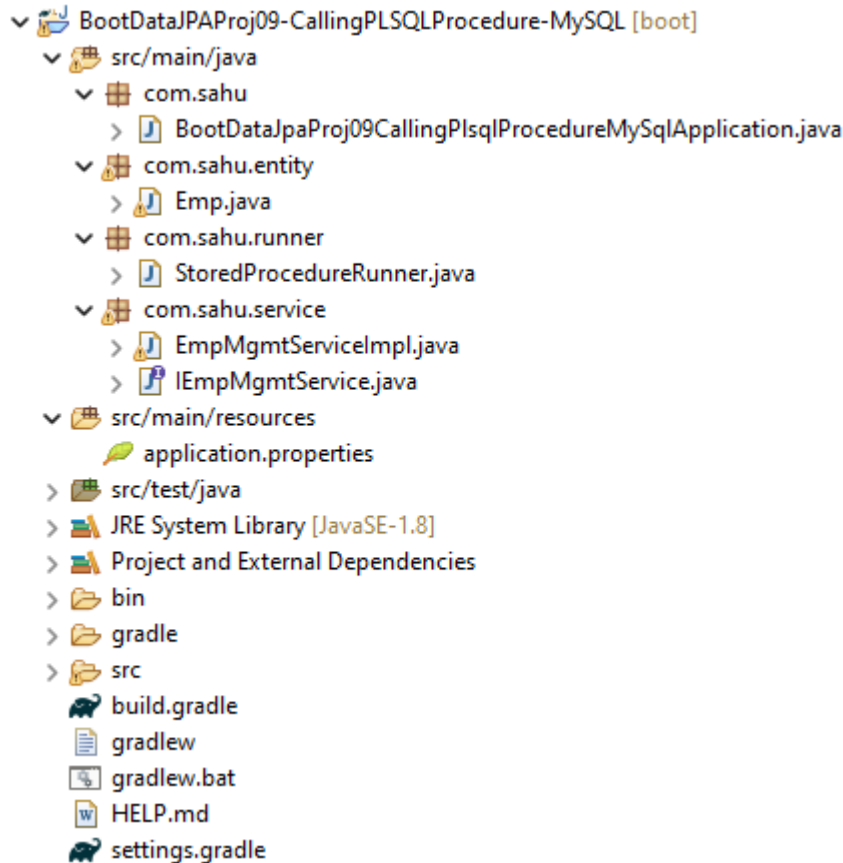
```
CREATE PROCEDURE `GET_EMP_DETAILS_BY_DESGS` (IN DESG1 VARCHAR(10),  
IN DESG2 VARCHAR(10))
```

BEGIN

SELECT EMPNO, ENAME, JOB, SAL FROM EMP WHERE JOB IN (DESG1, DESG2) ORDER BY JOB;

END

Directory Structure of BootDataJPAProj09-CallingPLSQLProcedure-MySQL:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Choose the below starters during creating the project.

```
X Lombok
X Spring Data JPA
X MySQL Driver
```

- Place the following code within their respective files.

application.properties

```
#DataSource Configuration
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///ntspbms714db
spring.datasource.username=root
spring.datasource.password=root
```

#JPA Hibernate properties

spring.jpa.database-platform=[org.hibernate.dialect.MySQL8Dialect](#)

spring.jpa.show-sql=[true](#)

spring.jpa.hibernate.ddl-auto=[create](#)

#Other possible values create, validate, create-drop

Emp.java

```
package com.sahu.entity;
```

```
import java.io.Serializable;
```

```
import javax.persistence.Column;
```

```
import javax.persistence.Entity;
```

```
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
```

```
import javax.persistence.Table;
```

```
import lombok.AllArgsConstructor;
```

```
import lombok.Data;
```

```
import lombok.NoArgsConstructor;
```

```
@Entity
```

```
@Table(name="emp")
```

```
@Data
```

```
@AllArgsConstructor
```

```
@NoArgsConstructor
```

```
public class Emp implements Serializable {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private Integer empno;
```

```
    @Column(length=20)
```

```
    private String ename;
```

```
    @Column(length=20)
```

```
    private String job;
```

```
    private Integer sal;
```

```
}
```

IEmpMgmtService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.Emp;

public interface IEmpMgmtService {
    public List<Emp> searchEmployeesByDesgs(String desg1, String
desg2);
}
```

EmpMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.ParameterMode;
import javax.persistence.StoredProcedureQuery;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.Emp;

@Service
public class EmpMgmtServiceImpl implements IEmpMgmtService {

    @Autowired
    private EntityManager manager;

    @Override
    public List<Emp> searchEmployeesByDesgs(String desg1, String desg2)
    {
        //Create StoredProcedureQuery object
        StoredProcedureQuery query =
manager.createStoredProcedureQuery("GET_EMP_DETAILS_BY_DESGS",
Emp.class);
        //Register Parameters of PL/SQL procedure
    }
```

```

        //register Parameters of PL/ SQL procedure
        query.registerStoredProcedureParameter(1, String.class,
ParameterMode.IN);
        query.registerStoredProcedureParameter(2, String.class,
ParameterMode.IN);
        //Set values to parameter
        query.setParameter(1, desg1);
        query.setParameter(2, desg2);
        //Call PL/ SQL procedure
        List<Emp> listEmps = query.getResultList();
        return listEmps;
    }
}

```

StoredProcedureRunner.java

```

package com.sahu.runner;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.IEmpMgmtService;

@Component
public class StoredProcedureRunner implements CommandLineRunner {

    @Autowired
    private IEmpMgmtService empMgmtService;

    @Override
    public void run(String... args) throws Exception {
        empMgmtService.searchEmployeesByDesgs("CLERK",
"MANAGER").forEach(System.out::println);
    }
}

```

Working with Date values using Java 8 Date and Time API

✚ This Date and Time API also Known as JODA date and time API.

In java 8

- `LocalTime`: To set time values and to get current time
- `LocalDate`: To set date values and to get current date
- `LocalDateTime`: To set date and time values and to get current date and time

In Oracle DB s/w

- `DATE` datatype - To store date values
- `TIMESTAMP` datatype - To store date and time values

Note: Time data type is not given.

In MySQL DB s/w

- `DATE` datatype - To store date values
- `DATETIME` datatype - To store data and time values
- `TIMESTAMP` datatype - To store date and time values

Directory Structure of BootDataJPAProj10-WorkigWithDataAndTime:

```
▼ BootDataJPAProj10-WorkingWithDataAndTime [boot]
  ▼ src/main/java
    ▼ com.sahu
      > BootDataJpaProj10WorkingWithDataAndTimeApplication.java
    ▼ com.sahu.entity
      > Customer.java
    ▼ com.sahu.repo
      > ICustomerRepo.java
    ▼ com.sahu.runner
      > DateAndTimeTestRunner.java
    ▼ com.sahu.service
      > CustomerMgmtServiceImpl.java
      > ICustomerMgmtService.java
  ▼ src/main/resources
    application.properties
  > src/test/java
  > JRE System Library [JavaSE-1.8]
  > Project and External Dependencies
  > bin
  > gradle
  > src
  build.gradle
  gradlew
  gradlew.bat
```


- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Choose the below starters during creating the project.

```
X Lombok
X Spring Data JPA
X MySQL Driver
X Oracle Driver
```

- Place the following code within their respective files.

application.properties

```
#DataSource Configuration
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///ntspbms714db
spring.datasource.username=root
spring.datasource.password=root

#JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
#Other possible values create, validate, create-drop
```

Customer.java

```
package com.sahu.entity;

import java.io.Serializable;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
```

```

import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Entity
@Table(name="CUSTOMER_INFO")
@Data
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
public class Customer implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer custId;
    @NonNull
    @Column(length = 20)
    private String custName;
    @NonNull
    @Column(length = 20)
    private String custAddr;
    @NonNull
    private LocalDateTime dob;
    @NonNull
    private LocalTime timeOfPurchase;
    @NonNull
    private LocalDate dateOfPurchase;
}

```

ICustomerRepo.java

```

package com.sahu.repo;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.Customer;

public interface ICustomerRepo extends JpaRepository<Customer, Integer>
{
}

```

ICustomerMgmtService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.entity.Customer;

public interface ICustomerMgmtService {
    public String registerCustomer(Customer customer);
    public List<Customer> getAllCustomer();
}
```

CustomerMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.Customer;
import com.sahu.repo.ICustomerRepo;

@Service("customerService")
public class CustomerMgmtServiceImpl implements ICustomerMgmtService
{
    @Autowired
    private ICustomerRepo customerRepo;

    @Override
    public String registerCustomer(Customer customer) {
        int idVal = customerRepo.save(customer).getCustId();
        return "Customer is saved with id : "+idVal;
    }

    @Override
    public List<Customer> getAllCustomer() {
        return customerRepo.findAll();
    }
}
```

DateAndTimeTestRunner.java

```
package com.sahu.runner;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.entity.Customer;
import com.sahu.service.ICustomerMgmtService;

@Component
public class DateAndTimeTestRunner implements CommandLineRunner {

    @Autowired
    private ICustomerMgmtService customerMgmtService;

    @Override
    public void run(String... args) throws Exception {
        Customer customer = new Customer("Raja", "Hyd",
            LocalDateTime.of(1999, 10, 23, 15, 10, 3),
            LocalTime.now(),
            LocalDate.now());

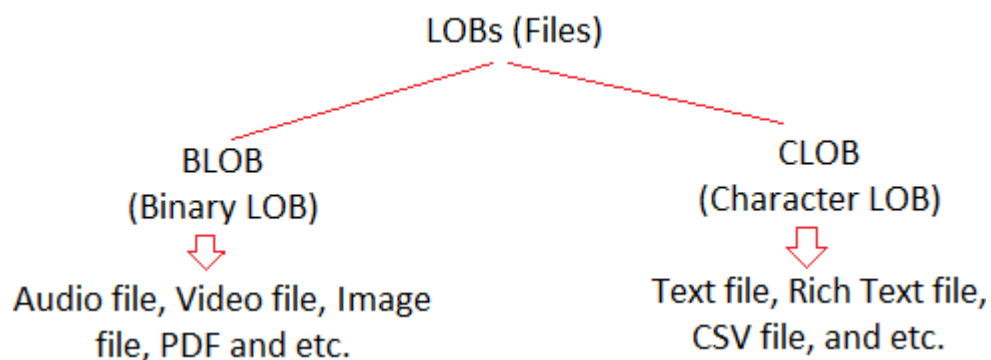
        System.out.println(customerMgmtService.registerCustomer(customer
));

        customerMgmtService.getAllCustomer().forEach(System.out::println);
    }
}
```

Working with Large Objects (LOBS)

- ✚ All most in all major DB s/w are supporting Large Object by giving different data types.
- ✚ Oracle gives BLOB, CLOB Data types.

- MySQL gives BLOB, Tiny Text, Medium Text, Large text data types.
- Only in standalone applications the large objects (files) will be saved directly in DB table columns.
E.g., Standalone matrimony apps, standalone job portal apps and etc.
- In web applications, distributed applications the LOBs (files) will be saved in Server machine file system and their address paths will be written to DB table columns as String values (To avoid performance issue with DB s/w).
E.g., Matrimony web applications, Job portal app, social networking app and etc.



- In some web applications and Distributed applications, the LOBS (files) will be saved in separate CDN (Content Delivery Network) server, DMS (Document Management System) server, NAS (Network Attached Storage), S3 Bucket servers and etc.
- We get token ids from those server representing LOBs (files) insertion and these token ids will be inserted into DB table columns as String values.
- The Entity class should have byte [] and char [] type properties representing BLOB, CLOB type data.
- In ORM frameworks (like hibernate) we take entity class as Serializable class for two reasons
 - Now a days, we are making entity class itself as the Model class (Java Bean) to carry data across the multiple layers like (Controller - Service - Repository/DAO class). Sometimes we need to send same data over the network from current project to another project so it needs to take Entity class as Serializable class (Flipkart sending Card Details to PayPal over the network).
 - The ORM framework like hibernate supports Disk caching in second level caching i.e., it uses hard disk memory through serialization while performing second Level caching. For This Entity classes should be taken as Serializable.

- ✚ On Entity class @Lob on byte [] property makes Spring Data JPA/ ORM f/w giving BLOB column in DB table.
- ✚ On Entity class @Lob on char [] property makes Spring Data JPA/ ORM f/w giving CLOB column in DB table.
- ✚ The boolean property data (true/ false) of Entity class object will be stored in DB table as number column value (0/ 1).

Directory Structure of BootDataJPAProj11-WorkingWithLOBS:

```

v BootDataJPAProj11-WorkingWithLOBS [boot]
v src/main/java
  v com.sahu
    > BootDataJpaProj11WorkingWithLobsApplication.java
  v com.sahu.entity
    > MarriageSeeker.java
  v com.sahu.repo
    > IMatrimonyRepo.java
  v com.sahu.runner
    > LOBsTestRunner.java
  v com.sahu.service
    > IMatrimonyServiceMgmt.java
    > MatrimonyServiceImpl.java
v src/main/resources
  application.properties
v src/test/java
  > com.sahu
> JRE System Library [JavaSE-11]
> Project and External Dependencies
> bin
> gradle
> src
  build.gradle
  gradlew
  gradlew.bat
  HELP.md
  settings.gradle

```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Choose the below starters during creating the project.

```

X Lombok
X Spring Data JPA
X MySQL Driver
X Oracle Driver

```

- Copy the application.properties file from previous project.
- Place the following code within their respective files.

MarriageSeeker.java

```
package com.sahu.entity;

import java.io.Serializable;
import java.time.LocalDateTime;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
public class MarriageSeeker implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @NonNull
    @Column(length = 20)
    private String name;
    @NonNull
    @Column(length = 20)
    private String address;
    @NonNull
    private LocalDateTime dob;
    @NonNull
    private Boolean indian;
    @NonNull
    @Lob
```

```
private byte[] photo;
@NotNull
@Lob
private char[] biodata;
}
```

IMatrimonyRepo.java

```
package com.sahu.repo;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.entity.MarriageSeeker;

public interface IMatrimonyRepo extends
PagingAndSortingRepository<MarriageSeeker, Long> {

}
```

IMatrimonyServiceMgmt.java

```
package com.sahu.service;

import java.util.Optional;

import com.sahu.entity.MarriageSeeker;

public interface IMatrimonyServiceMgmt {
    public String registerMarriageSeeker(MarriageSeeker
marriageSeeker);
    public Optional<MarriageSeeker> searchMarriageSeekerById(Long id);
}
```

MatrimonyServiceImpl.java

```
package com.sahu.service;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```



```

import com.sahu.entity.MarriageSeeker;
import com.sahu.repo.IMatrimonyRepo;

@Service("matrimonyService")
public class MatrimonyServiceImpl implements IMatrimonyServiceMgmt {

    @Autowired
    private IMatrimonyRepo matrimonyRepo;

    @Override
    public String registerMarriageSeeker(MarriageSeeker marriageSeeker)
    {
        return "Marriage Seeker information has Saved with id value : "
        +matrimonyRepo.save(marriageSeeker).getId();
    }

    @Override
    public Optional<MarriageSeeker> searchMarriageSeekerById(Long id)
    {
        return matrimonyRepo.findById(id);
    }
}

```

LOBsTestRunner.java

```

package com.sahu.runner;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.OutputStream;
import java.io.Reader;
import java.io.Writer;
import java.time.LocalDateTime;
import java.util.Optional;
import java.util.Scanner;

```

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.entity.MarriageSeeker;
import com.sahu.service.IMatrimonyServiceMgmt;

@Component
public class LOBsTestRunner implements CommandLineRunner {

    @Autowired
    private IMatrimonyServiceMgmt matrimonyServiceMgmt;

    @Override
    public void run(String... args) throws Exception {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter Person Name : ");
        String name = scanner.next();
        System.out.print("Enter Person Address : ");
        String address = scanner.next();
        System.out.print("Is Person Indian : ");
        boolean indian = scanner.nextBoolean();
        System.out.print("Enter Person Photo file complete Path : ");
        String photoPath = scanner.next();
        System.out.print("Enter Person Bio-Data file complete Path : ");
        String biodataPath = scanner.next();
        //Prepare byte[] representing photo file
        InputStream is = new FileInputStream(photoPath);
        byte[] photoData = is.readAllBytes();
        //Prepare char[] representing biodata file
        File file = new File(biodataPath);
        Reader reader = new FileReader(file);
        char[] biodataContent = new char[(int)file.length()];
        reader.read(biodataContent);
        //Prepare Entity class object
        MarriageSeeker marriageSeeker = new MarriageSeeker(name,
address,
                                LocalDateTime.of(1990, 11, 23, 12, 45),
                                indian, photoData, biodataContent);
    }
}

```

```

        System.out.println(matrimonyServiceMgmt.registerMarriageSeeker(m
arriageSeeker));

        Optional<MarriageSeeker> optional =
matrimonyServiceMgmt.searchMarriageSeekerById(4l);
        if (optional.isPresent()) {
            MarriageSeeker seeker = optional.get();
            System.out.println(seeker.getId()+"
"+seeker.getName()+" "+seeker.getAddress()+" "+seeker.getIndian());
            OutputStream os = new
FileOutputStream("retrieve_photo.png");
            os.write(seeker.getPhoto());
            os.flush();
            os.close();
            Writer writer = new FileWriter("retrieve_biodata.txt");
            writer.write(seeker.getBiodata());
            writer.flush();
            writer.close();
            System.out.println("LOBs are retrieved");
        }
        else {
            System.out.println("Record is not found");
        }
    }
}

```

To Run the same code in Oracle DB s/w environment just place the following code in application.properties.

application.properties

#DataSource Configuration

```

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

```

#JPA Hibernate properties

```

spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

```

Associations in Spring Data JPA

- It is also known as Relationships or Association Mapping or Multiplicity in Spring Data JPA or Hibernate.

Need of keeping DB tables in Association:

- Keeping multiple entities/ items data in single DB table as records is having the following problems
 - Data Redundancy (Duplication) Problem
 - Data Management Problem (Data Inconsistency Problem)

Problem:

customer_phone_details (DB table)

cno(n)	cname(vc2)	caddrs(vc2)	mobile(n)	provider(vc2)	type(vc2)
101	raja	hyd	999999999	airtel	residence
101	raja	hyd	99998887	vodafone	office
102	mahesh	vizg	99988673	jio	office
102	mahesh	vizg	99985553	vodafone	residence

Data Redundancy Problem

Solution 1: Keep multiple entities info in multiple DB tables like user information in separate DB tables and phone number details in separate DB table.

customer (DB table1)

cno(n)(pk)	cname(vc2)	caddrs(vc2)
101	raja	hyd
102	mahesh	vizg

phone_details (DB table2)

regNo(pk)	mobile(n)	provider(vc2)	type(vc2)
1	999999999	airtel	residence
2	99998887	vodafone	office
3	99988673	jio	office
4	99985553	vodafone	residence

- Now data redundancy problem is gone but data management/ navigation problem is created i.e., we can't access phone number details through customer table records and vice-versa.

Solution 2: Take two DB tables to maintain multiple entities info and keep them in relationship using FK (Foreign key column).

Advantages:

- No Data Redundancy Problem.

- b. No Data Navigation problem using FK column we can access child table records from parent DB table records and vice -versa.

customer (Parent DB table)			phone_details (Child DB table)			
cno(n)(pk)	cname(vc2)	caddrs(vc2)	regNo(pk)	mobile(n)	provider(vc2)	customer_no (fk with cno)
101	raja	hyd	1	999999999	airtel	101
102	maresh	vizg	2	99998887	vodafone	101
			3	99988673	jio	102
			4	99985553	vodafone	102

Note: If DB tables are in relationship, then the associated Entity classes should also take having relationship but we keep DB tables in relationship using FK column where as we keep Entity classes in relationship using Composition and Collections i.e., the way DB tables are in relationship is different from the way Entity classes are in relationship. To avoid this mismatch, use the advanced O-R mapping called Association Mapping.

Other mismatches:

- DB tables can't be there in the inheritance but the Entity classes can be there in the inheritance, so to map the Entity classes of inheritance with DB tables go for "Inheritance mapping".
- DB tables can't be there in the composition but the Entity classes can be there in the composition, so to map the Entity classes of composition with DB tables go for "Component mapping".
- DB tables do not have collection type columns but the Entity classes can have array/ collection type properties, so to map Entity classes with collection to DB tables use "Collection Mapping".

Spring Data JPA/ ORM framework supports 4 types associations:

- One to One
- One to Many
- Many To One
- Many to Many

E.g.,

- Student and Rank info (1 to 1)
- User and Phone Number (1 to M)
- Dept and Employee (1 to M)
- Student and Course (M to M)
- Faculty and Student (M to M)
- Driving License and Citizen (1 to 1)

- Passport and Citizen (1 to 1)
- Employee and Dept (M to 1)
- Student and College (M to 1)
- Doctor and Patient (M to M)
- Corona Vaccine and Citizen (1 to 1)

Associations in ORM frameworks or Spring Data JPA can be implemented in two modes

- Uni-Directional Association
(Either parent to child or child to parent access possible)
- Bi-Directional Association
(Parent to child and child to parent access possible)

Associations in ORM frameworks or Spring Data JPA can be built in Entity classes in two ways

- Using Reference type properties in Composition (non-collection property) (1 to 1 and M to 1)
- Using Collection type properties in Composition (1 to M and M to M)

Note:

- ✓ We can keep DB tables either Uni-directional or in Bi-directional association using single FK column.
- ✓ But in Java no FK column is available, so to keep Entity classes in association we have to use composition either collection or non-collection type properties.

One to Many Bi-Directional Association

- Bi-directional association means we can access parent objects from child objects and vice-versa.
- To build one to many Bi-Directional Association the parent class should have special property of type Collection (Set/ List/ Map) to hold bunch of child class objects, similarly child class should have special property of type Parent class (Parent class reference variable) to hold one parent class object.
- One to many associations is from parent to child where as from child to parent it is many to one association.
E.g.,
Person and phone number relationship is one to many relationships from person perspective because one person can have multiple phone numbers. The same is many to one relationship from phone number

perspective because multiple phone numbers can belong to a single person.

