spring

INDEX

Spring Data

Introduction

Two types of DB softwares:

1. SQL DB s/w (RDBMS DB s/w)

   e.g. Oracle, MySQL, PostgreSQL, DB/2, and etc.

   employee with 4 details
   employee with 10 details
   employee with 40 details

   - For this we need design DB table with 40 columns and most of time the unused columns memory is wasted.
   - Data with fixes structure then go for SQL DB softwares.

2. NoSQL DB s/w

   e.g. MongoDB, Cassandra, Kafka, Couchbase, Redis and etc.

   - Here No DB tables, rows and columns everything stored in form of documents So memory will be allocated only for given values, no wastage of memory.
     employee with 4 values - Doc with values
     employee with 10 values - Doc with values
   - Data with Dynamic structure whose data fields dynamically varies then go for No SQL.

Before arrival of Spring Data:

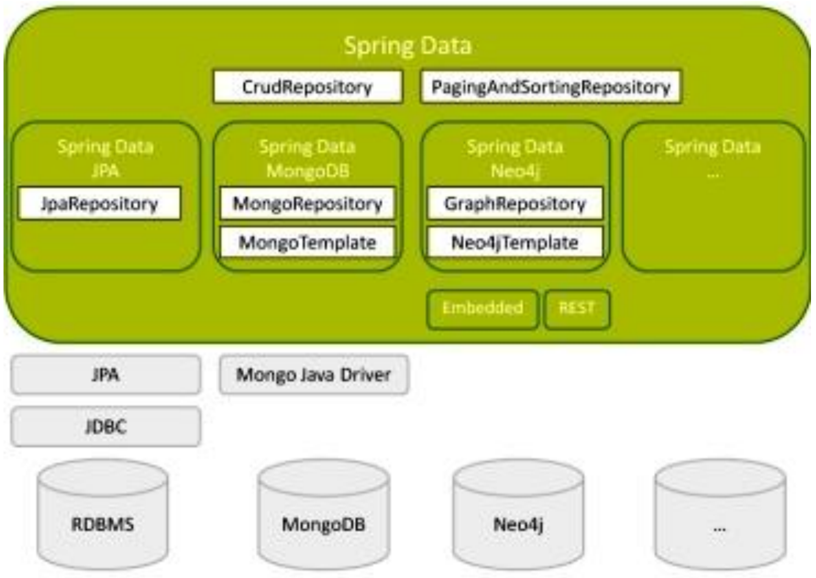
```
Java App -----------------------------------------> SQL DB s/w
             JDBC, Spring JDBC, ORM frameworks
                 like hibernate, Spring ORM

Java App -------------- MongoDB ----------------> MongoDB (NoSQL)
Java App -------------- Couchbase ---------------> Couchbase (NoSQL)
and etc.
```

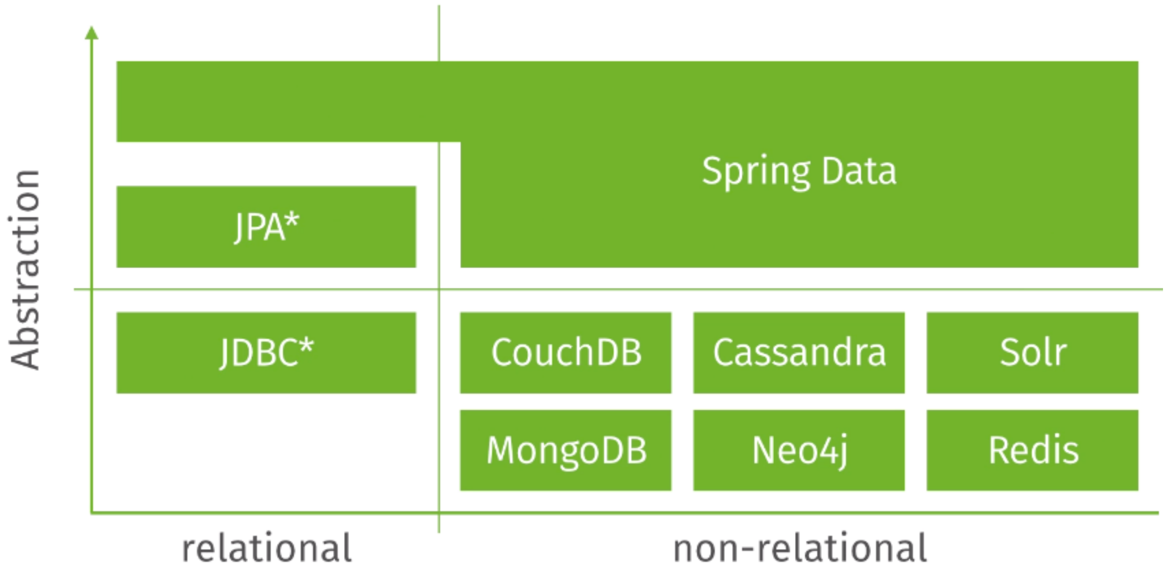
Note:

- ✓ No support from Spring to talk NoSQL DB softwares.
- ✓ No ORM frameworks to talk with NoSQL DB softwares.
- ✓ Before arrival of Spring data, we need to interact SQL, NoSQL DBS separately.
- ✓ Spring Data provides unified model programming to interact with both SQL and NoSQL DB s/w.
- ✓ Maintain data in the form of DB table memory is wasted.