Note:

- ✓ While designing DB tables do not take the columns that holds outside business values as PK column because the values may change based on outside world business policies.
E.g., Taking mobile no, Voter id, Aadhaar No, Passport No and etc. columns as PK column is bad practice because these values may change of GOVT policies (business policies).
- ✓ Always take that column as PK column which gets values from underlying app or DB s/w dynamically.
E.g., Column that holds sequence generated values (Oracle), column that hold auto increment values (MySQL).

Directory Structure of BootDataJPAProj12-AssociationMapping-OneToMany:

```
▼ BootDataJPAProj12-AssociationMapping-OneToMany [boot]
  ▼ src/main/java
    ▼ com.sahu
      > BootDataJpaProj12AssociationMappingOneToManyApplication.java
    ▼ com.sahu.entity
      > Person.java
      > PhoneNumber.java
    ▼ com.sahu.repository
      > IPersonRepo.java
      > IPhoneNumberRepo.java
    ▼ com.sahu.runner
      > AssociationTestRunner.java
    ▼ com.sahu.service
      > IPersonMgmtService.java
      > PersonMgmtServiceImpl.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Project and External Dependencies
  > bin
  > gradle
  > src
  > build.gradle
  > gradlew
  > gradlew.bat
  > HELP.md
  > settings.gradle
```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.

- Previous project starter is enough and copy the application.properties file also.
- Place the following code within their respective files.

Person.java

```
package com.sahu.entity;

import java.io.Serializable;
import java.util.Set;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
import lombok.Setter;

@Setter
@Getter
@Entity
@Table(name="OTM_PERSON")
@AllArgsConstructor
@RequiredArgsConstructor
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer pId;
    @NonNull
    @Column(length = 20)
    private String pName;
    @NonNull
    @Column(length = 20)
```



```

    private String pAddress;

    @OneToMany(targetEntity = PhoneNumber.class, cascade =
CascadeType.ALL, mappedBy = "person", fetch = FetchType.EAGER)
    // @JoinColumn(name = "PERSON_ID", referencedColumnName =
"P_ID")
    private Set<PhoneNumber> contactDetails; // For One to Many

    public Person() {
        System.out.println("Person.Person()");
    }

    @Override
    public String toString() {
        return "Person [pId=" + pId + ", pName=" + pName + ",
pAddress=" + pAddress + "];"
    }
}

```

PhoneNumber.java

```

package com.sahu.entity;

import java.io.Serializable;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

```

```

import lombok.Setter;

@Setter
@Getter
@Entity
@Table(name="OTM_PHONENUMBER")
@AllArgsConstructor
@RequiredArgsConstructor
public class PhoneNumber implements Serializable {
    @Id
    @SequenceGenerator(name = "gen1", sequenceName = "regno_seq",
initialValue = 100, allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator
= "gen1")
    private Long regNo;
    @NotNull
    private Long phoneNo;
    @NotNull
    @Column(length = 20)
    private String provider;
    @NotNull
    @Column(length = 20)
    private String type;

    @ManyToOne(targetEntity = Person.class, cascade =
CascadeType.ALL)
    @JoinColumn(name = "PERSON_ID", referencedColumnName =
"pId")
    private Person person; //For Many to One

    public PhoneNumber() {
        System.out.println("PhoneNumber.PhoneNumber()");
    }

    @Override
    public String toString() {
        return "PhoneNumber [regNo=" + regNo + ", phoneNo=" +
phoneNo + ", provider=" + provider + ", type=" + type
+ "];"
    }
}

```

IPersonRepo.java

```
package com.sahu.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.Person;

public interface IPersonRepo extends JpaRepository<Person, Integer> {

}
```

IPhoneNumberRepo.java

```
package com.sahu.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.entity.PhoneNumber;

public interface IPhoneNumberRepo extends
JpaRepository<PhoneNumber, Long> {

}
```

IPersonMgmtService.java

```
package com.sahu.service;

import com.sahu.entity.Person;
import com.sahu.entity.PhoneNumber;

public interface IPersonMgmtService {

    public String savePerson(Person person);

    public String savePhoneNumbers(Iterable<PhoneNumber>
phoneNos);

}
```

PersonMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.Set;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.Person;
import com.sahu.entity.PhoneNumber;
import com.sahu.repository.IPersonRepo;
import com.sahu.repository.IPhoneNumberRepo;

@Service("personMgmtService")
public class PersonMgmtServiceImpl implements IPersonMgmtService {

    @Autowired
    private IPersonRepo personRepo;

    @Autowired
    private IPhoneNumberRepo phoneNumberRepo;

    @Override
    public String savePerson(Person person) {
        int idVal= personRepo.save(person).getId();
        return "Person and his phone numbers has saved with the id
value : "+idVal;
    }

    @Override
    public String savePhoneNumbers(Iterable<PhoneNumber>
phoneNos) {
        for (PhoneNumber phone : phoneNos) {
            phoneNumberRepo.save(phone);
        }
        return ((Set<PhoneNumber>) phoneNos).size()+" phone
numbers has saved.";
    }
}
```

AssociationTestRunner.java

```
package com.sahu.runner;

import java.util.Set;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.entity.Person;
import com.sahu.entity.PhoneNumber;
import com.sahu.service.IPersonMgmtService;

@Component
public class AssociationTestRunner implements CommandLineRunner {

    @Autowired
    private IPersonMgmtService personMgmtService;

    @Override
    public void run(String... args) throws Exception {
        //----- Save Operation child to Parent -----
        //Prepare object having associated child objects
        PhoneNumber ph1 = new PhoneNumber(99999999L, "Airtel",
"Office");
        PhoneNumber ph2 = new PhoneNumber(888888888L,
"Vodafone", "Residence");
        //Parent object
        Person person1 = new Person("Rajesh", "Hyd");
        //Parent to child
        ph1.setPerson(person1);
        ph2.setPerson(person1);
        //Child to parent
        Set<PhoneNumber> phoneSet = Set.of(ph1, ph2);
        person1.setContactDetails(phoneSet);
        //Invoke Service method
        System.out.println(personMgmtService.savePerson(person1));

        //----- Save Operation child to Parent -----
    }
}
```

```

        //Child Objects
        PhoneNumber phone1 = new PhoneNumber(75757463343L,
        "Jio", "Office");
        PhoneNumber phone2 = new PhoneNumber(6734635334L,
        "Vodafone", "Residence");
        //Parent object
        Person person = new Person("Karan", "Hyd");
        //Add Parent to Child
        phone1.setPerson(person);
        phone2.setPerson(person);
        //Add Child to Parent
        Set<PhoneNumber> phones = Set.of(phone1, phone2);
        person.setContactDetails(phones);

        System.out.println(personMgmtService.savePhoneNumbers(phones))
    ;
    }
}

```

Note:

- ✓ In Bi-directional associations instead of specifying foreign key column using @JoinColumn in both parent and child classes we can specify only at one side with the support of "mappedBy" param.
- ✓ The property of Associated class on which @JoinColumn is used to specify FK Column


```
@OneToMany(targetEntity = PhoneNumber.class, cascade = CascadeType.ALL, mappedBy = "person")
private Set<PhoneNumber> contactDetails; //For One to Many
```
- ✓ In One-to-Many association we need specify mappedBy at one Side (parent class).
- ✓ In Many to Many and One to One association we can specify mappedBy any side.
- ✓ In associating cascading means the non-select persistence operations performed on the main object will be propagated/ cascaded to the associated objects possible values are
 - cascade = CascadeType.ALL (Best)
 - cascade = CascadeType.DETACH
 - cascade = CascadeType.MERGE
 - cascade = CascadeType.PERSIST

- cascade = CascadeType.REFRESH
 - cascade = CascadeType.REMOVE
- ^{SF} ALL : CascadeType - CascadeType
^{SF} DETACH : CascadeType - CascadeType
^{SF} MERGE : CascadeType - CascadeType
^{SF} PERSIST : CascadeType - CascadeType
^{SF} REFRESH : CascadeType - CascadeType
^{SF} REMOVE : CascadeType - CascadeType

Select Operation in One-To-Many Association Mapping:

IPersonMgmtService.java

```

public Iterable<Person> fetchByPerson();
public Iterable<PhoneNumber> fetchByPhoneNumber();
  
```

PersonMgmtServiceImpl.java

```

@Override
public Iterable<Person> fetchByPerson() {
    return personRepo.findAll();
}

@Override
public Iterable<PhoneNumber> fetchByPhoneNumber() {
    return phoneNumberRepo.findAll();
}
  
```

AssociationTestRunner.java

```

@Override
public void run(String... args) throws Exception {
    //-----Load operation parent to child -----
    personMgmtService.fetchByPerson().forEach(person->{
        System.out.println("Parent : "+person);
        person.getContactDetails().forEach(System.out::println);
    });
    // ----- Load operation Child to parent
    personMgmtService.fetchByPhoneNumber().forEach(phone-> {
        System.out.println("Child : "+phone);
        System.out.println("Parent : "+phone.getPerson());
    });
}
  
```

Note:

- ✓ In association mapping the cascading type are related non-select persistence operations which indicates any non-select persistence operation on main object will be cascaded to the associated child objects.

CascadeType.ALL/ DETACH/ MERGE/ PERSIST/ REFRESH/ REMOVE

- ✓ In association mapping the fetch type are related select persistence operations which indicates the associated child objects should be loaded along with parent objects or not.

FetchType.EAGER:

- Child objects will be loaded along with Parent objects.

FetchType.LAZY:

- Parent objects will be load normally but the associated child objects will be loaded lazily on demand basis.
- ✓ One to Many, One to One, Many to Many Associations the default fetch type is "LAZY".
- ✓ Many To One Association the default fetch type is "EAGER".

Delete operation in Association Mapping:

- In One-to-Many association if we delete parent object then it not only deletes parent table record but it also deletes the associated records in child table because of CascadeType.ALL.

IPersonMgmtService.java

```
public String deleteByPerson(Integer personId);
```

PersonMgmtServiceImpl.java

```
@Override
public String deleteByPerson(Integer personId) {
    //Load parent object
    Optional<Person> optPerson = personRepo.findById(personId);
    if(optPerson.isPresent()) {
        personRepo.delete(optPerson.get());
        return "Person add his Phone Numbers has deleted";
    }
    else {
        return "Person not found";
    }
}
```


AssociationTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    System.out.println(personMgmtService.deleteByPerson(24));
}
```

Deleting only Childs of parent In One-to-Many Association:

- Here Load parent object, get associated child objects of a parent object, nullify Parent object from child objects then perform delete operation on child objects.

IPersonMgmtService.java

```
public String deleteAllPhoneNumbersOfAPerson(Integer personId);
```

PersonMgmtServiceImpl.java

```
@Override
public String deleteAllPhoneNumbersOfAPerson(Integer personId) {
    //Load Parent object
    Optional<Person> optPerson = personRepo.findById(personId);
    if(optPerson.isPresent()) {
        //Get all child of a Parent
        Set<PhoneNumber> phoneNos =
optPerson.get().getContactDetails();
        //Delete all child
        phoneNos.forEach(phone -> phone.setPerson(null));
        phoneNumberRepo.deleteAllInBatch(phoneNos);
        return phoneNos.size()+" Phone number of "+personId+"
person has deleted";
    }
    return personId+" person not found";
}
```

AssociationTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    System.out.println(personMgmtService.deleteAllPhoneNumbersOfAPerson(25));
}
```

Joins in Spring Data JPA using HQL/ JPQL

- Joins are given to get data from two DB tables of association having some implicit conditions.
- We can also add new conditions on the top of implicit conditions.
- In SQL to work with joins the two DB tables need not be in relationship where as in HQL/ JPQL the two DB tables needs to be in association to apply joins.

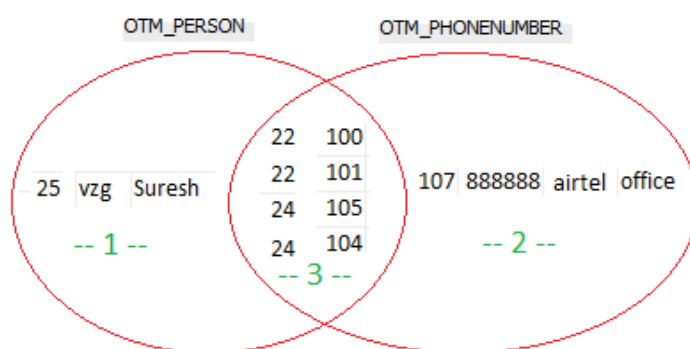
HQL/ JPQL supports 4 types of Joins:

- Inner Join
- Right Join/ Right Outer Join
- Left Join/ Left Outer Join
- Full Join/ Full Outer Join

OTM_PERSON		
P_ID	P_ADDRESS	P_NAME
1	22 Hyd	Rajesh
2	24 Hyd	Karan
3	25 vzg	Suresh

OTM_PHONENUMBER					
REG_NO	PHONE...	PROVIDER	TYPE	PERSON_ID	
1	100	8888888888	Vodafone	Residence	22
2	101	9999999	Airtel	Office	22
3	105	75757463343	Jio	Office	24
4	104	6734635334	Vodafone	Residence	24
5	107	888888	airtel	office	(null)

Venn Diagram



- Inner join gives common data of both DB tables (left side and right-side DB tables). i.e., 3rd area content.
- Right join/ Right outer join gives common data of both DB tables (left side and right-side DB tables) and also gives uncommon data of right-side DB table. i.e., 3rd + 2nd area content.
- Left join/ Left outer join gives common data of both DB tables (left side and right-side DB tables) and also gives uncommon data of left side DB table. i.e., 1st + 3rd area content

- Full join/ Full outer join gives common and uncommon data of both DB tables. i.e., 1st + 2nd + 3rd area content.

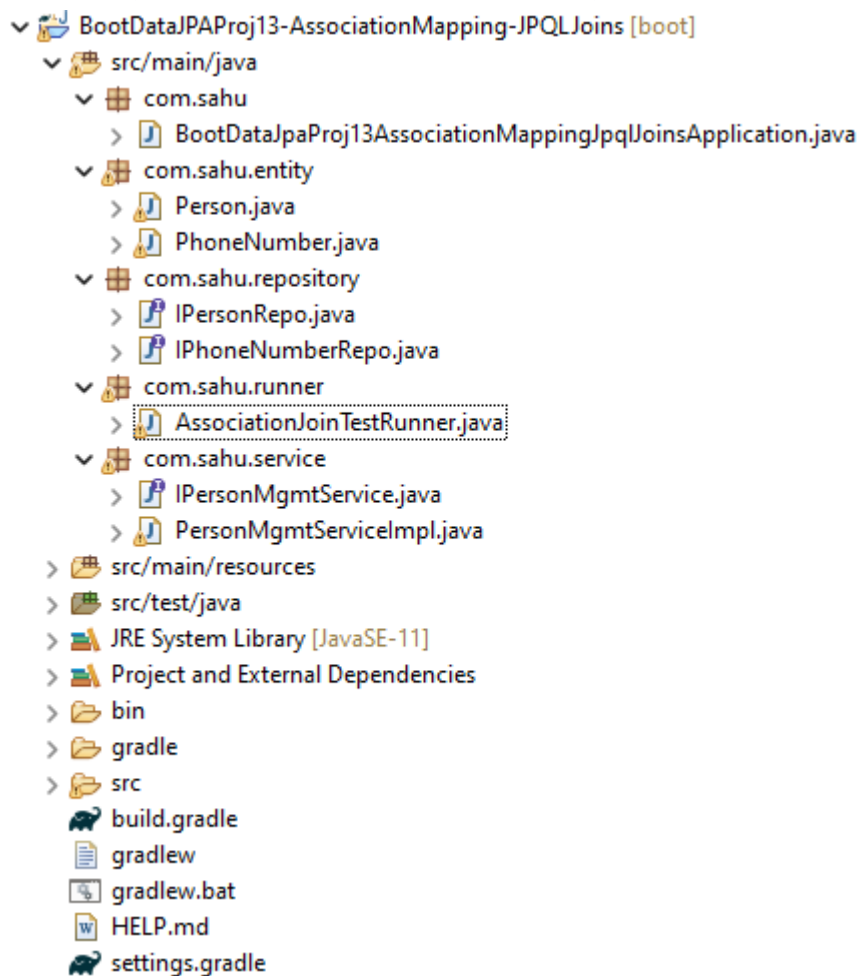
Note: To work with HQL/ JPQL Joins we need keep DB tables and their entity classes in relationship using Association Mapping concepts.

HQL/ JPQL Query syntax parent to child:

select <parent class properties>, <child class properties> from <Parent class> <alias name> <join type> <parent class HAS-A property> <alias name>

- <Parent class>: Parent table/ Left side table
- <parent class HAS-A property>: child table/ right side table
- <join type>: Inner join, Right join, Left join, Full join

Directory Structure of BootDataJPAProj13-AssociationMapping-JPQLJoins:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.

- Previous project starter is enough and copy the application.properties file and entity classes Person.java and PhoneNumber.java from the previous project also the IPhoneNumberRepo.java repository interface.
- Place the following code within their respective files.

IPersonRepo.java

```
package com.sahu.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.sahu.entity.Person;

public interface IPersonRepo extends JpaRepository<Person, Integer> {
    //@Query("SELECT p.pId, p.pName, p.pAddress, ph.regNo,
    ph.phoneNo, ph.provider, ph.type FROM Person p INNER JOIN
    p.contactDetails ph")
    //@Query("SELECT p.pId, p.pName, p.pAddress, ph.regNo,
    ph.phoneNo, ph.provider, ph.type FROM Person p RIGHT JOIN
    p.contactDetails ph")
    //@Query("SELECT p.pId, p.pName, p.pAddress, ph.regNo,
    ph.phoneNo, ph.provider, ph.type FROM Person p LEFT JOIN
    p.contactDetails ph")
    @Query("SELECT p.pId, p.pName, p.pAddress, ph.regNo,
    ph.phoneNo, ph.provider, ph.type FROM Person p FULL JOIN
    p.contactDetails ph")
    public List<Object[]> fetchDataUsingJoinsByParent();
}
```

IPersonMgmtService.java

```
package com.sahu.service;

import java.util.List;

public interface IPersonMgmtService {
    public List<Object[]> fetchDataByJoinUsingParent();
}
```

PersonMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.repository.IPersonRepo;
import com.sahu.repository.IPhoneNumberRepo;

@Service("personMgmtService")
public class PersonMgmtServiceImpl implements IPersonMgmtService {

    @Autowired
    private IPersonRepo personRepo;

    @Autowired
    private IPhoneNumberRepo phoneNumberRepo;

    @Override
    public List<Object[]> fetchDataByJoinUsingParent() {
        return personRepo.fetchDataUsingJoinsByParent();
    }
}
```

AssociationJoinTestRunner.java

```
package com.sahu.runner;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.IPersonMgmtService;

@Component
public class AssociationJoinTestRunner implements CommandLineRunner {

    @Autowired
    private IPersonMgmtService personMgmtService;
```

```

@Autowired
private IPersonMgmtService personMgmtService;

@Override
public void run(String... args) throws Exception {

    personMgmtService.fetchDataByJoinUsingParent().forEach(row->{
        for (Object val : row) {
            System.out.print(val+" ");
        }
        System.out.println();
    });
}
}

```

Spring Data MongoDB

- Spring Data JPA is given to interact with SQL DB s/w and Spring Data provides separate sub module to interact with each NoSQL DB s/w
- Spring Data provides unified model to work with both SQL and NoSQL DB softwares.
- SQL DB softwares are Oracle, PostgreSQL, MySQL and etc. (DB tables based).
- NoSQL DB software are MongoDB, Cassandra, Couchbase, Neo4j and etc. (key-value based, document based, graph based and etc.).
- If data having fixed format and schema the go for SQL DB s/w.
- If data is dynamically growing and no fixed format then go for NoSQL DB s/w.

Difference between SQL and NoSQL

Parameter	SQL	NOSQL
Definition	SQL databases are primarily called RDBMS or Relational databases.	NoSQL databases are primarily called as non-relational or distributed database.
Design for	Traditional RDBMS uses SQL syntax and queries to analyze and get the data for further insights.	NoSQL database system consists of various kind of database technologies. These

	They are used for OLAP systems.	databases were developed in response to the demands presented for the development of the modern application.
Query Language	Structured query language (SQL).	No declarative query language.
Type	SQL databases are table-based databases.	NoSQL databases can be document based, key-value pairs, graph databases.
Schema	SQL databases have a predefined schema.	NoSQL databases use dynamic schema for unstructured data.
Ability to scale	SQL databases are vertically scalable.	NoSQL databases are horizontally scalable.
Examples	Oracle, Postgres, and MS-SQL.	MongoDB, Redis, Neo4j, Cassandra, Hbase.
Best suited for	An ideal choice for the complex query intensive environment.	It is not good fit complex queries.
Hierarchical data storage	SQL databases are not suitable for hierarchical data storage.	More suitable for the hierarchical data store as it supports key-value pair method.
Variations	One type with minor variations.	Many different types which include key-value stores, document databases, and graph databases.
Development Year	It was developed in the 1970s to deal with issues with flat file storage.	Developed in the late 2000s to overcome issues and limitations of SQL databases.
Open-source	A mix of open-source like Postgres & MySQL, and commercial like Oracle Database.	Open-source

Consistency	It should be configured for strong consistency.	It depends on DBMS as some offers strong consistency like MongoDB, whereas others offer only offers eventual consistency, like Cassandra.
Best Used for	RDBMS database is the right option for solving ACID problems.	NoSQL is a best used for solving data availability problems.
Importance	It should be used when data validity is super important.	Use when it's more important to have fast data than correct data.
Best option	When you need to support dynamic queries.	Use when you need to scale based on changing requirements.
Hardware	Specialized DB hardware (Oracle Exadata, etc.).	Commodity hardware.
Network	Highly available network (InfiniBand, Fabric Path, etc.).	Commodity network (Ethernet, etc.).
Storage Type	Highly Available Storage (SAN, RAID, etc.).	Commodity drives storage (standard HDDs, JBOD).
Best features	Cross-platform support, Secure and free.	Easy to use, High performance, and Flexible tool.
Top Companies Using	Hootsuite, CircleCI, Gauges.	Airbnb, Uber, Kickstarter.
ACID vs. BASE Model	ACID (Atomicity, Consistency, Isolation, and Durability) is a standard for RDBMS	BASE (Basically Available, Soft state, Eventually Consistent) is a model of many NoSQL systems.

MongoDB

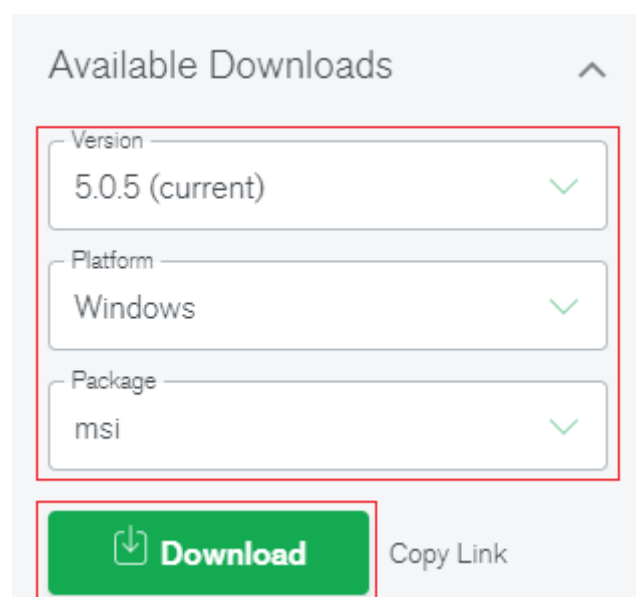
- It is Document based NoSQL DB s/w.
- It internally maintains the data in the form of JSON (Java Script Object Notation) document

Examples of JSON Data:

```
{
  "api": {
    "title": "Example API",
    "links": {
      "author": "mailto:api-admin@example.com",
      "describedBy": "https://example.com/api-docs/"
    }
  },
  "resources": {
    "tag:me@example.com,2016:widgets": {
      "href": "/widgets/"
    },
    "tag:me@example.com,2016:widget": {
      "hrefTemplate": "/widgets/{widget_id}",
      "hrefVars": {
        "widget_id": "https://example.org/param/widget"
      },
      "hints": {
        "allow": ["GET", "PUT", "DELETE", "PATCH"],
        "formats": {
          "application/json": {}
        },
        "acceptPatch": ["application/json-patch+json"],
        "acceptRanges": ["bytes"]
      }
    }
  }
}
```

MongoDB Software Installation

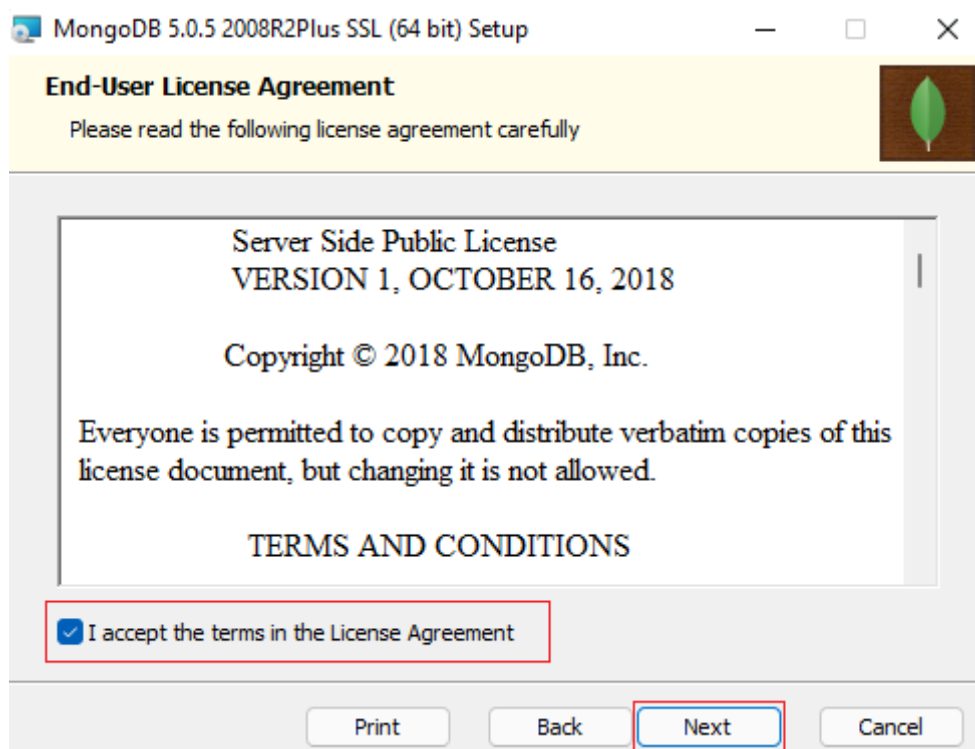
Step 1: Download the MongoDB software [\[Download\]](#)



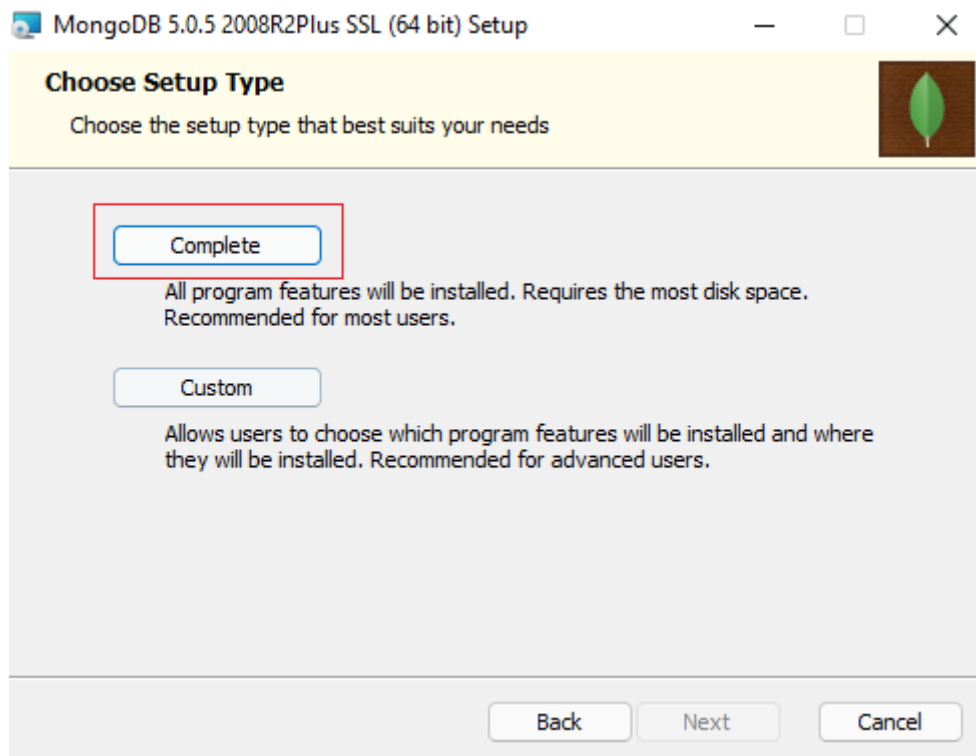
Step 2: After downloaded double click on the msi file then you will get the MongoDB Setup, there click on Next.



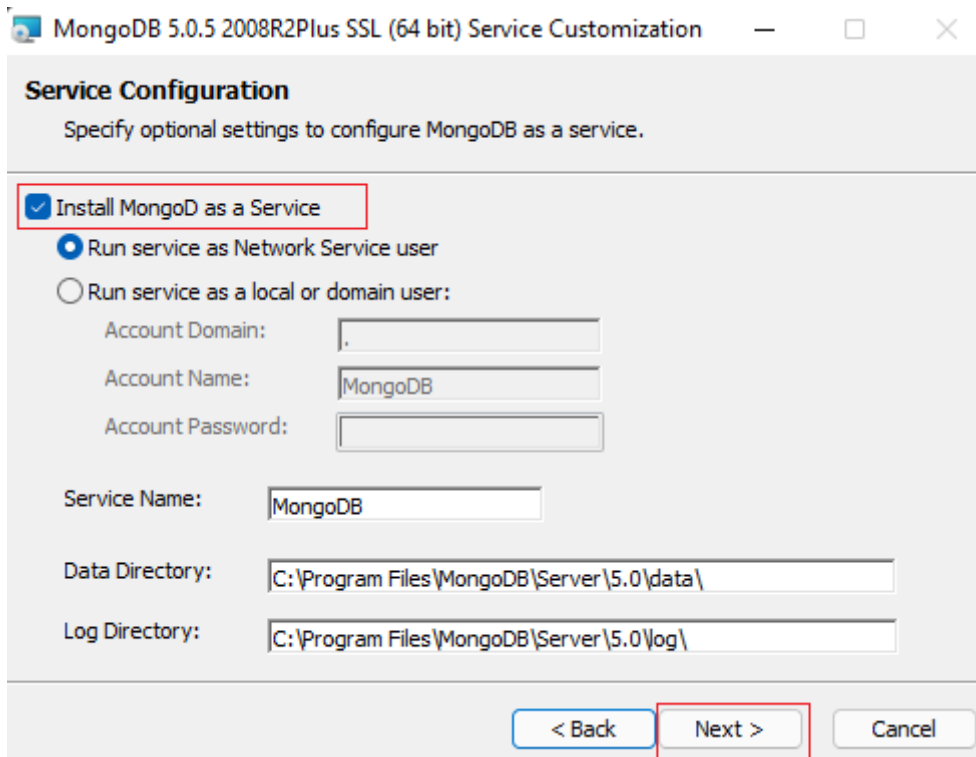
Step 3: Choose “I accept the terms in the License Agreement” then click on Next.



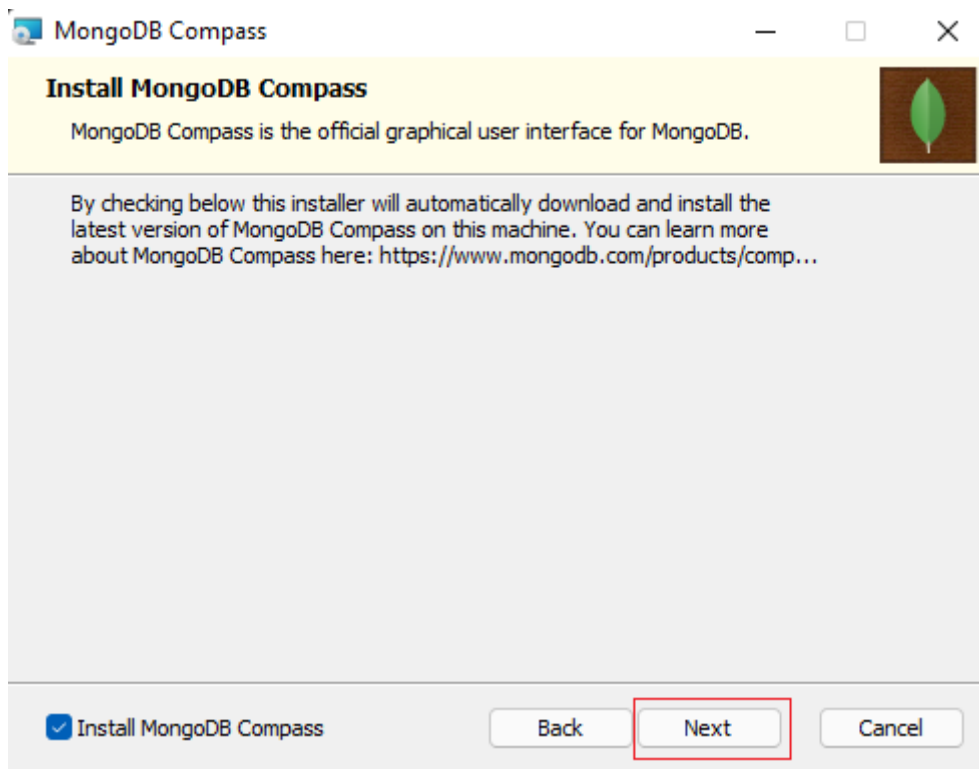
Step 4: Then click on the Complete Button for installing all program features and this is recommended to choose.



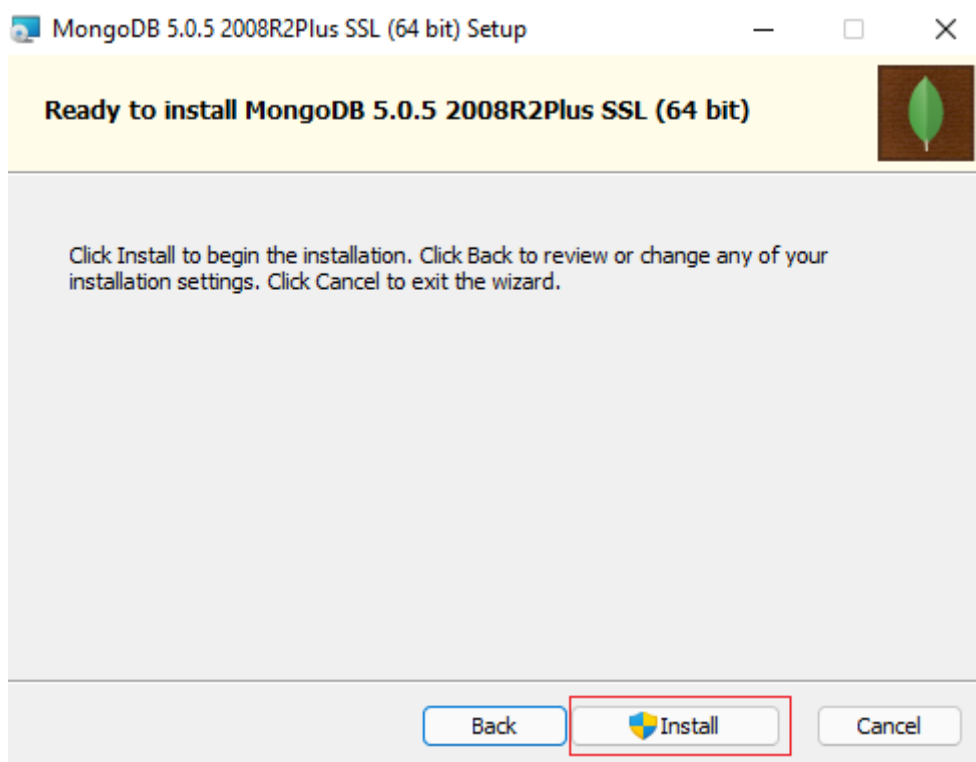
Step 5: Choose Install MongoDB as a Service user which is default selection the click on Next.



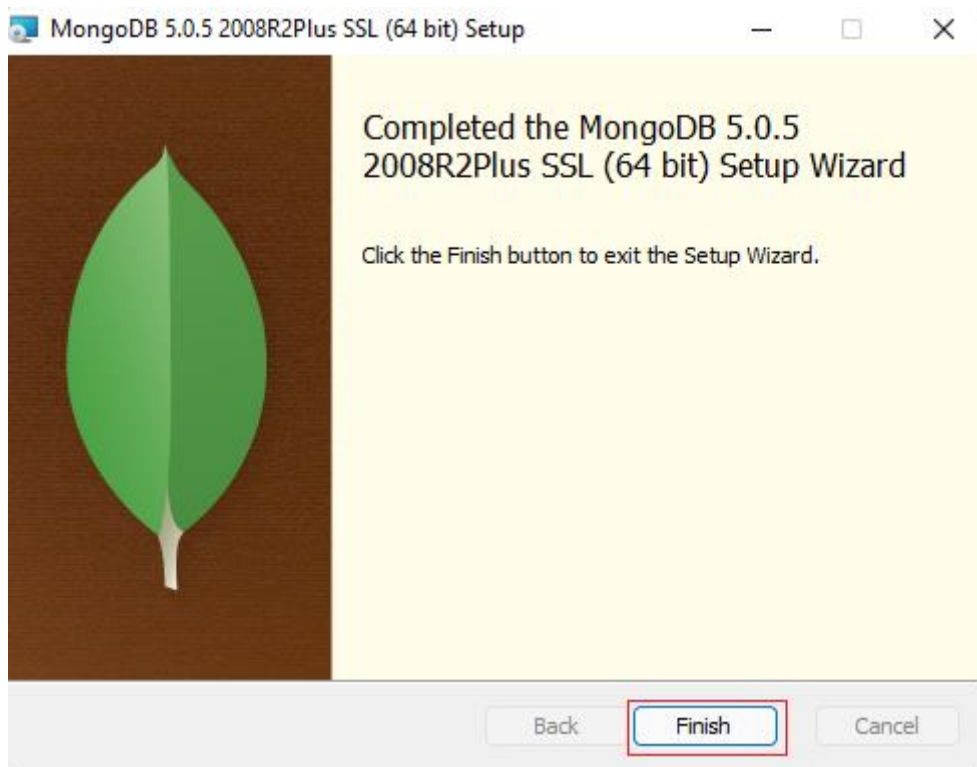
Step 6: Then again click on Next.



Step 7: Then click on Install.



Step 8: After the installation process (it may take little time) complete click on the Finish button.



Note: We have multiple MongoDB GUI tools or client's s/w like Compass, Robot3T and etc.

To start MongoDB software:

Go to C:\Program Files\MongoDB\Server\5.0\bin and use mango.exe file.

SQL terminologies:

Physical DB s/w (like Oracle)

|----> Logical DB1 (SID)

|----> DB table1 (columns and rows)

|----> DB table2 (columns and rows)

|----> DB table3 (columns and rows)

|----> Logical DB2 (SID)

DB tables and Columns maintain records/ rows.

NoSQL MongoDB terminologies:

Physical Db s/w (like MongoDB)

|----> Logical DB1

|----> Collection1 (document1, document2, (JSON docs)

|----> Collection2 (document1, document2, (JSON docs)

|----> Logical DB2

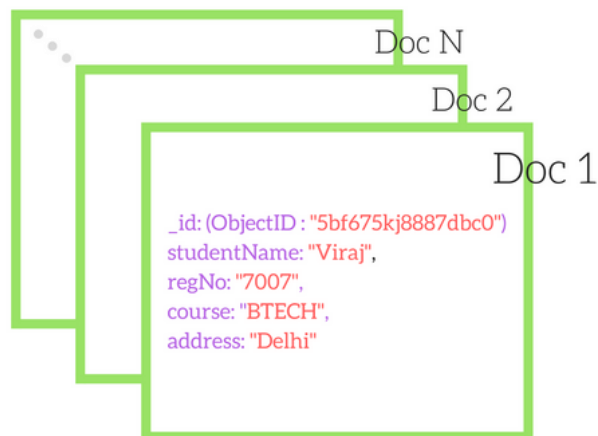
Collections maintain document (equal to records)

Note: When we insert document to collection the MongoDB dynamically generates unique ID as Hexadecimal number.

- Binary (Base 2): 0-1
- Decimal (Base 10): 0-9
- Octal (Base 8) 0-7
- Hexadecimal (Base 16): 0-9, a-f

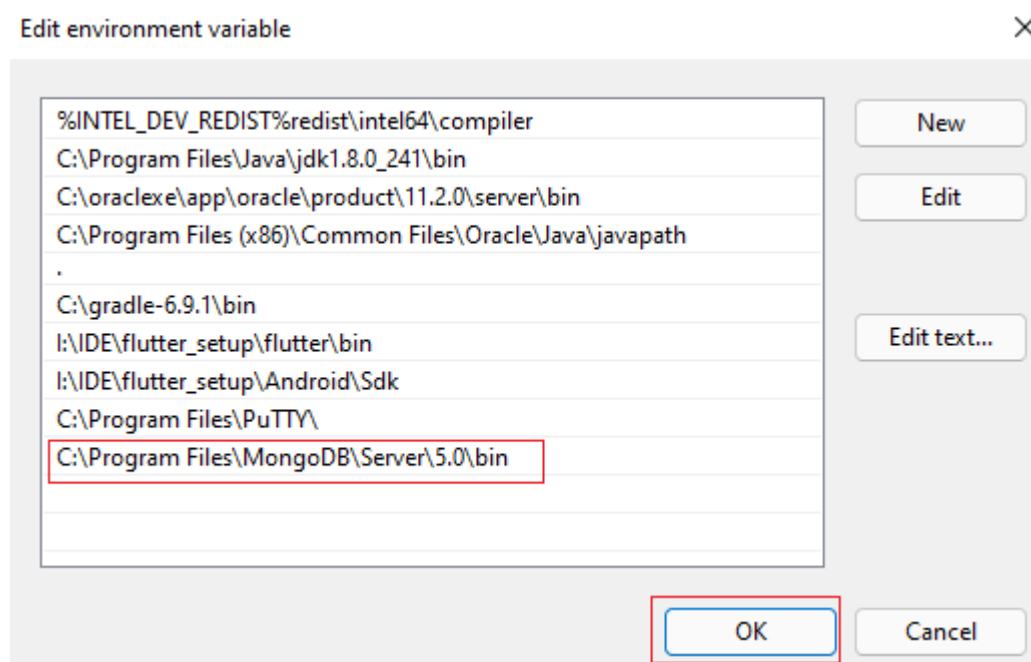
Collection

A collection can have multiple documents



MongoDB Shell commands

1. Add <MongoDB home>\bin folder to PATH environment variable
This PC --> Properties --> Advanced system settings --> Environment Variables... --> System Variable
Name: Path
Value: C:\Program Files\MongoDB\Server\5.0\bin



Then click on OK, OK and again OK.

2. Open MongoDB Shell using mongo command from any command prompt.

```
CMD> mongo
```

3. To list all Logical DBs

```
CMD> show dbs
```

```
> show dbs
admin    0.000GB
config  0.000GB
local    0.000GB
```

4. To create new Logical DB

```
CMD> use ntsp714db
```

```
> use ntsp714db
switched to db ntsp714db
```

Unless and until we add collections to Logical DB, the newly created Logical DB will not appear in the list of Logical DBs.

5. To know current Logical DB

```
CMD> db
```

```
> db
ntsp714db
```

6. To create collection with one document.

```
CMD> db.customer.insertOne({cno:1001, cname:"Raja",
                           cadd:"Hyd", billAmt:89000.0})
```

```
> db.customer.insertOne({cno:1001, cname:"Raja", caddress:"Hyd",
billAmt:8900.0})
{
  "acknowledged" : true,
  "insertedId" : ObjectId("61c96bb99eb42874344f354b")
}
```

7. To list out documents of the collection

```
CMD> db.customer.find()
```

```
CMD> db.customer.find().pretty()
```

```
> db.customer.find()
{ "_id" : ObjectId("61c96bb99eb42874344f354b"), "cno" : 1001,
  "cname" : "Raja", "caddress" : "Hyd", "billAmt" : 8900 }
> db.customer.find().pretty()
{
  "_id" : ObjectId("61c96bb99eb42874344f354b"),
  "cno" : 1001,
  "cname" : "Raja",
  "caddress" : "Hyd",
  "billAmt" : 8900
}
```

8. To insert many documents at once to a Collection

```
CMD> db.customer.insertMany([{cno:1002, cname:"Ramesh"},  
                             {cno:1003, cname:"Rakesh"}])
```

```
> db.customer.insertMany([{cno:1002, cname:"Ramesh"}, {cno:1003, cname:"Rakesh"}])  
{  
  "acknowledged" : true,  
  "insertedIds" : [  
    ObjectId("61c96e989eb42874344f354c"),  
    ObjectId("61c96e989eb42874344f354d")  
  ]  
}
```

9. To insert document to a collection with array values.

```
CMD> db.customer.insertOne({cno:1003, cname:"rajesh",  
                           mobileNo:[987671245,6788886777]})
```

```
> db.customer.insertOne({cno:1003, cname:"rajesh",  
... mobileNo:[987671245,6788886777]})  
{  
  "acknowledged" : true,  
  "insertedId" : ObjectId("61c96fbc9eb42874344f3551")  
}
```

10.To find docs of a collection with condition

```
CMD> db.customer.find({cname:"Raja"}).pretty()
```

```
> db.customer.find({cname:"Raja"}).pretty()  
{  
  "_id" : ObjectId("61c96bb99eb42874344f354b"),  
  "cno" : 1001,  
  "cname" : "Raja",  
  "address" : "Hyd",  
  "billAmt" : 8900  
}  
> db.customer.find({address:"Hyd",cname:"Raja"}).pretty()  
{  
  "_id" : ObjectId("61c96bb99eb42874344f354b"),  
  "cno" : 1001,  
  "cname" : "Raja",  
  "address" : "Hyd",  
  "billAmt" : 8900  
}
```

11.To delete MongoDB document

```
CMD> db.customer.remove({cno:1003})
```

```
CMD> db.customer.deleteOne({cno:1003})
```

```
CMD> db.customer.Many({cno:1003})
```

```
> db.customer.remove({cno:1003})  
WriteResult({ "nRemoved" : 5 })
```

12.To list all collections

```
CMD> show collections
```

```
> show collections  
customer
```


13.To remove collection

```
CMD> db.customer.drop()  
> db.customer.drop()  
true
```

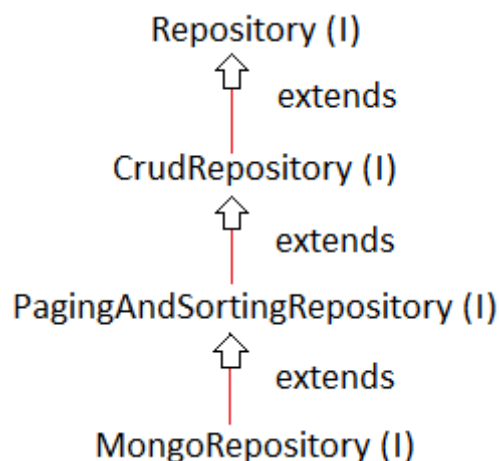
14.To switch to other Logical DB

```
CMD> use ntsp714db  
> use ntsp714db  
switched to db ntsp714db
```

Note: _id key holds dynamically generated unique value as PK value.

Working MongoRepository

- ✓ Spring Data provides two approaches to perform CRUD Operations and other operations on MongoDB
 - Using MongoRepository (more recommended)
 - Using MongoTemplate



Spring Boot allows to work with

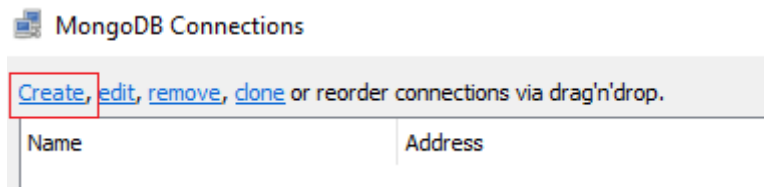
- a. Embedded InMemory MongoDB s/w (only testing)
- b. External MongoDB s/w (Recommended)

Different GUI Tools to work with MongoDB

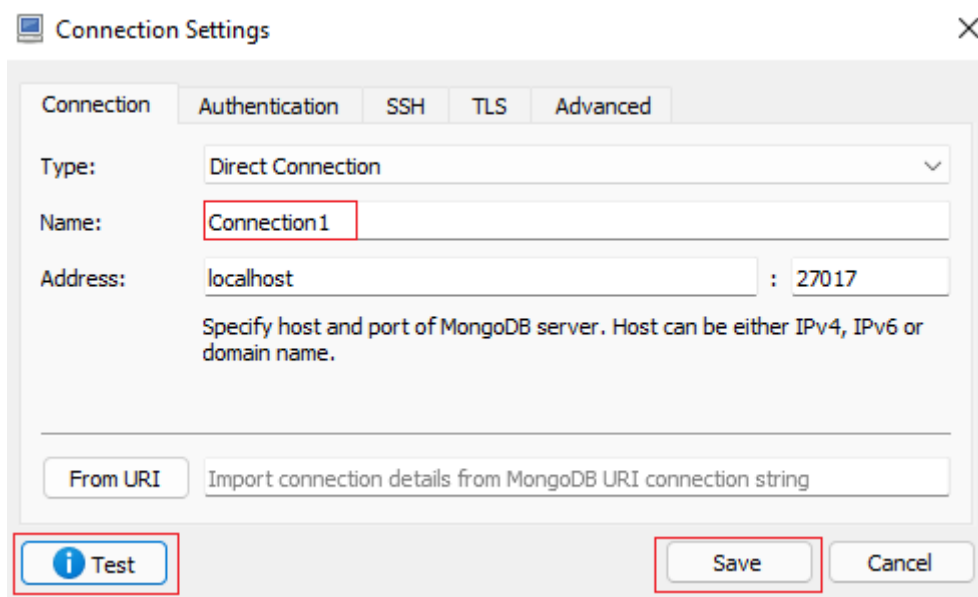
- These are the following GUI tools that we can use for working with MongoDB software
 - a. Compass (Bit heavy s/w) - comes with MongoDB installation
 - b. Rob03T (Light weight)
Now a day this is use a huge.
To download the software [\[Click Here\]](#)
Download and install the software, normal installation process.

Procedure to create Logical DB with collection in MongoDB using Robo3T

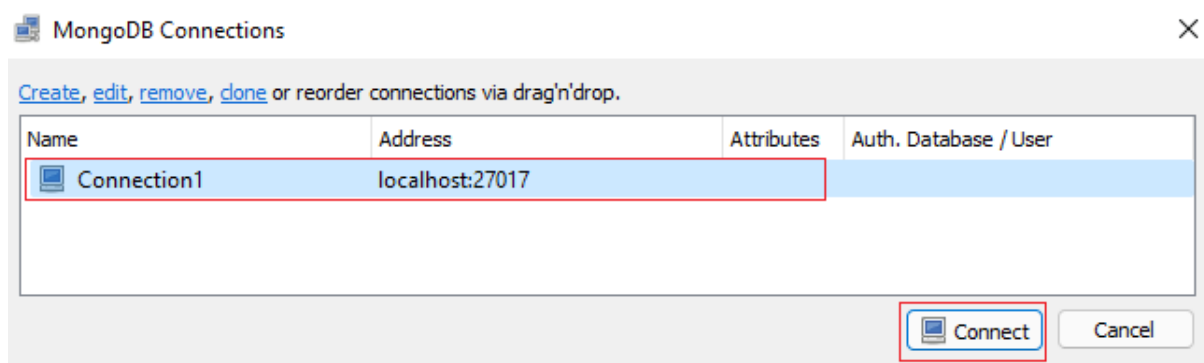
Step 1: Launch Robo3T and click on create option to create a new Connection



Step 2: Then give the connection name and click on Test, you will get a success message the click on Save. Now your connection is ready.

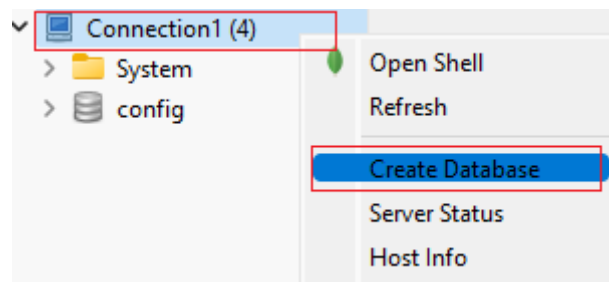


Step 3: Now Choose your connection and click on Connect.

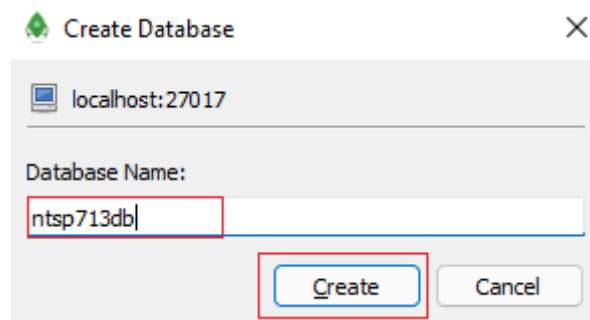


Step 4: To Create Logical DB

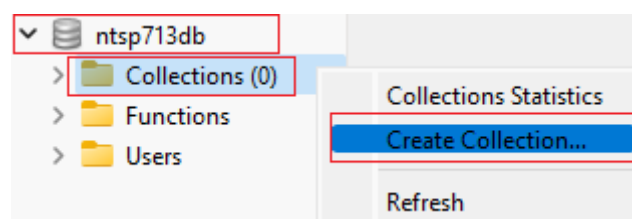
Right click on the connection then click on Create Database.



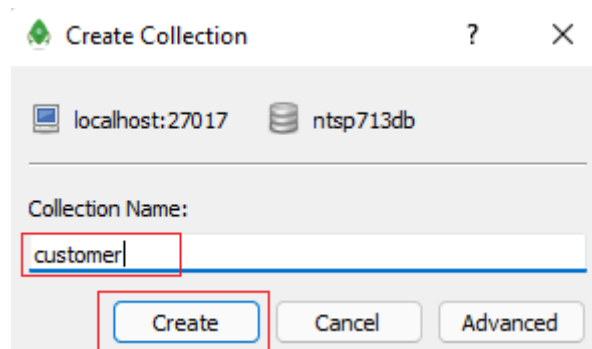
Step 5: Now give the Database Name then click on Create.



Step 6: To create Collection, expand your Database then right click on Collections then click on Create Collection...

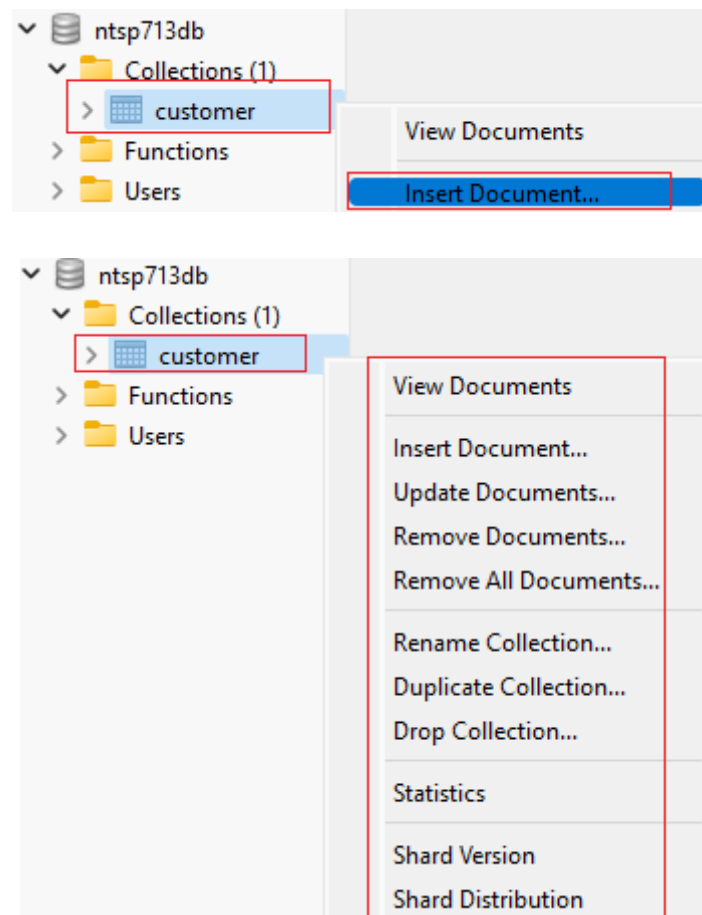


Step 7: Give a Collection Name then click on Create.

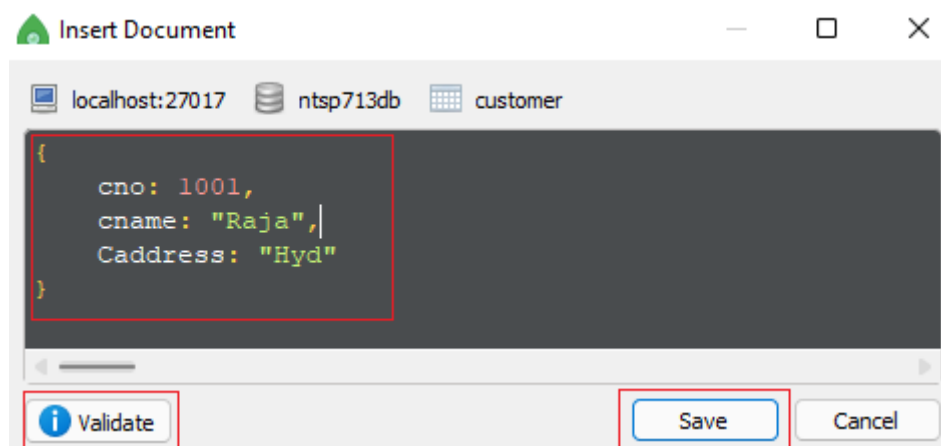


Step 8: To inset document right click on Your Collection (customer) then click on Insert Document...

Note: When you right click on your collection you will get all the listed operations below.

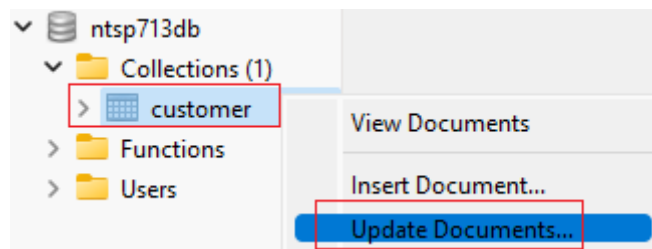


Step 9: Fill the required information then click on Validate you will get a Successful validation message then click on Save.

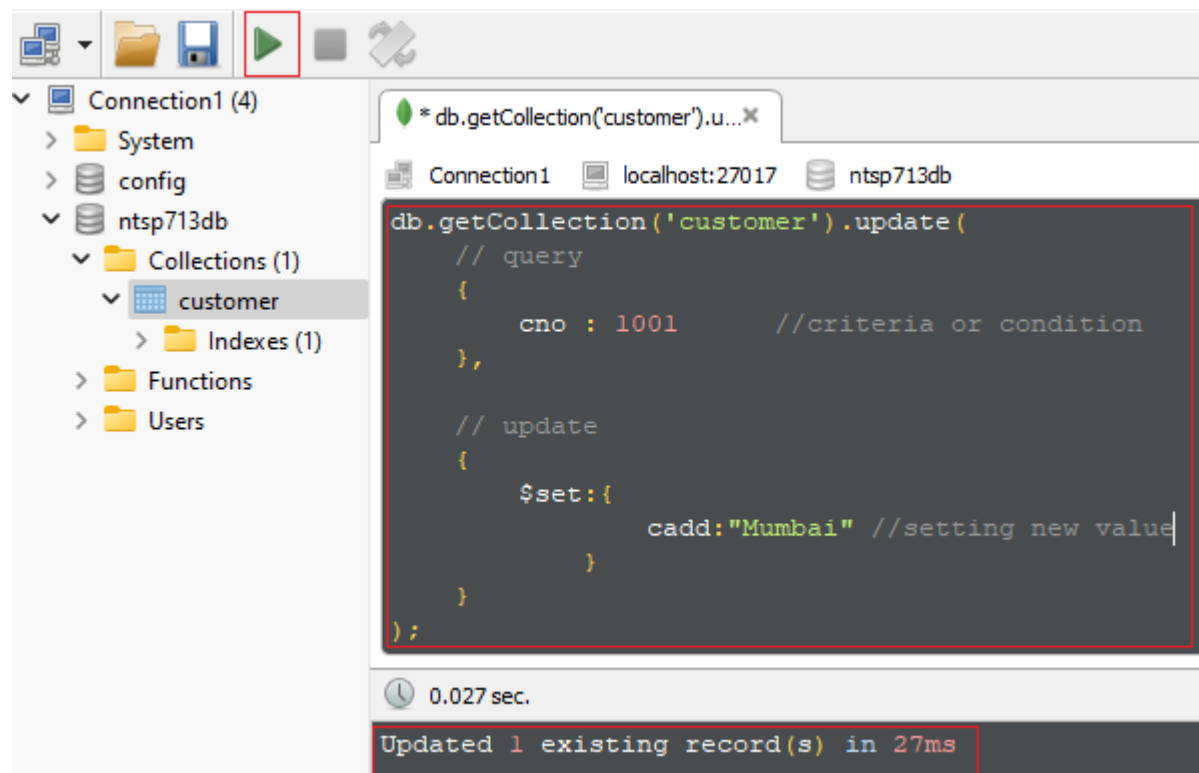


Step 10: To update the document

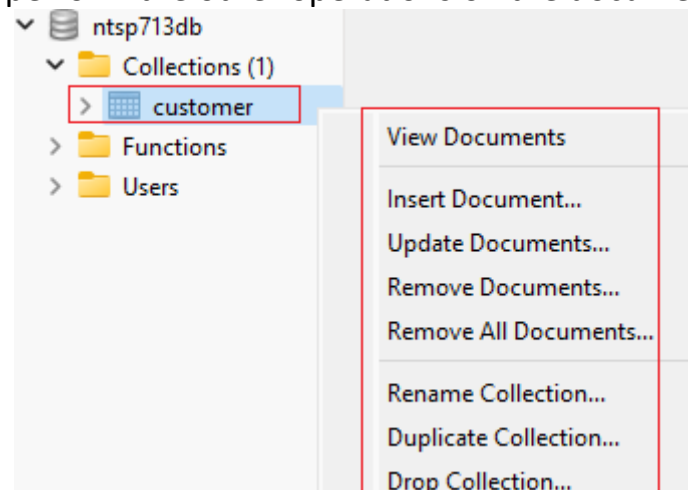
- View the document edit document and modify attribute values (or)
- Right click on customer (document) then click on Update Document...



Step 11: Now put the condition on query section and update or new values in update section using \$set: { } after that click on the Execute button or click on CTRL + enter then you will get a success message as like below .

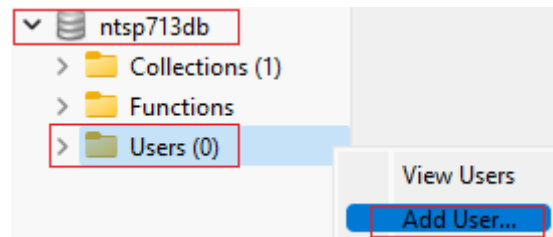


Like this we can perform the other operations on the document and collection.

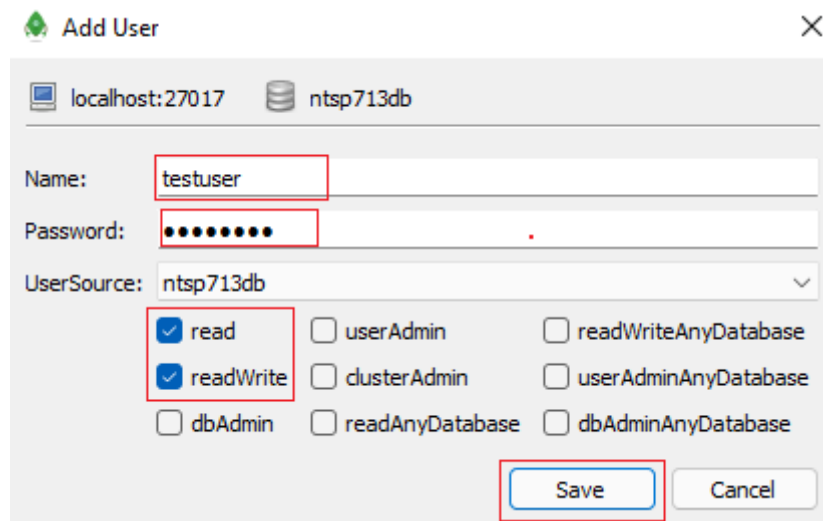


Add Username, Password to Logical DB for Authentication using Robo3T:

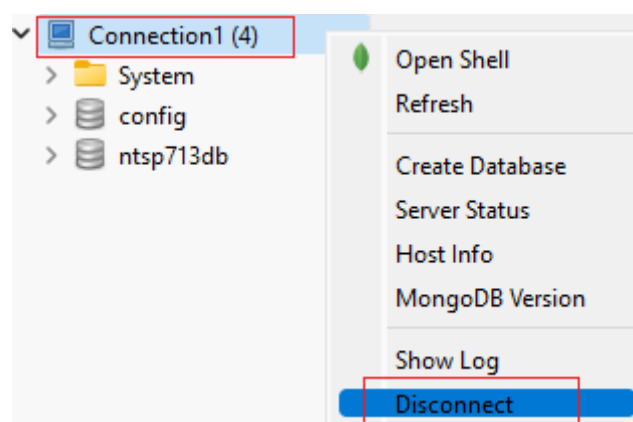
Step 1: To add user expand your Logical DB the right click on your Users then click on Add User...



Step 2: Give the Name, Password then choose require permissions then click on Save.

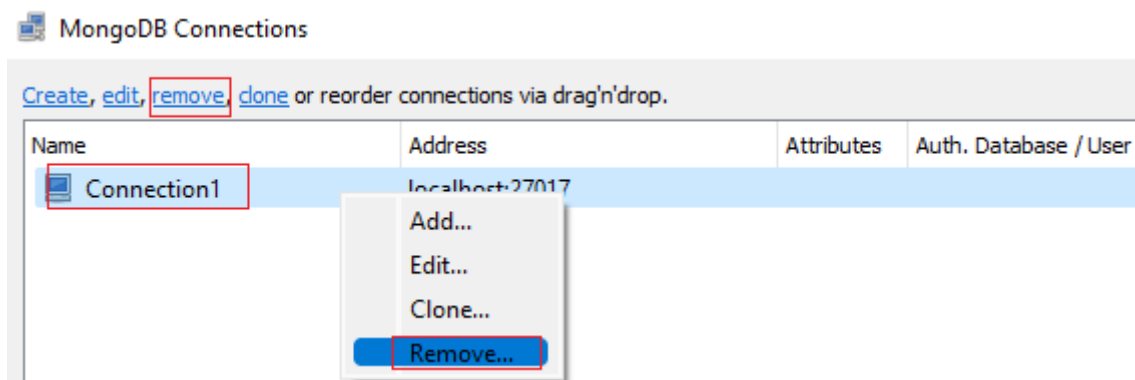


Step 3: Now we have to disconnect for that right click on the Connection the click on Disconnect

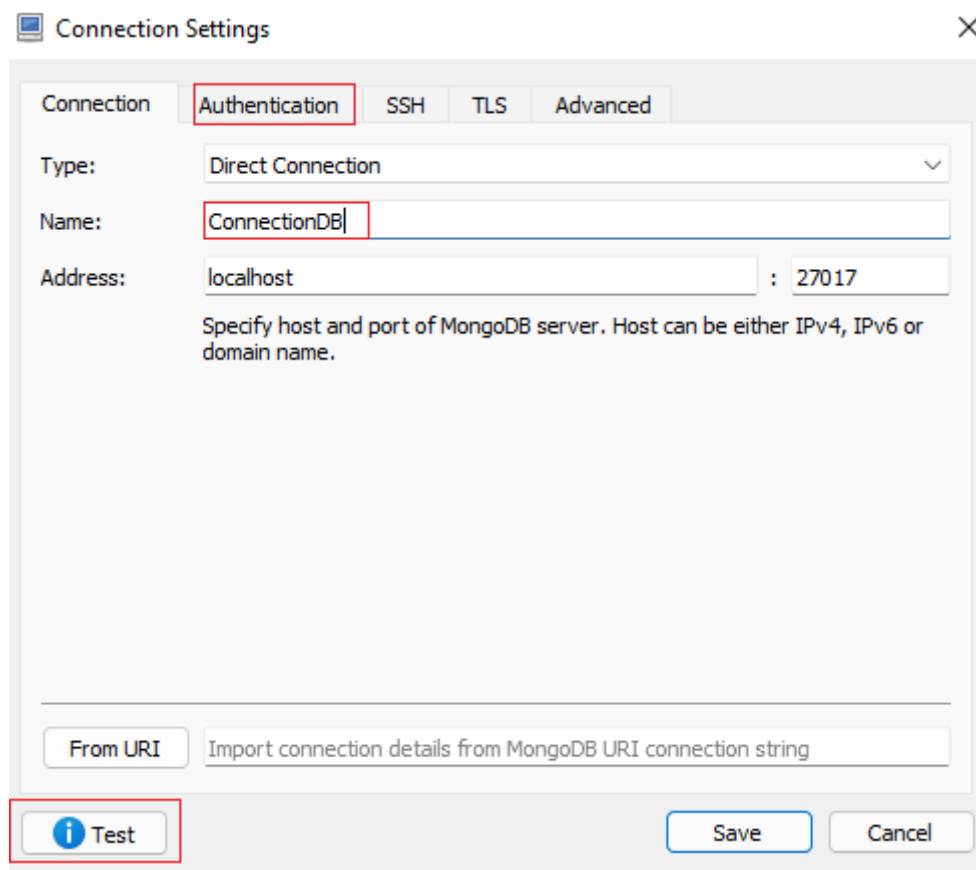


Step 4: Now remove the existing connection for that choose the connection the simply click on remove otherwise right click on the select connection the

click on Remove...



Step 5: Now click on the Create option then give a Name and click on Test for checking after getting success message click on Authentication.



Step 5: Now check the Perform authentication option then give the existing Database name the give the User Name and Password that we create previously then click on Test to check is everything is ok on not then click on Save. After that just click on Connect you will connect successfully.

Note: The Local DB and the user must available previously other we will get authentication error.

Connection Settings

Connection Authentication SSH TLS Advanced

☒ Perform authentication

Database:

The admin database is unique in MongoDB. Users with normal access to the admin database have read and write access to **all databases**.

User Name:

Password:

Auth Mechanism:

☐ Manually specify visible databases

Developing Document class whose object represent documents of a Collection in MongoDB

Collection:

customer		
--> cno		Document 1 attributes
--> cname		
--> cadd		

customer		
--> cno		Document 2 attributes
--> cname		
--> cadd		
--> billAmt		

Note:

- ✓ If we want to represent the documents of a collection using the objects of Java class then that class must be designed having highest possible properties representing the attributes of the Document.
- ✓ The java class whose objects represent documents of Collection is called Document class (@Document class).

Version 1:

@Document

@Data

```
public class Customer {  
    @Id  
    private String id;  
    private Integer cno;  
    private String cname;  
    private String cadd;  
    private Double billAmt;  
}
```

- If you want to use MongoDB generated id value as the id value of Document object then take String property having @Id as the id property as shown above.
- If you are not interested to use MongoDB generated id value as the id value of the Document then we can take support one or another generator support to generate the id value (Like UUIDGenerator class).

Version 2:

@Document

@Data

```
public class Customer {  
    @Id  
    private Integer cno;  
    private String cname;  
    private String cadd;  
    private Double billAmt;  
}
```

- If you feel that you can store unique value in "cno" property then make "cno" property as the id property otherwise follow Version 1.

Note:

- ✓ Like Hibernate/ JPA we do not have readymade generators configuration for ID property in MongoDB document class.
- ✓ In MongoDB officially there is no PK, FK constraints but we can bring that effect indirectly.
- ✓ PK constraint using _id attribute.
- ✓ FK constraint using documents nesting or chaining.

- ✓ The Java Document object representing the MongoDB document of a collection can have dynamically generated string value as the id value
 - or custom String value as the id value (given by ID Generator). (Version 1)
 - or our choice property value as the id value. (Version2)

Procedure to Develop First Spring Data MongoDB Application using MongoRepository

Step 1: Create Spring Boot starter project and select the following starters,

- Spring Data MongoDB
- Lombok

Step 2: Add the following properties in application.properties file.

```
#MongoDB connection properties
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=ntsp713db
spring.data.mongodb.username=testuser
spring.data.mongodb.password=testuser
```

Step 3: Develop the Document class.

Note: If collection in MongoDB is not already there then it creates the collection with Document class name and creates attributes with the Document class property names.

Step4: Develop Repository Interface extending from MongoRepository (I).

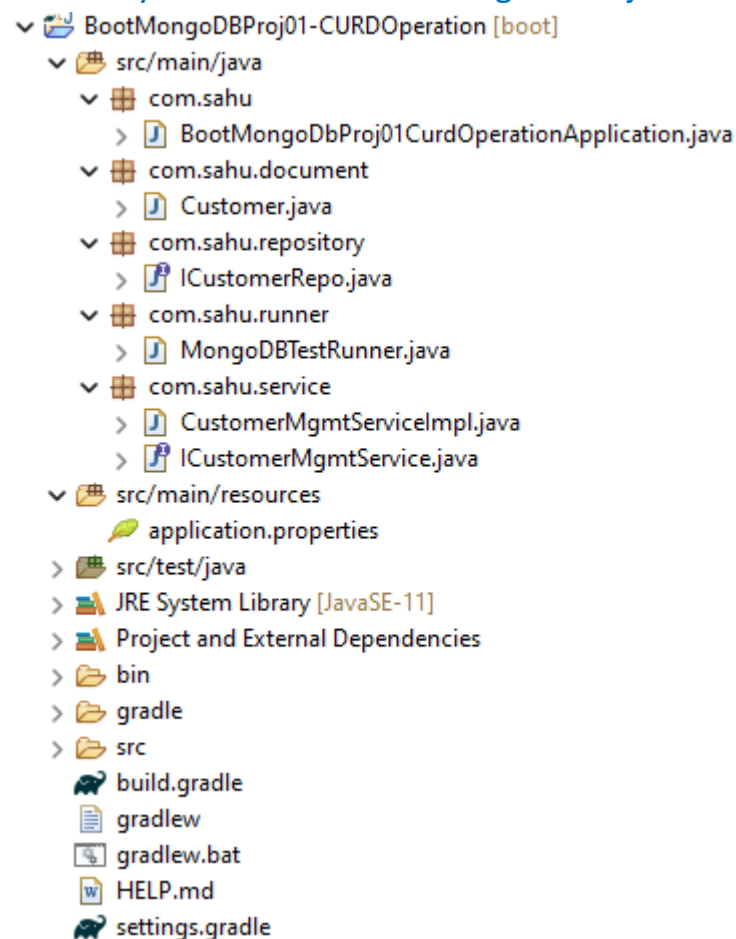
MongoRepository has the following methods,

```
▼ ⓘ MongoRepository<T, ID>
  ● ▲ saveAll(Iterable<S>) <S extends T> : List<S>
  ● ▲ findAll() : List<T>
  ● ▲ findAll(Sort) : List<T>
  ● ▲ insert(S) <S extends T> : S
  ● ▲ insert(Iterable<S>) <S extends T> : List<S>
  ● ▲ findAll(Example<S>) <S extends T> : List<S>
  ● ▲ findAll(Example<S>, Sort) <S extends T> : List<S>
```

Step 5: Develop the Service interface and service implementation class.

Step 6: Develop the Runner class for testing the application and run the application.

Directory Structure of BootMongoDBProj01-CURDOperation:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Choose the following starters during the project creation.

```
X Lombok
X Spring Data MongoDB
```

- Place the following code within their respective files.

application.properties

```
#MongoDB connection properties
spring.data.mongodb.host=localhost
#Default port number
spring.data.mongodb.port=27017
#The Logical DB name
spring.data.mongodb.database=ntsp713db
#For Authentication the username and password
spring.data.mongodb.username=testuser
spring.data.mongodb.password=testuser
```

Customer.java

```
package com.sahu.document;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Document
@Data
public class Customer {
    @Id
    private String id;
    private Integer cno;
    private String cname;
    private String caddress;
    private Double billAmt;
}
```

ICustomerRepo.java

```
package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.Customer;

public interface ICustomerRepo extends MongoRepository<Customer,
String> {

}
```

ICustomerMgmtService.java

```
package com.sahu.service;

import com.sahu.document.Customer;

public interface ICustomerMgmtService {
    public String registerCustomer(Customer customer);
}
```

CustomerMgmtServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.document.Customer;
import com.sahu.repository.ICustomerRepo;

@Service("custService")
public class CustomerMgmtServiceImpl implements ICustomerMgmtService
{

    @Autowired
    private ICustomerRepo customerRepo;

    @Override
    public String registerCustomer(Customer customer) {
        return "Customer
"+customerRepo.save(customer).getCname()+" has saved.";
    }

}
```

MongoDBTestRunner.java

```
package com.sahu.runner;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.document.Customer;
import com.sahu.service.ICustomerMgmtService;

@Component
public class MongoDBTestRunner implements CommandLineRunner {

    @Autowired
    private ICustomerMgmtService customerMgmtService;
```

```

@Override
public void run(String... args) throws Exception {
    Customer customer1 = new Customer();
    customer1.setCno(1001);
    customer1.setCname("Ramesh");
    customer1.setCaddress("Hyd");
    customer1.setBillAmt(34533.0);

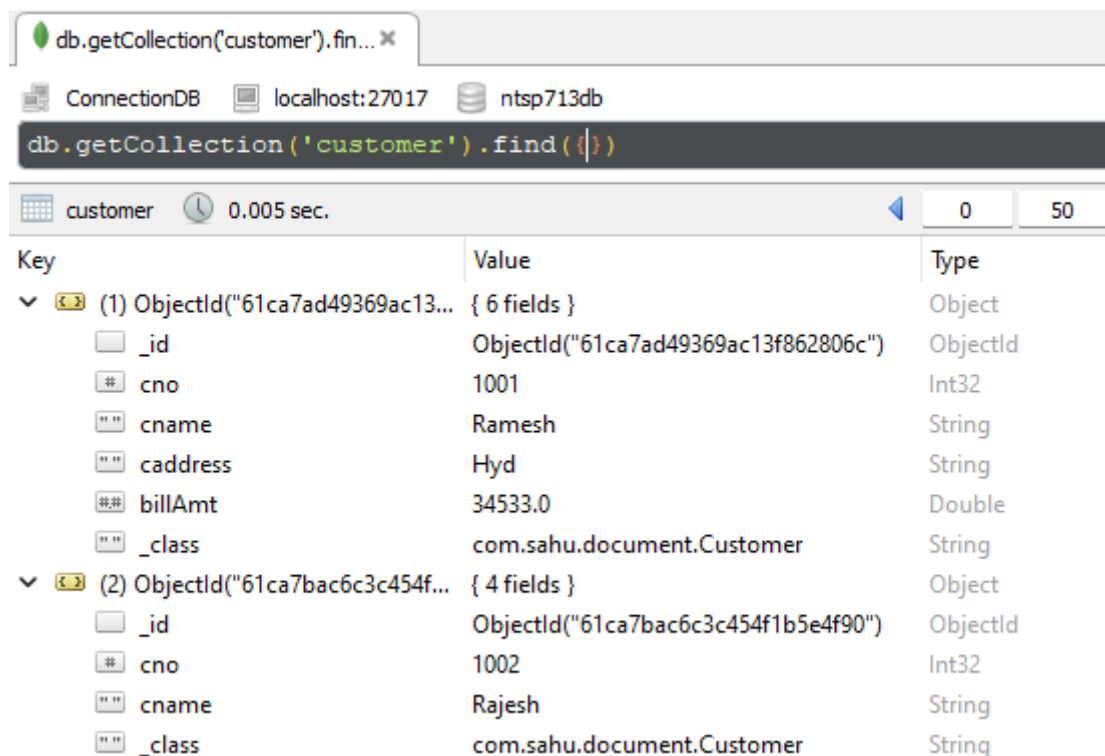
    System.out.println(customerMgmtService.registerCustomer(customer1));

    Customer customer2 = new Customer();
    customer2.setCno(1002);
    customer2.setCname("Rajesh");

    System.out.println(customerMgmtService.registerCustomer(customer2));
}
}

```

After run the project go to Robo3T you will get the following thing there.



db.getCollection('customer').find({})

customer 0.005 sec. 0 50

Key	Value	Type
▼ (1) ObjectId("61ca7ad49369ac13...")	{ 6 fields }	Object
_id	ObjectId("61ca7ad49369ac13f862806c")	ObjectId
cno	1001	Int32
cname	Ramesh	String
caddress	Hyd	String
billAmt	34533.0	Double
_class	com.sahu.document.Customer	String
▼ (2) ObjectId("61ca7bac6c3c454f...")	{ 4 fields }	Object
_id	ObjectId("61ca7bac6c3c454f1b5e4f90")	ObjectId
cno	1002	Int32
cname	Rajesh	String
_class	com.sahu.document.Customer	String



Note: By default, every document gets two default attributes if the documents are inserted from any Front-end application using MongoDB API having String property as the @Id property.

Q. What is the difference CrudRepository (I) save (-) and MongoRepository (I) insert (-) method?

Ans.

Save (-)	Insert (-)
a. Can perform insert/ update entity/ document operation.	a. Can perform only insert document operation.
b. Common method for both SQL and NoSQL DB s/w	Specific method MongoDB
c. Supports Code portability from SQL to NoSQL DB and vice-versa.	c. Does not support.
d. Recommended to use.	d. Not recommended to use.

Working with Customer ID Generator:

- For that create the below package and class below and place the following code in their respective files.
 - ▼  com.sahu.generator
 - >  IDGenerator.java
- To show we can change document class any time add a new property in document class.

IDGenerator.java

```
private Long mobileNo;
```

IDGenerator.java

```
package com.sahu.generator;

import java.util.UUID;

public class IDGenerator {

    public static String generateId() {
        return UUID.randomUUID().toString().replace("-",
""").substring(0, 10);
    }
}
```

MongoDBTestRunner.java

```
@Override
public void run(String... args) throws Exception {
    Customer customer2 = new Customer();
    customer2.setId(IDGenerator.generateId());
    customer2.setCno(1002);
    customer2.setCname("Mahesh");
    customer2.setCaddress("Vizag");
    customer2.setBillAmt(3453.67);
    customer2.setMobileNo(2345678976l);

    System.out.println(customerMgmtService.registerCustomer(customer2));
}
```

Select Operation:

ICustomerMgmtService.java

```
public List<Customer> findAllCustomer();
```

CustomerMgmtServiceImpl.java

```
@Override
public List<Customer> findAllCustomer() {
    return customerRepo.findAll();
}
```

MongoDBTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    customerMgmtService.findAllCustomer().forEach(System.out::println)
;
}
```

Delete Operation:

ICustomerMgmtService.java

```
public String removeCustomer(String id);
```


CustomerMgmtServiceImpl.java

```
@Override
public String removeCustomer(String id) {
    //Get Document by id
    Optional<Customer> optCustomer =
customerRepo.findById(id);
    if (optCustomer.isPresent()) {
        customerRepo.delete(optCustomer.get());
        return "Docuemnt has deleted";
    }
    return "Document not found";
}
```

MongoDBTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    System.out.println(customerMgmtService.removeCustomer("61ca7b
ac6c3c454f1b5e4f90"));
}
```

Finder methods in MongoRepository/ Spring Data MongoDB

✚ Same as Spring Data JPA.

✚ Syntax:

<ReturnType> findBy<Proeprties><Condition>(parameters ..)

ICustomerRepo.java

```
public List<Customer> findByBillAmtBetween(Double start, Double
end);
public List<Customer>
findByCaddressInAndMobileNoIsNotNull(String... addresses);
```

ICustomerMgmtService.java

```
public List<Customer> fetchCustomerByBillAmtRange(Double start,
Double end);
public List<Customer>
fetchCustomerByUsingCaddressAndHavingMobileNo(String... addresses);
```

CustomerMgmtServiceImpl.java

```
@Override
public List<Customer> fetchCustomerByBillAmtRange(Double start,
Double end) {
    return customerRepo.findByBillAmtBetween(start, end);
}

@Override
public List<Customer>
fetchCustomerByUsingCaddressAndHavingMobileNo(String... addresses) {
    return
customerRepo.findByCaddressInAndMobileNoIsNotNull(addresses);
}
```

MongoDBTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    customerMgmtService.fetchCustomerByBillAmtRange(3000.0,10000.
0).forEach(System.out::println);

    customerMgmtService.fetchCustomerByUsingCaddressAndHavingMo
bileNo("Hyd", "Delhi").forEach(System.out::println);
}
```

Special types in MongoDB:

- java.util.List type properties
- java.util.Set type propeties
- Array type properties
- java.util.Map type proeptries
- java.util.Properties type properties
- HAS-A property (non-collection type)
For One to One and Many to One association
- HAS-A property (Collection type)
For One to Many and Many to Many Association

Note:

- ✓ In MongoDB Array/ List/ Set properties will be treated as same properties.

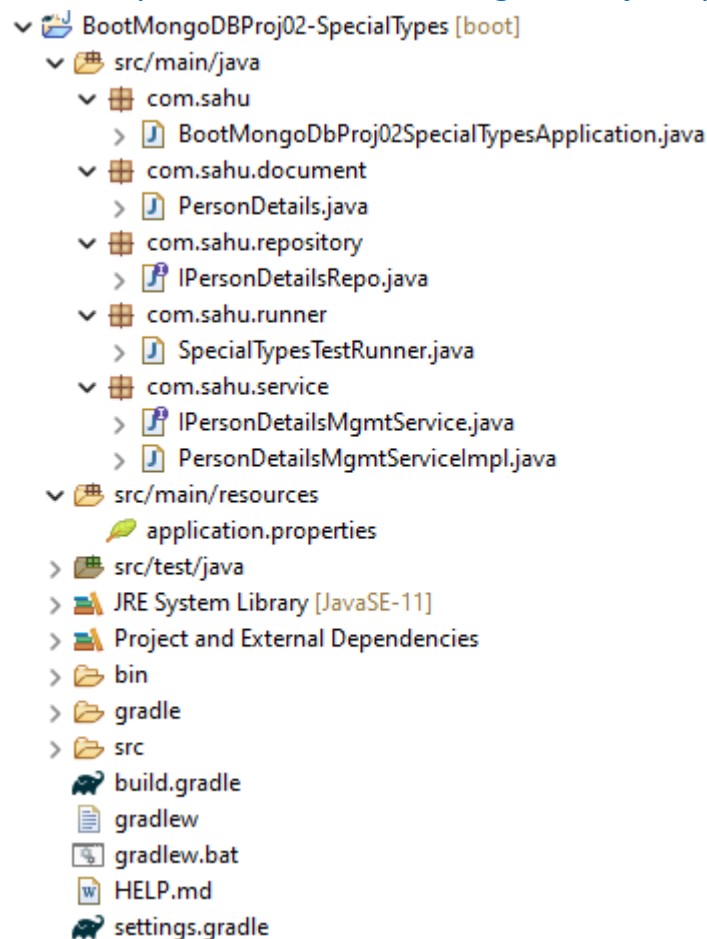
Syntax: ["...", "...", "...", ...].

- ✓ In MongoDB Map/ Properties type properties will be treated as same properties.

Syntax: {"key1": "value1", "key2": "value2", ...}.

- ✓ Array/ Collection can hold simple/ wrapper values as part Entity/ Document object [Collection Mapping (JPA)/ Special types Mapping (MongoDB)].
- ✓ Array/ Collection can hold the Associated Entity/ Document object(s) as part of Association Mapping (JPA)/ Special types Mapping (MongoDB)

Directory Structure of BootMongoDBProj02-SpecialTypes:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Choose the following starters during the project creation.

X Lombok
X Spring Data MongoDB

- Copy the application.properties file from previous project.
- Place the following code within their respective files.

PersonDetails.java

```
package com.sahu.document;

import java.time.LocalDateTime;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Document
@Data
public class PersonDetails {
    @Id
    private Integer pId;
    private String pName;
    private String pAddress;
    private LocalDateTime dob;
    private Boolean maritalStatus;
    private Set<Long> mobileNumbers;
    private String[] nickNames;
    private List<String> academics;
    private Map<String, Long> bankAccounts;
    private Properties idDetails;
}
```

IPersonDetailsRepo.java

```
package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.PersonDetails;

public interface IPersonDetailsRepo extends
MongoRepository<PersonDetails, Integer> {

}
```

IPersonDetailsMgmtService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.document.PersonDetails;

public interface IPersonDetailsMgmtService {
    public String registerPerson(PersonDetails person);
    public List<PersonDetails> fetchAllPersons();
}
```

PersonDetailsMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.document.PersonDetails;
import com.sahu.repository.IPersonDetailsRepo;

@Service("personDetailsService")
public class PersonDetailsMgmtServiceImpl implements
IPersonDetailsMgmtService {

    @Autowired
    private IPersonDetailsRepo personDetailsRepo;

    @Override
    public String registerPerson(PersonDetails person) {
        return personDetailsRepo.save(person).getPName()+" details
has saved";
    }

    @Override
    public List<PersonDetails> fetchAllPersons() {
        return personDetailsRepo.findAll();
    }
}
```

IPersonDetailsMgmtService.java

```
package com.sahu.runner;

import java.time.LocalDateTime;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Random;
import java.util.Set;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.document.PersonDetails;
import com.sahu.service.IPersonDetailsMgmtService;

@Component
public class SpecialTypesTestRunner implements CommandLineRunner {

    @Autowired
    private IPersonDetailsMgmtService personDetailsMgmtService;

    @Override
    public void run(String... args) throws Exception {
        //Prepare Document object
        PersonDetails person = new PersonDetails();
        person.setPid(new Random().nextInt());
        person.setPName("Raja");
        person.setPAddress("Hyd");
        person.setDob(LocalDateTime.of(1998, 11, 24, 12, 35));
        person.setMaritalStatus(false);
        person.setMobileNumbers(Set.of(888888888l, 999999999l,
77777l));
        person.setNickNames(new String[] {"Raj", "King", "K"});
        person.setAcademics(List.of("10th", "+2", "BCA", "MCA"));
        person.setBankAccounts(Map.of("SBI", 7654356754L, "ICICI",
564356434L));
        Properties idDetails = new Properties();
        idDetails.put("Aadhaar", "4646424343");
    }
}
```

```

        idDetails.put("Pan No", "6DB424S43");
        person.setIdDetails(idDetails);

        System.out.println(personDetailsService.registerPerson(person)
);

        personDetailsService.fetchAllPersons().forEach(System.out::pri
ntln);
    }

}

```

Association Mapping in Spring Boot MongoDB

- These are the following special types or properties in MongoDB we have for Association Mapping,
 - a. HAS-A property (non-collection type)
To build One to One and Many to One Association
 - b. HAS-A property (Collection type - List/Set/Map)
To build One to Many and Many to Many Association

HAS-A property (non-collection type):

- Documents for One-to-One Association is one Person can have only one Driving License.

Equivalent JSON document (From Parent to Child)

```

{
    _id: .....
    ..... //Person object properties ad key = value pairs
    .....
    license: {
        _id: .....
        ..... //License object properties as key = value pairs
        .....
    }
}

```

Equivalent JSON document (From Child to Parent)

```

{

```

```

    _id: .....
    .....    // License object properties ad key = value pairs
    .....
    person: {
        _id: .....
        .....    //Person object properties as key = value pairs
        .....
    }
}

```

Directory Structure of BootMongoDBProj03-Associations-OneToOne:

```

v BootMongoDBProj03-Associations-OneToOne [boot]
v src/main/java
  v com.sahu
    > BootMongoDbProj03AssociationsOneToOneApplication.java
  v com.sahu.document
    > DrivingLicense.java
    > Person.java
  v com.sahu.repository
    > IDrivingLicenseRepo.java
    > IPersonRepo.java
  v com.sahu.runner
    > AssociationTestRunner.java
  v com.sahu.service
    > IRTOMgmtService.java
    > RTOMgmtServiceImpl.java
v src/main/resources
  application.properties
> src/test/java
> JRE System Library [JavaSE-11]
> Project and External Dependencies
> bin
> gradle
> src
  build.gradle
  gradlew
  gradlew.bat
  HELP.md
  settings.gradle

```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starters are sufficient so choose them during the project creation.
- Copy the application.properties file from previous project.
- Place the following code within their respective files.

Person.java

```
package com.sahu.document;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Getter;
import lombok.Setter;

@Document
@Getter
@Setter
public class Person {
    @Id
    private Integer pId;
    private String pName;
    private String pAddress;
    private DrivingLicense license; //HAS-A Property

    public Person() {
        System.out.println("Person.Person()");
    }

    @Override
    public String toString() {
        return "Person [pId=" + pId + ", pName=" + pName + ",
pAddress=" + pAddress + "];"
    }
}
```

DrivingLicense.java

```
package com.sahu.document;

import java.time.LocalDate;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Getter;
```

```

import lombok.Setter;

@Document
@Getter
@Setter
public class DrivingLicense {
    @Id
    private Long lId;
    private String type;
    private LocalDate expiryDate;
    private Person person;

    public DrivingLicense() {
        System.out.println("DrivingLicense.DrivingLicense()");
    }

    @Override
    public String toString() {
        return "DrivingLicense [lId=" + lId + ", type=" + type + ",
expiryDate=" + expiryDate + "]";
    }
}

```

IPersonRepo.java

```

package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.Person;

public interface IPersonRepo extends MongoRepository<Person, Integer> {

}

```

IDrivingLicenseRepo.java

```

package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

```

```
import com.sahu.document.DrivingLicense;
```

```
public interface IDrivingLicenseRepo extends  
MongoRepository<DrivingLicense, Long> {  
  
}
```

IRTOMgmtService.java

```
package com.sahu.service;
```

```
import com.sahu.document.DrivingLicense;  
import com.sahu.document.Person;
```

```
public interface IRTOMgmtService {  
    public String savePersonWithLicense(Person person);  
    public String saveLicenseWithPerson(DrivingLicense license);  
}
```

RTOMgmtServiceImpl.java

```
package com.sahu.service;
```

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;
```

```
import com.sahu.document.DrivingLicense;  
import com.sahu.document.Person;  
import com.sahu.repository.IDrivingLicenseRepo;  
import com.sahu.repository.IPersonRepo;
```

```
@Service("RTOService")
```

```
public class RTOMgmtServiceImpl implements IRTOMgmtService {
```

```
    @Autowired
```

```
    private IPersonRepo personRepo;
```

```
    @Autowired
```

```
    private IDrivingLicenseRepo drivingLicenseRepo;
```

```
    @Override
```

```

    public String savePersonWithLicense(Person person) {
        return person.getPName()+"'s details has saved with his
License No. : "+personRepo.save(person).getLicense().getLId();
    }

    @Override
    public String saveLicenseWithPerson(DrivingLicense license) {
        return
drivingLicenseRepo.save(license).getPerson().getPName()+"'s details has
saved with his License No. : "+license.getLId();
    }
}

```

AssociationTestRunner.java

```

package com.sahu.runner;

import java.time.LocalDate;
import java.util.Random;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.document.DrivingLicense;
import com.sahu.document.Person;
import com.sahu.service.IRTOMgmtService;

@Component
public class AssociationTestRunner implements CommandLineRunner {

    @Autowired
    private IRTOMgmtService irtoMgmtService;

    @Override
    public void run(String... args) throws Exception {
        //Prepare Person Object
        Person person = new Person();
        person.setPid(new Random().nextInt(1000000));
    }
}

```

```

        person.setPName("Ritesh");
        person.setPAddress("Hyd");
        //Prepare Child Object
        DrivingLicense license = new DrivingLicense();
        license.setLId(Long.valueOf(new Random().nextInt(5000000)));
        license.setType("Two-wheeler");
        license.setExpiryDate(LocalDate.of(2025, 12, 30));
        //Map objects
        person.setLicense(license);
        //Invoke method

        System.out.println(irtoMgmtService.savePersonWithLicense(person));

        //Prepare Person Object
        Person person1 = new Person();
        person1.setPId(new Random().nextInt(1000000));
        person1.setPName("Hitesh");
        person1.setPAddress("Vizg");
        //Prepare Child Object
        DrivingLicense license1 = new DrivingLicense();
        license1.setLId(Long.valueOf(new
Random().nextInt(5000000)));
        license1.setType("Two-wheeler");
        license1.setExpiryDate(LocalDate.of(2025, 12, 30));
        //Map objects
        license1.setPerson(person1);
        //Invoke method

        System.out.println(irtoMgmtService.saveLicenseWithPerson(license1
    );

    }

}

```

Note: No FK in MongoDB. Documents in association means one document holding another document as nested document.

Select operation on One-to-One Association Mapping:

- Place the following code with their respective files.

IRTOmgmtService.java

```
public List<Person> fetchAllPersons();  
public List<DrivingLicense> fetchAllDrivingLicenses();
```

RTOMgmtServiceImpl.java

```
@Override  
public List<Person> fetchAllPersons() {  
    return personRepo.findAll();  
}  
  
@Override  
public List<DrivingLicense> fetchAllDrivingLicenses() {  
    return drivingLicenseRepo.findAll();  
}
```

AssociationTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
    irtoMgmtService.fetchAllPersons().forEach(per ->{  
        System.out.println("Parent : "+per);  
        System.out.println("Child : "+per.getLicense());  
    });  
  
    irtoMgmtService.fetchAllDrivingLicenses().forEach(dl ->{  
        System.out.println("Child : "+dl);  
        System.out.println("Parent : "+dl.getPerson());  
    });  
}
```

HAS-A property (collection type):

- We take List or Map type collections type properties as the HAS-A properties in Document to build One to Many or Many to Many associations.

HAS-A property type is List/ Set collection then the Document

```
{  
    _id: ....  
    .... //Parent object properties
```

```

....
HAS-A Property: {      //Child object documents
    {...},
    {...},
    {...}
}
}

```

HAS-A property type is Map collection then the Document

```

{
    _id: ....
    .... //Parent object properties

    ....
    HAS-A Property: {      //Child object properties
        {...},
        {...},
        {...}
    }
}

```

E.g.,

- One Person can have multiple Visas (One to Many).
- One Person can have multiple vehicles (One to Many).

Note: One Document can have multiple HAS-A properties to representing multiple types of associations with child documents.

Directory Structure of BootMongoDBProj04-Associations-OneToMany:

```

▼ BootMongoDBProj04-Associations-OneToMany [boot]
  ▼ src/main/java
    ▼ com.sahu
      > BootMongoDbProj04AssociationsOneToManyApplication.java
    ▼ com.sahu.document
      > Medal.java
      > MedalType.java
      > Player.java
      > Sport.java
    ▼ com.sahu.repository
      > IMedalRepo.java
      > IPlayerRepo.java
      > ISportRepo.java
    ▼ com.sahu.runner
      > AssociationsTestRunner.java
    ▼ com.sahu.service

```

```

    > ISportsMgmtService.java
    > SportsMgmtServiceImpl.java
  ▾ src/main/resources
    application.properties
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Project and External Dependencies
  > bin
  > gradle
  > src
  build.gradle
  gradlew
  gradlew.bat
  HELP.md
  settings.gradle

```

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starters are sufficient so choose them during the project creation.
- Copy the application.properties file from previous project.
- Place the following code within their respective files.

Player.java

```

package com.sahu.document;

import java.util.Map;
import java.util.Set;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Getter;
import lombok.Setter;

@Document
@Setter
@Getter
public class Player {
    @Id
    private Integer pId;
    private String pName;
    private String pAddress;
}

```



```

private String country;
private Set<Sport> sports;
private Map<String, Medal> medals;

public Player() {
    System.out.println("Player.Player()");
}

@Override
public String toString() {
    return "Player [pId=" + pId + ", pName=" + pName + ",
pAddress=" + pAddress + ", country=" + country + "]";
}
}

```

Sport.java

```

package com.sahu.document;

import java.util.Arrays;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Getter;
import lombok.Setter;

@Document
@Setter
@Getter
public class Sport {
    @Id
    private Long id;
    private String name;
    private Boolean teamSport;
    private Boolean isOlympicSport;
    private String[] kitItems;

    public Sport() {
        System.out.println("Sport.Sport()");
    }
}

```

```

@Override
public String toString() {
    return "Sport [id=" + id + ", name=" + name + ", teamSport=" +
teamSport + ", isOlympicSport=" + isOlympicSport
        + ", kitItems=" + Arrays.toString(kitItems) + "];"
    }
}

```

MedalType.java

```

package com.sahu.document;

public enum MedalType {
    GOLD,
    SILVER,
    BRONZ
}

```

Medal.java

```

package com.sahu.document;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Getter;
import lombok.Setter;

@Document
@Setter
@Getter
public class Medal {
    @Id
    private String id;
    private MedalType type;
    private String sportName;
    private String eventName;
    private Double priceMoney;

    public Medal() {
        System.out.println("Medal.Medal()");
    }
}

```

```

    }

    @Override
    public String toString() {
        return "Medal [id=" + id + ", type=" + type + ", sportName=" +
sportName + ", eventName=" + eventName
        + ", priceMoney=" + priceMoney + "]\n";
    }
}

```

ISportRepo.java

```

package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.Sport;

public interface ISportRepo extends MongoRepository<Sport, Long> {
}

```

IPlayerRepo.java

```

package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.Player;

public interface IPlayerRepo extends MongoRepository<Player, Integer> {
}

```

IMedalRepo.java

```

package com.sahu.repository;

import org.springframework.data.mongodb.repository.MongoRepository;

import com.sahu.document.Medal;

public interface IMedalRepo extends MongoRepository<Medal, String> {
}

```

ISportsMgmtService.java

```
package com.sahu.service;

import java.util.List;

import com.sahu.document.Player;

public interface ISportsMgmtService {
    public String registerPlayer(Player player);
    public List<Player> getAllPlayers();
}
```

SportsMgmtServiceImpl.java

```
package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.document.Player;
import com.sahu.repository.IPlayerRepo;

@Service("sportsService")
public class SportsMgmtServiceImpl implements ISportsMgmtService {

    @Autowired
    private IPlayerRepo playerRepo;

    @Override
    public String registerPlayer(Player player) {
        return "Player "+playerRepo.save(player).getPName()+" has
registered successfully.";
    }

    @Override
    public List<Player> getAllPlayers() {
        return playerRepo.findAll();
    }
}
```

AssociationsTestRunner.java

```
package com.sahu.runner;

import java.util.Map;
import java.util.Random;
import java.util.Set;
import java.util.UUID;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.document.Medal;
import com.sahu.document.MedalType;
import com.sahu.document.Player;
import com.sahu.document.Sport;
import com.sahu.service.ISportsMgmtService;

@Component
public class AssociationsTestRunner implements CommandLineRunner {

    @Autowired
    private ISportsMgmtService sportsMgmtService;

    @Override
    public void run(String... args) throws Exception {
        //Prepare player object
        Player player = new Player();
        player.setPid(new Random().nextInt(100000));
        player.setPName("P. V. Sindhu");
        player.setPAddress("Hyd");
        player.setCountry("India");

        //Child objects - 1
        Sport sport1 = new Sport();
        sport1.setPid(Long.valueOf(new Random().nextInt(100000)));
        sport1.setName("batminton");
        sport1.setIsOlympicSport(true);
        sport1.setTeamSport(false);
        Sport sport2 = new Sport();
```

```

sport2.setId(Long.valueOf(new Random().nextInt(100000)));
sport2.setName("cricket");
sport2.setIsOlympicSport(false);
sport2.setTeamSport(true);

//Child objects - 2
Medal medal1 = new Medal();
medal1.setId(UUID.randomUUID().toString().replace("-",
"").substring(0, 10));
medal1.setType(MedalType.BRONZ);
medal1.setPriceMoney(100000000.0);
medal1.setSportName("Batminton");
medal1.setEventName("Tokyo-Olympics-2021");
Medal medal2 = new Medal();
medal2.setId(UUID.randomUUID().toString().replace("-",
"").substring(0, 10));
medal2.setType(MedalType.SILVER);
medal2.setPriceMoney(50000000.0);
medal2.setSportName("Batminton");
medal2.setEventName("RIO-Olympics-2016");

player.setSports(Set.of(sport1, sport2));
player.setMedals(Map.of("Tokyo-Bronze", medal1, "RIO-
Silver", medal2));

//System.out.println(sportsMgmtService.registerPlayer(player));

sportsMgmtService.getAllPlayers().forEach(playerDetails->{
    System.out.println("Parent : "+playerDetails);
    playerDetails.getSports().forEach(sport->{
        System.out.println("Child-1 : "+sport);
    });
    playerDetails.getMedals().forEach((name, medal)->{
        System.out.println("Child-2 : "+name+" :
"+medal);
    });
});
}
}

```

MongoDB @Query method with Projections

- ✚ Projections in Query means selecting either specific single column or multiple column values.
- ✚ By default, @Query placed in MongoDB app selects all the fields of the Document class
- ✚ Use "fields" attribute of @Query annotation having field name with 0 or 1 value.

Syntax: @Query (fields="{property:0/1, property:0/1}").

- 1 indicates involve the field/ variable/ property in the select query.
- 0 indicates do not involve the field/ variable/ property in the select query.
- For all properties default value is 0, but for @Id property the default value is 1.

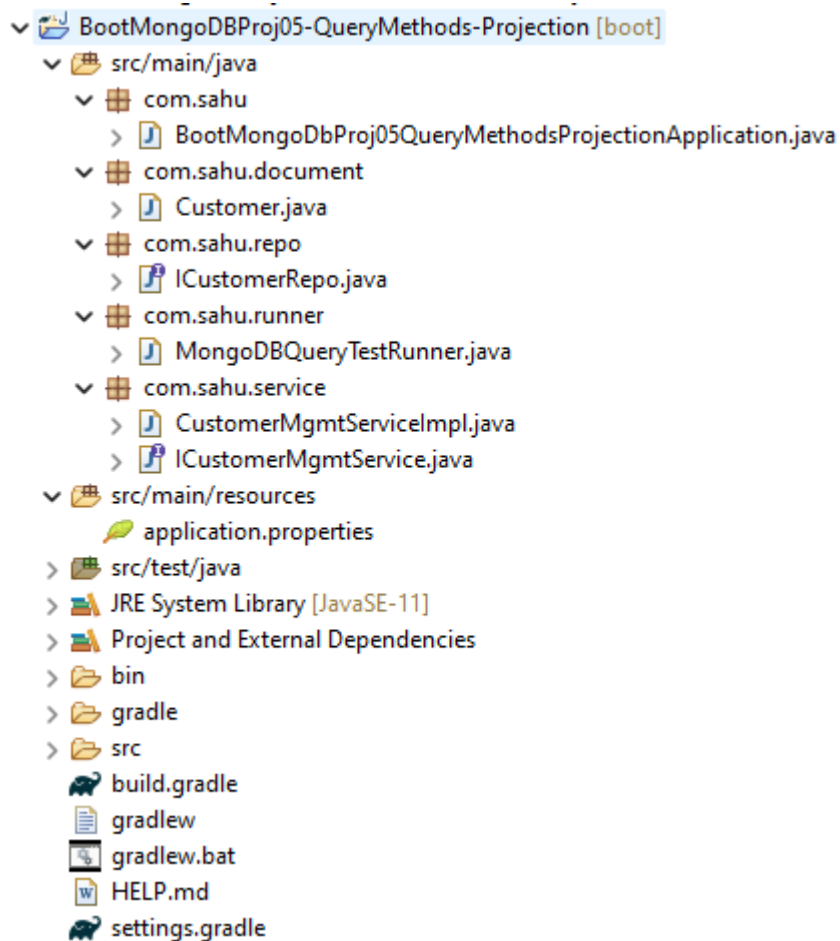
Note: Only for @Id property we can 0 or 1 in Projection operations for remaining only "1" is allowed (i.e., involve the property) if you do not want to involve the property do not take "0" just ignore to place that property.

- ✚ "value" attribute of @Query is useful to specify the where condition clauses [\[Reference Document\]](#).

E.g.,

- @Query (value="{cadd:?0}", fields="{cno:0, cname:1, billAmt:1}")
This is equal to "SELECT CNO, CNAME, BILLAMT FROM CUSTOMER WHERE CADD=?" (SQL)
(By Default, @Id property will be selected because its default value is 1).
- @Query (value="{cadd:?0}", fields="{cname:1, billAmt:1}")
This is equal to "SELECT CNAME, BILLAMT FROM CUSTOMER WHERE CADD=?" (SQL).
- @Query (value="{cadd:?0}", fields="{ }")
(or) @Query (value="{cadd:?0}") [Special case]
To get all fields/ property values and this is equal to "SELECT * FROM CUSTOMER WHERE CADD=?" (SQL).
- @Query (value="{cadd:?0, cname:?1}")
This is equal to "SELECT * FROM CUSTOMER WHERE CADD=? AND CNAME=?" (SQL).

Directory Structure of BootMongoDBProj05-QueryMethods-Projection:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starters are sufficient so choose them during the project creation.
- Copy the application.properties file from previous project.
- Place the following code within their respective files.

Customer.java

```
package com.sahu.document;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Document
@Data
```



```

public class Customer {
    @Id
    private String id;
    private Integer cno;
    private String cname;
    private String caddress;
    private Double billAmt;
    private Long mobileNo;
}

```

ICustomerRepo.java

```

package com.sahu.repo;

import java.util.List;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import com.sahu.document.Customer;

public interface ICustomerRepo extends MongoRepository<Customer,
Integer> {
    @Query(fields = "{id:1, cno:1, cname:1, caddress:1}", value =
"{caddress:?0}")
    public List<Object[]> getCustomersDataAddress(String cAddress);

    // @Query(fields = "{}", value = "{caddress:?0}")
    @Query(value = "{caddress:?0}")
    public List<Customer> getCustomersByAddress(String cAddress);

    @Query(fields = "{id:1, cno:1, cname:1, caddress:1}", value =
"{caddress:?0, cname:?1}")
    public List<Object[]> getCustomersByAddressAndName(String
cAddress, String cName);

    // @Query(value="{ $or: [{field1:?0}, {field2:?1}]}")
    @Query(fields = "{id:1, cno:1, cname:1, caddress:1}", value =
"{ $or: [{caddress:?0}, {cname:?1}]}")
    public List<Object[]> getCustomersByAddressOrName(String
cAddress, String cName);
}

```

```

        //@Query(value="{field1:{$gte:?0}, field2:{$lte:?1}}")
        //@Query(fields = "{id:1, cno:1, cname:1, caddress:1, billAmt:1}",
value = "{billAmt:{$gte:?0}, billAmt:{$lte:?1}}")
        @Query(fields = "{id:1, cno:1, cname:1, caddress:1, billAmt:1}", value
= "{billAmt:{$gte:?0, $lte:?1}}")
        public List<Object[]> getCustomersByBillAmountRange(Double start,
Double end);
    }

```

ICustomerMgmtService.java

```

package com.sahu.service;

import java.util.List;

import com.sahu.document.Customer;

public interface ICustomerMgmtService {
    public List<Object[]> fetchCustomersDataByAddress(String cAddress);
    public List<Customer> fetchCustomersByAddress(String cAddress);
    public List<Object[]> fetchCustomersByAddressAndName(String
cAddress, String cName);
    public List<Object[]> fetchCustomersByAddressOrName(String
cAddress, String cName);
    public List<Object[]> fetchCustomersByBillAmountRange(Double
start, Double end);
}

```

CustomerMgmtServiceImpl.java

```

package com.sahu.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.document.Customer;
import com.sahu.repo.ICustomerRepo;

@Service("customerService")

```

```

public class CustomerMgmtServiceImpl implements
ICustomerMgmtService {

    @Autowired
    private ICustomerRepo customerRepo;

    @Override
    public List<Object[]> fetchCustomersDataByAddress(String cAddress)
    {
        return customerRepo.getCustomersDataAddress(cAddress);
    }

    @Override
    public List<Customer> fetchCustomersByAddress(String cAddress) {
        return customerRepo.getCustomersByAddress(cAddress);
    }

    @Override
    public List<Object[]> fetchCustomersByAddressAndName(String
cAddress, String cName) {
        return
customerRepo.getCustomersByAddressAndName(cAddress, cName);
    }

    @Override
    public List<Object[]> fetchCustomersByAddressOrName(String
cAddress, String cName) {
        return
customerRepo.getCustomersByAddressOrName(cAddress, cName);
    }

    @Override
    public List<Object[]> fetchCustomersByBillAmountRange(Double
start, Double end) {
        return customerRepo.getCustomersByBillAmountRange(start,
end);
    }
}

```

MongoDBQueryTestRunner.java

```
package com.sahu.runner;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.service.ICustomerMgmtService;

@Component
public class MongoDBQueryTestRunner implements CommandLineRunner {

    @Autowired
    private ICustomerMgmtService customerMgmtService;

    @Override
    public void run(String... args) throws Exception {
        //invoke method

        customerMgmtService.fetchCustomersDataByAddress("Hyd").forEach
(doc -> {
            for (Object attr : doc) {
                System.out.print(attr+" ");
            }
            System.out.println();
        });

        customerMgmtService.fetchCustomersByAddress("Hyd").forEach(System.out::println);

        customerMgmtService.fetchCustomersByAddressAndName("Hyd",
"Ramesh").forEach(doc -> {
            for (Object attr : doc) {
                System.out.print(attr+" ");
            }
            System.out.println();
        });
    }
}
```

```

        customerMgmtService.fetchCustomersByAddressOrName("Hyd",
"Rajesh").forEach(doc -> {
            for (Object attr : doc) {
                System.out.print(attr+" ");
            }
            System.out.println();
        });

        customerMgmtService.fetchCustomersByBillAmountRange(10000.0,
50000.0).forEach(doc -> {
            for (Object attr : doc) {
                System.out.print(attr+" ");
            }
            System.out.println();
        });
    }
}

```

Regular Expression in MongoDB @Query methods

	In SQL	In MongoDB
Starting with given data	data%	^data
Ending with given data	%data	data\$
Having given data	%data%	data or ^data\$ (It is working in few versions of MongoDB)

For Documentation: [Click Here](#)

E.g., @Query (value="{cadd:{\$regex:?0}}")
public List<Customer> getRegData(String address);

While calling method from Client app

- List<Customer> list = repo.getRegData("^h") -> gives all docs whose cadd starts with "h".
- List<Customer> list = repo.getRegData("h\$") -> gives all docs whose cadd ends with "h".
- List<Customer> list = repo.getRegData("h") -> gives all docs whose cadd contains "h".

ICustomerRepo.java

```
@Query(value = "{caddress:{regex:?0}}")  
public List<Customer> getCustomersByRegAddress(String address);
```

ICustomerMgmtService.java

```
public List<Customer> fetchCustomersByRegAddress(String address);
```

CustomerMgmtServiceImpl.java

```
@Override  
public List<Customer> fetchCustomersByRegAddress(String address) {  
    return customerRepo.getCustomersByRegAddress(address);  
}
```

MongoDBQueryTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
  
    customerMgmtService.fetchCustomersByRegAddress("^H").forEach(S  
ystem.out::println);  
  
    customerMgmtService.fetchCustomersByRegAddress("g$").forEach(S  
ystem.out::println);  
  
    customerMgmtService.fetchCustomersByRegAddress("y").forEach(Sy  
stem.out::println);  
}
```

Some other Parameters in @Query method

- ✚ The "count" param of @Query Annotation returns numeric value representing the count of document for the given condition.
- ✚ The Query in MongoDB whose inputs are not hardcoded and passed through parameters like ?0, ?1 and etc. is called Dynamic Query.
- ✚ The MongoDB queries do not support named params they support only JPA style ordinal params (?0, ?1).
- ✚ "sort" param of @Query annotation can sort the data (documents or selected fields) based on the given field indicating "-1" for descending order and "1" for ascending order.

Note: The input values "1, -1" must be hardcoded and can't be taken through parameters/ arguments.

- ✚ Using delete=true param in @Query we can delete the selected document based on the condition placed in "value" param.
- ✚ Using exists=true param in @Query we can check whether documents are available within given condition or not. If available method returns true otherwise returns false.

ICustomerRepo.java

```
@Query(value = "{billAmt:{$gte:?0, $lte:?1}}", count = true)
public Integer getCustomersCountByBillAmountRange(Double start,
Double end);

@Query(value="{}", sort = "{billAmt:-1}")
public List<Customer> getCustomersByBillAmountSorted();

@Query(value="{billAmt:null}", delete = true)
public Integer deleteCustomersWithNoBillAmount();

@Query(value="{billAmt:{$gte:?0, $lte:?1}}", exists = true)
public Boolean areThereCustomersWithBillAmountRang(Double start,
Double end);
```

ICustomerMgmtService.java

```
public Integer fetchCustomersCountByBillAmountRange(Double start,
Double end);
public List<Customer> fetchCustomersByBillAmountSorted();
public Integer removeCustomersWithNoBillAmount();
public Boolean checkCustomersExistWithBillAmountRang(Double
start, Double end);
```

CustomerMgmtServiceImpl.java

```
@Override
public Integer fetchCustomersCountByBillAmountRange(Double start,
Double end) {
    return
customerRepo.getCustomersCountByBillAmountRange(start, end);
}
```

```

@Override
public List<Customer> fetchCustomersByBillAmountSorted() {
    return customerRepo.getCustomersByBillAmountSorted();
}

@Override
public Integer removeCustomersWithNoBillAmount() {
    return customerRepo.deleteCustomersWithNoBillAmount();
}

@Override
public Boolean checkCustomersExistWithBillAmountRang(Double
start, Double end) {
    return
customerRepo.areThereCustomersWithBillAmountRang(start, end);
}

```

MongoDBQueryTestRunner.java

```

@Override
public void run(String... args) throws Exception {

    System.out.println("Count of docs having bill amount range
10000 to 50000 :
"+customerMgmtService.fetchCustomersCountByBillAmountRange(10000.0
, 50000.0));

    customerMgmtService.fetchCustomersByBillAmountSorted().forEach(
System.out::println);

    System.out.println("No. of document deleted :
"+customerMgmtService.removeCustomersWithNoBillAmount());

    System.out.println("Is customers exist with bill amount range
10000 to 50000 :
"+customerMgmtService.checkCustomersExistWithBillAmountRang(10000.
0, 50000.0));

}

```


Spring Boot MongoDB using MongoTemplate

- ✚ MongoTemplate is given based Template Method Design pattern that says, I take care of logics and you just perform specific logics development.
- ✚ This is very much similar to working with JdbcTemplate, HibernateTemplate, JndiTemplate classes.
- ✚ If the persistence operations simple then prefer using MongoRepository style persistence logics.
- ✚ If the persistence operations complex then prefer using MongoTemplate style persistence logics.
- ✚ If we add Spring Boot MongoDB starter to the project, then MongoTemplate class object will come automatically as Spring Bean through autoconfiguration. This object can be injected to service implementation class in order to use them for persistence operations.

Note:

- ✓ If needed we can place both styles of persistence logic in one application.
- ✓ Performing bulk non-select operations is bit complex using MongoRepository that is very much simplified in MongoTemplate.
- ✓ Working with complex queries is bit difficult in MongoRepository that the same process is simplified in MongoTemplate.
- ✓ MongoTemplate provides both direct methods and methods with Callback interfaces in order to persistence operations.
- ✓ Callback interfaces provide callback methods allowing us to write logics directly in Native API like JDBC API, Mongo API, Hibernate API etc. by using container supplied readymade objects.
- ✓ While working MongoTemplate there is no need of taking MongoRepository.

Procedure to develop Spring Boot MongoDB Application using MongoTemplate

Step 1: Create Spring Boot starter project adding Spring Boot MongoDB, Lombok API starters.

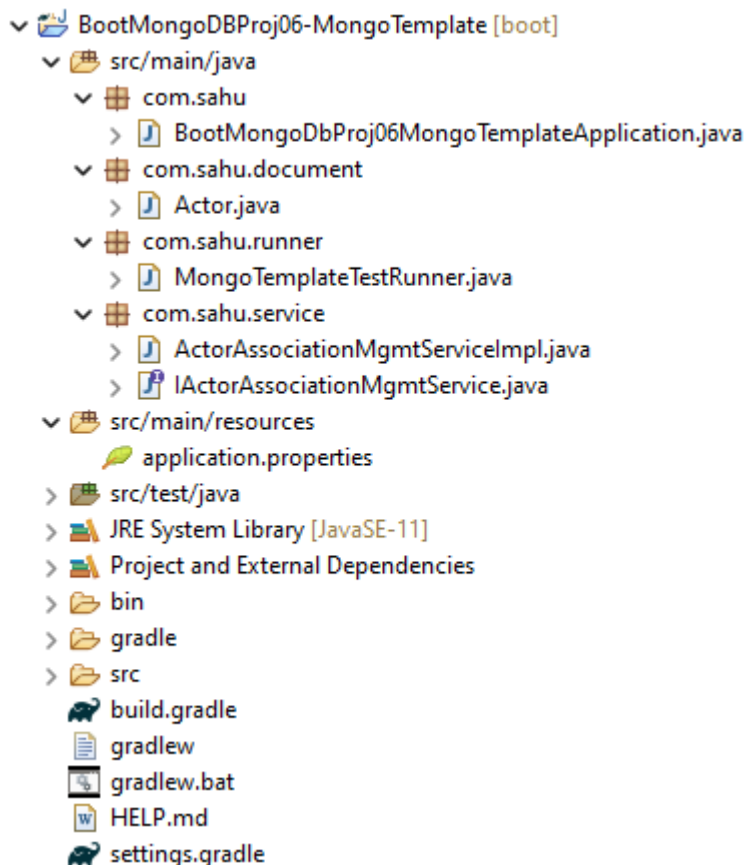
Step 2: Develop Document class.

Step 3: Develop application.properties having MongoDB connection properties.

Step 4: Develop service interface and implementation class and use MongoTemplate class there and no need of Repository interface.

Step 5: Develop the runner class and run it.

Directory Structure of BootMongoDBProj06-QueryMethods-Projection:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Previous project starters are sufficient so choose them during the project creation.
- Copy the application.properties file from previous project.
- Place the following code within their respective files.

Actor.java

```
package com.sahu.document;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Document
@Data
public class Actor {
    @Id
```

```

    private Integer actorId;
    private String actorName;
    private String category;
    private Float age;
    private Long mobileNo;
}

```

IActorAssociationMgmtService.java

```

package com.sahu.service;

import com.sahu.document.Actor;

public interface IActorAssociationMgmtService {
    public String registerActor(Actor actor);
}

```

ActorAssociationMgmtServiceImpl.java

```

package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Service;

import com.sahu.document.Actor;

@Service("actorService")
public class ActorAssociationMgmtServiceImpl implements
IActorAssociationMgmtService {

    @Autowired
    private MongoTemplate mongoTemplate;

    @Override
    public String registerActor(Actor actor) {
        //Actor savedActor = mongoTemplate.save(actor, "Artist");
        //Creates the collection with the given name Artist
        Actor savedActor = mongoTemplate.save(actor); //Creates the
collection with the given Document class name "Actor"
        return savedActor.getActorName()+" actor has registered";
    }
}

```

MongoTemplateTestRunner.java

```
package com.sahu.runner;

import java.util.Random;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.document.Actor;
import com.sahu.service.IActorAssociationMgmtService;

@Component
public class MongoTemplateTestRunner implements CommandLineRunner
{

    @Autowired
    private IActorAssociationMgmtService actorAssociationMgmtService;

    @Override
    public void run(String... args) throws Exception {
        Actor actor = new Actor();
        actor.setActorId(new Random().nextInt(100000));
        actor.setActorName("Akshay Kumar");
        actor.setAge(58.0f);
        actor.setCategory("Hero");
        actor.setMobileNo(9865766754l);

        System.out.println(actorAssociationMgmtService.registerActor(actor)
    );
    }
}
```

Q. What is the difference between insert (-) and saver (-) method of MongoTemplate?

Ans.

- insert (-) method supports only insert document operation whereas save (-) supports both insert document, update document operation.

- Insert (-) method support bulk/ batch insertion by taking collection of Document class objects whereas save (-) does not support the same.

Note: upsert (-) and save (-) method functionality is same.

insertAll(Collection)b and findAll(Document Class):

IActorAssociationMgmtService.java

```
public String registerActorsGroup(List<Actor> actorsList);  
public List<Actor> getAllActors();
```

ActorAssociationMgmtServiceImpl.java

```
@Override  
public String registerActorsGroup(List<Actor> actorsList) {  
    return actorsList.size()+" no. of actors are  
    saved."+mongoTemplate.insertAll(actorsList);  
}  
  
@Override  
public List<Actor> getAllActors() {  
    return mongoTemplate.findAll(Actor.class);  
}
```

MongoTemplateTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
    Actor actor1 = new Actor();  
    actor1.setActorId(new Random().nextInt(100000));  
    actor1.setActorName("Sonu Sood");  
    actor1.setAge(45.0f);  
    actor1.setCategory("Hero");  
    actor1.setMobileNo(6765766754l);  
    Actor actor2 = new Actor();  
    actor2.setActorId(new Random().nextInt(100000));  
    actor2.setActorName("Salman Khan");  
    actor2.setAge(55.0f);  
    actor2.setCategory("Hero");  
    actor2.setMobileNo(8865908674l);  
}
```

```

        System.out.println(actorAssociationMgmtService.registerActorsGroup(
            List.of(actor1, actor2)));

        actorAssociationMgmtService.getAllActors().forEach(System.out::println);
    }

```

Note: Every Repository interface of Spring Data module binds to one entity/document class, whereas the Template class object do not bind to any entity/document class and the Template class object is common for working with all entity/document classes.

Finding documents with Conditions/ Criteria:

- We can use find (Query query, ...) methods for this operation, where Query object represents the Criteria condition (where clause condition through Java statements).

IActorAssociationMgmtService.java

```

public List<Actor> getAllActorsByCategory(String category);

```

ActorAssociationMgmtServiceImpl.java

```

@Override
public List<Actor> getAllActorsByCategory(String category) {
    Query query = new Query();
    query.addCriteria(Criteria.where("category").is(category));
    List<Actor> actorList = mongoTemplate.find(query,
        Actor.class);
    return actorList;
}

```

MongoTemplateTestRunner.java

```

@Override
public void run(String... args) throws Exception {

    actorAssociationMgmtService.getAllActorsByCategory("Hero").forEach(
        System.out::println);
}

```

findByld (id, doc class), findByld (id, doc class, collection name) of MongoTemplate:

- This method is given to search and get single Document object based on given id value.
- Signature:
public <T> T findByld(Object id, Class<T> entityClass);

IActorAssociationMgmtService.java

```
public Actor getActorByActorId(Integer actorId);
```

ActorAssociationMgmtServiceImpl.java

```
@Override  
public Actor getActorByActorId(Integer actorId) {  
    return mongoTemplate.findByld(actorId, Actor.class);  
}
```

MongoTemplateTestRunner.java

```
@Override  
public void run(String... args) throws Exception {  
    System.out.println(actorAssociationMgmtService.getActorByActorId(  
31434));  
}
```

- findByld (-, -): for single document retrieving.
- findAll (-): for all documents retrieving.
- find (Query, -, -): for retrieving single or multiple documents based on condition.

findAndModify (Query, Update, doc class):

- Performs single document retrieving based on given Query object condition and modifies the document with the given Update object data.
- Signature:
public <T> T findAndModify(Query query, UpdateDefinition update, Class<T> entityClass);

IActorAssociationMgmtService.java

```
public String updateActorByActorId(Integer id, String newAddress,  
Long newMobileNo);
```

ActorAssociationMgmtServiceImpl.java

```
@Override
public String updateActorByActorId(Integer id, String newAddress,
Long newMobileNo) {
    //Prepare Query object
    Query query = new Query();
    query.addCriteria(Criteria.where("actorId").is(id));
    //Prepare update object for modifying the doc
    Update update = new Update();
    update.set("actorAddress", newAddress);
    update.set("mobileNo", newMobileNo);
    //modify
    Actor actor = mongoTemplate.findAndModify(query, update,
Actor.class);
    return actor==null?"Actor not found for updation":"Actor
found and updated";
}
```

MongoTemplateTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    System.out.println(actorAssociationMgmtService.updateActorByActo
rId(31434, "Mumbai", 987985454));
}
```

updateMulti(Query, update, doc class):

- This method is useful to perform bulk update operation for the given Query condition with given Update object data.
- Signature:
public UpdateResult updateMulti(Query query, UpdateDefinition update, Class<?> entityClass);

Note: Document class is compulsory to provide.

IActorAssociationMgmtService.java

```
public String updateActorsRenumerationByCategoryAndAge(String
category, Float age, Double renumeration);
```


ActorAssociationMgmtServiceImpl.java

```
@Override
    public String updateActorsRenumerationByCategoryAndAge(String
category, Float age, Double renumeration) {
    //Prepare query object
    Query query = new
Query().addCriteria(Criteria.where("category").is(category).andOperator(Cri
teria.where("age").gte(age)));
    //Prepare update object
    Update update = new Update().set("renumeration",
renumeration);
    //Invoke updateMulti method
    UpdateResult result = mongoTemplate.updateMulti(query,
update, Actor.class);
    return result.getModifiedCount()+" no. of actors are modified";
}
```

MongoTemplateTestRunner.java

```
@Override
    public void run(String... args) throws Exception {

        System.out.println(actorAssociationMgmtService.updateActorsRenu
merationByCategoryAndAge("Hero", 40.0f, 100000000.0));
    }
```

upsert(Query, Update , doc class):

- Capable of performing both insert or update operation.
- If document is not found for the given Query object condition, then it will try to insert new document with given Update object data.
- If document is found for the given Query object condition, then it will try to update the documents with given Update object data.
- Signature:

```
public UpdateResult upsert(Query query, UpdateDefinition update,
Class<?> entityClass);
```

IActorAssociationMgmtService.java

```
public String saveOrUpdateActorByRenumeration(Double start,
Double end, String newAddress, Float newAge, Long newMobileNo);
```

ActorAssociationMgmtServiceImpl.java

```
@Override
public String saveOrUpdateActorByRenumeration(Double start,
Double end, String newAddress, Float newAge, Long newMobileNo) {
    //Prepare object
    Query query = new
Query().addCriteria(Criteria.where("renumeration").gte(start).andOperator(
Criteria.where("renumeration").lte(end)));
    //Prepare update object
    Update update = new Update().set("actorAddress",
newAddress).set("age", newAge).set("mobileNo", newMobileNo);
    //Invoke upsert method
    UpdateResult result = mongoTemplate.upsert(query, update,
Actor.class);
    return result.getModifiedCount()+" has modified and inserted
document is"+result.getUpsertedId();
}
```

MongoTemplateTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    System.out.println(actorAssociationMgmtService.saveOrUpdateActor
ByRenumeration(5000000.0, 10000000.0, "Hyd", 53.0f, 9876578933l));
}
```

Note:

- ✓ upsert(-) can perform single document updating or insertion if the Query object condition gives multiple documents, then it picks first document from that list to perform update operation.
- ✓ If Query object condition does not give any document or documents then it performs insert document operation using the data given in update object.
- ✓ upsert(-) method and save(-) is similar. The only difference is save (-) takes id value as the criteria value whereas upsert(-) take given Query object condition data value.

findAndRemove(Query, doc class):

- To perform single document removing operation by finding it through

Query object condition.

- Signature:

```
public <T> T findAndRemove(Query query, Class<T> entityClass;
```

[IActorAssociationMgmtService.java](#)

```
public String removeActorByAge(Float age);
```

[ActorAssociationMgmtServiceImpl.java](#)

```
@Override
public String removeActorByAge(Float age) {
    Query query = new
    Query().addCriteria(Criteria.where("age").is(age));
    Actor actor = mongoTemplate.findAndRemove(query,
    Actor.class);
    return actor.getActorId()+" document has removed.";
}
```

[MongoTemplateTestRunner.java](#)

```
@Override
public void run(String... args) throws Exception {

    System.out.println(actorAssociationMgmtService.removeActorByAge(
    40.0f));
}
```

Note: If Query object condition finds multiple documents, then it will delete the first document from the list. But `findAllAndRemove(-, -)` method will deleted all the fetched record.

[findAllAndRemove\(Query, doc class\):](#)

- To perform bulk, delete operation of the document based on given Query object condition.
- Signature

```
public <T> List<T> findAllAndRemove(Query query, Class<T> entityClass);
```

[IActorAssociationMgmtService.java](#)

```
public String removeAllActorsByAge(Float age);
```

ActorAssociationMgmtServiceImpl.java

```
@Override
public String removeAllActorsByAge(Float age) {
    Query query = new
Query().addCriteria(Criteria.where("age").is(age));
    List<Actor> actorList =
mongoTemplate.findAllAndRemove(query, Actor.class);
    return actorList.size()+" no. of documents has removed.";
}
```

MongoTemplateTestRunner.java

```
@Override
public void run(String... args) throws Exception {

    System.out.println(actorAssociationMgmtService.removeAllActorsBy
Age(40.0f));
}
```

Interacting with multiple DB s/w

- ✚ Concept: Interacting with multiple DB s/w from Spring application using Spring Data JAP or Creating multiple DataSource from Spring application using Spring Data JPA
- ✚ To interact with multiple DB s/w or to interact with different Logical DBs of same Bb s/w we need to use this concept. Here we cannot enjoy DataSource object that comes through AutoConfiguration.
- ✚ The AutoConfiguration based DataSource object always point single DB s/w or single logical DB. But we need pointing multiple DB s/w or multiple Logical DBs of same DB s/w.
- ✚ So, for that we need to go for manual configuration of Spring beans including DataSource using 100% code driven configurations or Java config configurations in Spring Boot application (indicates that we need to go for lots of manual configurations in Spring Boot app).

Use cases

- a. Transferring bank accounts details from one bank to another bank if one bank acquires another bank.
- b. Transfer Money operation between two banks.
- c. Save the product details in multiple DB s/w one for main use, another

- for backup.
- d. One app/ project saving different products with different DB s/w like customers info Oracle DB s/w and products or offers info in MySQL DB s/w.
 - e. Website displaying the info/ report by collecting from different DB s/w. and etc.

Basics recap

- If do not provide bean id for @Bean method-based Spring bean then method name itself will be taken as the default bean id.

```
@Bean
public DataSource createDS() {
    .....
    returns ds;
}
```

The default bean id is: createDS

- If we want to use IoC container supplied object in the @Bean method, it can be done in two ways,
 - a. Inject to @Configuration class and use in @Bean methods (the injected object is visible in all @Bean methods).

```
@Configuration
public class DBConfig {

    @Autowired
    private Environment env;

    @Bean
    public DataSource createDS1() {
        .....
        //use "env" object
    }

    @Bean
    public DataSource createDS2() {
        .....
        //use "env" object
    }
}
```

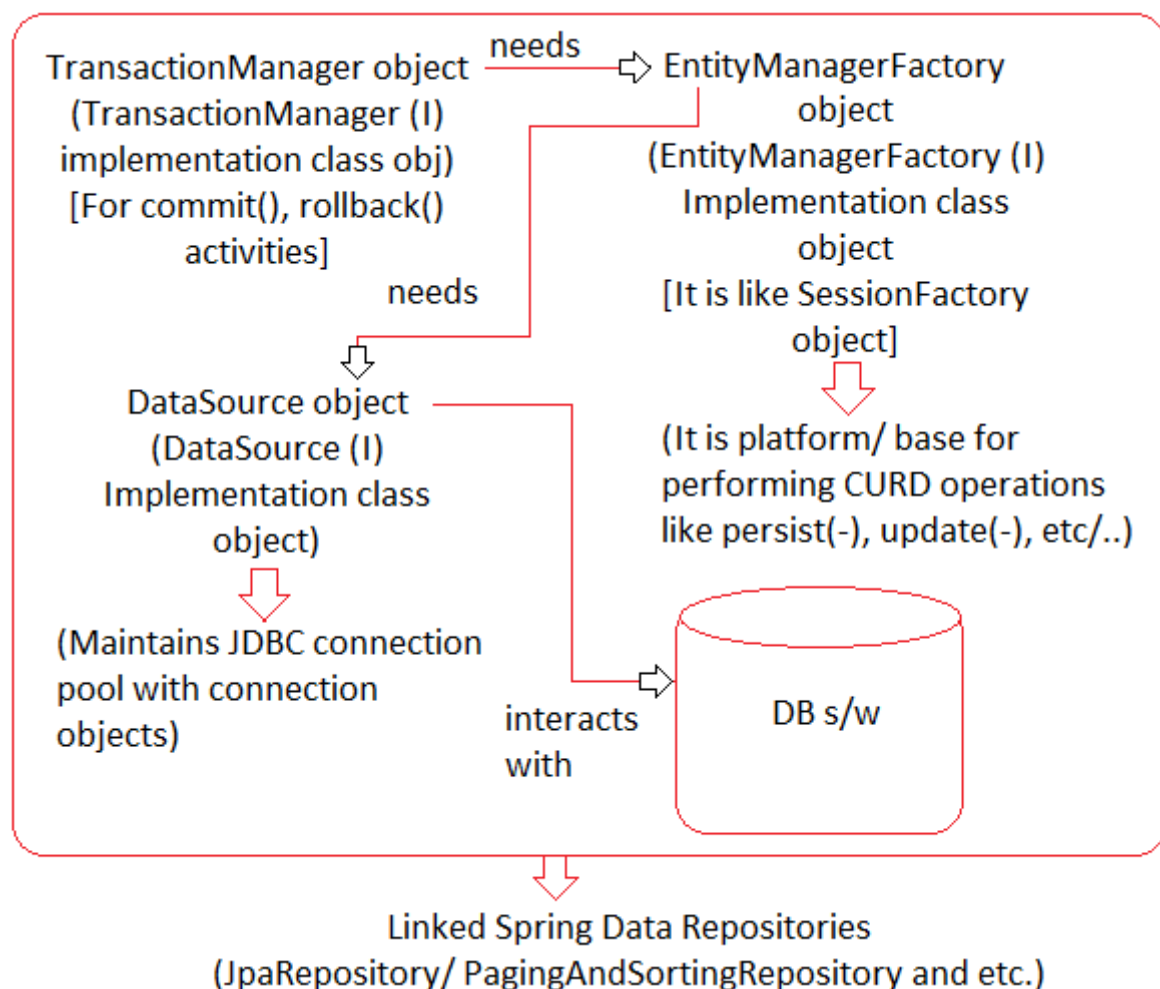
- b. Pass it as the parameter of @Bean method (here the Injected object can be made specific to one @Bean method).

```
@Configuration
public class DBConfig {

    @Bean
    public DataSource createDS1(Environment env) {
        .....
        //use env object
    }

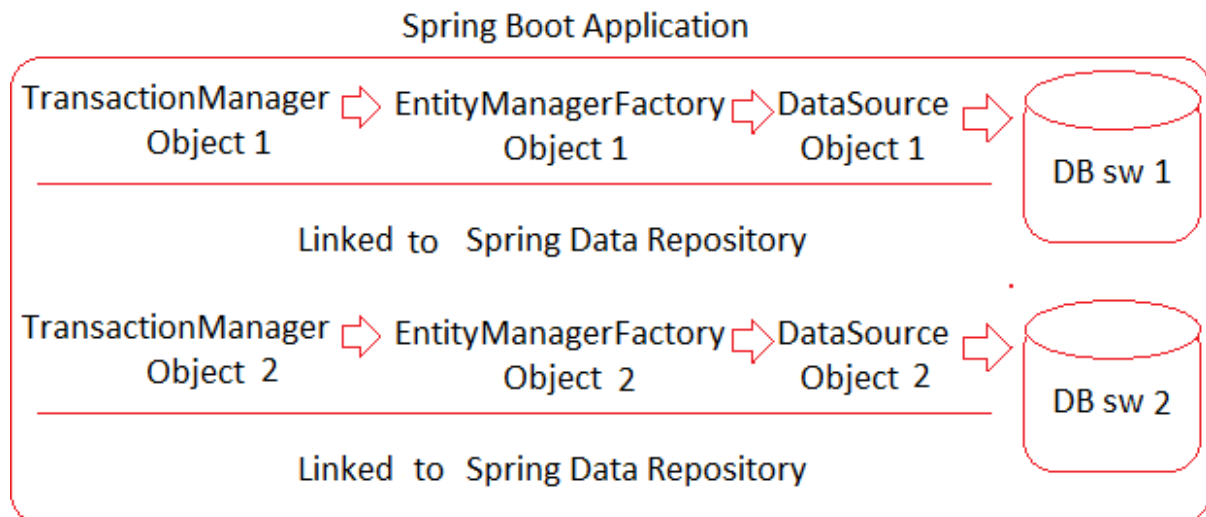
}
```

- ✚ When we get DataSource object through AutoConfiguration the following other object will be created internally while dealing Spring Data JPA



- While interacting with multiple DB s/w, stop using Autoconfiguration based DataSource, EntityManagerFactory, TransactionManager objects start creating them manually using Java config approach 100% code driven configuration with the support of @Bean methods.

To interact with two DB s/w or two logical DB of a DB s/w



- Like this we need 'n' sets of TransactionManager, EntityManagerFactory, DataSource objects to interact with "n" DB softwares.

Another way of creating DataSource object using @Bean method (without using Autoconfiguration DataSource object)

[application.properties](#)

oracle.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

oracle.datasource.jdbc-url=jdbc:oracle:thin:@localhost:1521:xe

oracle.datasource.username=system

oracle.datasource.password=manager

oracle is not fixed, remaining all are fixed in keys.

mysql.datasource.driver-class-name= com.mysql.cj.jdbc.Driver

mysql.datasource.jdbc-url=jdbc:mysql:localhost:3306/newdb

mysql.datasource.username=root

mysql.datasource.password=root

mysql is not fixed, remaining all are fixed in keys.

[Configuration class](#)

@Configuration

Public class DBConfig {

```

@Bean
@ConfigurationProperty(prefix="oracle.datasource")
public DataSource createOracleDS() {
    return DataSourceBuilder.create().build();
}

@Bean
@ConfigurationProperties(prefix="mysql.datasource")
public DataSource createMySQLDS() {
    return DataSourceBuilder.create().build();
}
}

```

Note: If we add spring-data-jpa-starter to the project it internally uses HikariCP DataSource by default.

Example application

Use case: Making Spring Boot Data JPA app inserting product info in Oracle DB table and offers info in MySQL DB table.

Step 1: Keep both MySQL and Oracle DB s/w ready.

Step 2: Create Spring Boot starter standalone project adding the following starters (jars).

```

X Lombok
X Spring Data JPA
X MySQL Driver
X Oracle Driver

```

Step 3: Write JDBC properties for both DB s/w having two different custom prefixes for keys in application.properties file (for the same keys suffixes are fixed according to DataSourceBuilder class).

Step 4: Create two Configuration classes for two different DB s/w having @Bean methods creating DataSource, EntityManagerFactory, TransactionManager objects

Note:

- ✓ FactoryBean is selfless bean i.e., when we ask container to give object of FactoryBean it does not give that object it gives the resultant object of FactoryBean.

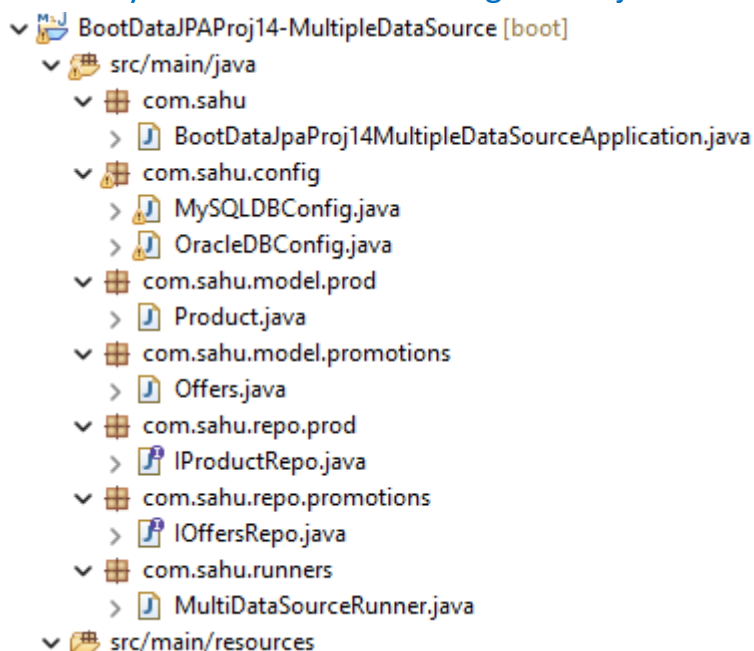
- ✓ If we FactoryBean as dependent to target bean then the FactoryBean object will not be injected the Resultant object given by FactoryBean will be injected to target bean.
- ✓ Generally, FactoryBean classes implements FactoryBean (I) and contains FactoryBean word at the end of the class name.
- ✓ LocalContainerEntityManagerFactoryBean is factory bean (selfless bean) that gives EntityManagerFactoryBean class object as the resultant.
- ✓ To create LocalContainerEntityManagerFactoryBean class object we need EntityManagerFactoryBuilder object which comes through autoconfiguration when we add "spring-data-jpa-starter".
- ✓ A bean that implements FactoryBean (I) cannot be used as a normal bean. A FactoryBean is defined in a bean style, but the object exposed for bean references (getObject()) is always the object that it creates.
- ✓ Based on @Primary kept in OracleDBConfig.java class, the spring-boot-jpa-starter related autoconfiguration takes the required DataSource, EntityManagerFactory and TransactionManager objects from OracleDBConfig.java class.

Step 5: Develop two separate model class in two different packages.

Step 6: Develop two separate Repository interfaces in two different packages.

Step 7: Develop Runner class to inject the repository objects and to test the application.

Directory Structure of BootMongoDBProj06-QueryMethods-Projection:



- application.properties
- > src/test/java
- > JRE System Library [JavaSE-11]
- > Maven Dependencies
- > src
- > target
- HELP.md
- mvnw
- mvnw.cmd
- pom.xml

- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- Use the following starters during project creation.

- X Lombok
- X Spring Data JPA
- X MySQL Driver
- X Oracle Driver

- Place the following code within their respective files.

application.properties

```
# jdbc properties for oracle
oracle.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
oracle.datasource.jdbc-url=jdbc:oracle:thin:@localhost:1521:xe
oracle.datasource.username=system
oracle.datasource.password=manager

# jdbc properties for mysql
mysql.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
mysql.datasource.url=jdbc:mysql:localhost:3306/ntspbms714db
mysql.datasource.username=root
mysql.datasource.password=root
```

OracleDBConfig.java

```
package com.sahu.config;

import java.util.HashMap;
import java.util.Map;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Qualifier;
```

```

import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import
org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import
org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.PlatformTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

```

```

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.sahu.repo.prod",
                      entityManagerFactoryRef =
"createOracleEntityManagerFactoryBean",
                      transactionManagerRef =
"createOracleEntityManagerFactoryBean")
public class OracleDBConfig {

```

```

    @Bean
    @ConfigurationProperties(prefix = "oracle.datasource")
    @Primary
    public DataSource createOracleDS() {
        return DataSourceBuilder.create().build();
    }

```

```

    @Bean
    @Primary
    public LocalContainerEntityManagerFactoryBean
createOracleEntityManagerFactoryBean(EntityManagerFactoryBuilder
builder) {

```

```

        //Create map object having hibernate properties
        Map<String, Object> props = new HashMap<>();

```

```

        props.put("hibernate.dialect",
"org.hibernate.dialect.Oracle10gDialect");
        props.put("hibernate.hbm2ddl.auto", "update");
        //Create and return LocalContainerEntityManagerFactoryBean
class object which means EntityManagerFactory ad the Spring bean
        return builder.dataSource(createOracleDS()) //DataSource
                .packages("com.sahu.model.prod") //Model class
package
                .properties(props) //Hibernate properties.
                .build();
    }

    @Bean
    @Primary
    public PlatformTransactionManager
createOracleTxMgmt(@Qualifier("createOracleEntityManagerFactoryBean"
) EntityManagerFactory factory) {
        return new JpaTransactionManager(factory);
    }
}

```

MySQLDBConfig.java

```

package com.sahu.config;

import java.util.HashMap;
import java.util.Map;

import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Qualifier;
import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import

```

```

org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import
org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.transaction.PlatformTransactionManager;
import
org.springframework.transaction.annotation.EnableTransactionManagement;

```

```

@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.nt.repo.promotions",
                        entityManagerFactoryRef =
"mysqlEMF",
                        transactionManagerRef =
"mysqlTxMgmr")
public class MySQLDBConfig {

```

```

    @Bean
    @ConfigurationProperties(prefix = "mysql.datasource")
    public DataSource createMySQLDs() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "mysqlEMF")
    public LocalContainerEntityManagerFactoryBean
createMySQLEntityManagerFactoryBean(
        EntityManagerFactoryBuilder builder) {
        // create Map object having hibernate properties
        Map<String, Object> props = new HashMap<>();
        props.put("hibernate.dialect",
"org.hibernate.dialect.MySQL8Dialect");
        props.put("hibernate.hbm2ddl.auto", "update");
        // create and return LocalContainerEntityManagerFactoryBean
class object which makes EntityManagerFactory as the Spring bean
        return builder.dataSource(createMySQLDs()) //DataSource
        .packages("com.nt.model.promotions") //Model
class package
        .properties(props) //Hibernate properties
        .build();

```

```

    }

    @Bean(name = "mysqlTxMgmr")
    public PlatformTransactionManager
createMysqlTxMgmr(@Qualifier("mysqlEMF") EntityManagerFactory
factory) {

        return new JpaTransactionManager(factory);

    }

}

```

Product.java

```

package com.sahu.model.prod;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Entity
@Table(name = "MDS_PRODUCT")
@Data
@NoArgsConstructor
@AllArgsConstructor
@RequiredArgsConstructor
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer pid;
    @Column(length = 20)
    @NonNull
    private String pname;
}

```

```
@NonNull
private Double qty;
@NonNull
private Double price;
}
```

Offers.java

```
package com.sahu.model.promotions;

import java.time.LocalDate;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Entity
@Table(name = "MDS_OFFERS")
@Data
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
public class Offers {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer offerId;
    @Column(length = 20)
    @NonNull
    private String offerName;
    @Column(length = 20)
    @NonNull
    private String offerCode;
```

```
@NonNull
private Double discountPercentage;
@NonNull
private LocalDate expireDate;
}
```

IProductRepo.java

```
package com.sahu.repo.prod;

import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.model.prod.Product;

public interface IProductRepo extends JpaRepository<Product, Integer> {

}
```

IOffersRepo.java

```
package com.sahu.repo.promotions;
IOffersRepo
import org.springframework.data.jpa.repository.JpaRepository;

import com.sahu.model.promotions.Offers;

public interface extends JpaRepository<Offers, Integer> {

}
```

MultiDataSourceRunner.java

```
package com.sahu.runners;

import java.time.LocalDate;
import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

import com.sahu.model.prod.Product;
```



```

import com.sahu.model.prod.Product;
import com.sahu.model.promotions.Offers;
import com.sahu.repo.prod.IProductRepo;
import com.sahu.repo.promotions.IOffersRepo;

@Component
public class MultiDataSourceRunner implements CommandLineRunner {

    @Autowired
    private IProductRepo productRepo;

    @Autowired
    private IOffersRepo offersRepo;

    @Override
    public void run(String... args) throws Exception {
        //save objects
        productRepo.saveAll(Arrays.asList(new Product("Table", 100.0,
6000.0),
new
Product("Chair", 10.0, 7000.0),
new
Product("Sofa", 11.0, 62000.0)));
        System.out.println("Products has saved");
        offersRepo.saveAll(Arrays.asList(new Offers("By 1 Get 1",
"B1G1", 100.0, LocalDate.of(2022, 10, 11)),
new
Offers("By 1 Get 2", "B1G2", 200.0, LocalDate.of(2022, 10, 11)),
new
Offers("By 2 Get 2", "B2G2", 100.0, LocalDate.of(2022, 10, 11))));
        System.out.println("Offers has saved");
        System.out.println("\n----- Display Records -----");
        productRepo.findAll().forEach(System.out::println);
        offersRepo.findAll().forEach(System.out::println);
    }
}

```

----- The END -----