Spring Data is given based on two design patterns:

1. Repository Pattern
2. Proxy Pattern





- Spring Data JDBC to talk with SQL DB s/w in JDBC style.
- Spring Data JPA to talk with SQL DB s/w in ORM style.
- Spring Data MongoDB to talk with NoSQL DB s/w (MongoDB).
- Spring Data CouchDB to talk with NoSQL DB s/w (CouchDB).
  and etc.

- Spring data is extension module for spring framework developed based on spring to interact both SQL and NoSQL DB softwares in a unified model.

Spring modules:

Core
AOP
JDBC/DAO
ORM
Web MVC
Context

Web
TxMgmt
Mail
and etc.

Spring Extension

Data
Batch
Social
Security
and etc.

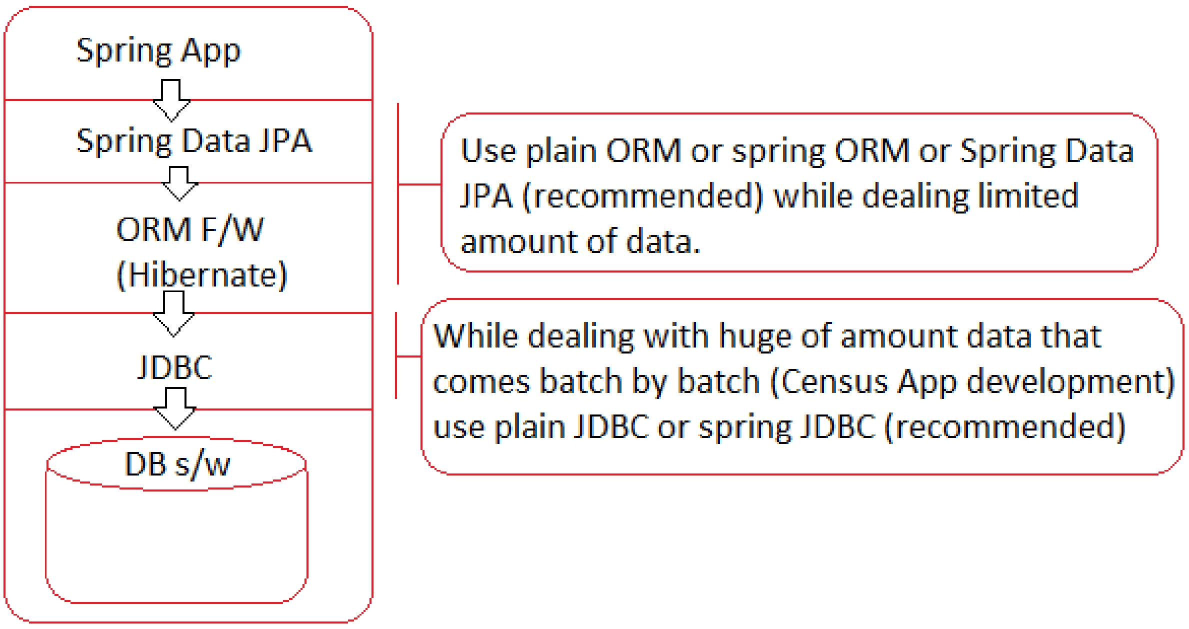
Spring Data features:

- Powerful repository and custom object-mapping abstractions.
- Dynamic query derivation from repository method names.
- Implementation domain base classes providing basic properties.
- Support for transparent auditing (created, last changed).
- Possibility to integrate custom repository code.
- Easy Spring integration via Java Config and custom XML namespaces.
- Advanced integration with Spring MVC controllers.
- Experimental support for cross-store persistence.

We cover:

Spring Data JPA (for SQL DB s/w)
Spring Data MongoDB (for NoSQL DB s/w)

Spring Data JPA



Use plain ORM or spring ORM or Spring Data JPA (recommended) while dealing limited amount of data.

While dealing with huge of amount data that comes batch by batch (Census App development) use plain JDBC or spring JDBC (recommended)

- To learn and use Spring Data JPA we need strong plain hibernate knowledge because it internally uses hibernate.

- No DAO classes while working with Spring Data modules because work with Repository Interfaces by specifying Entity class name and its ID Property type. For that interface Spring Data internally generates Proxy class having persistence logic i.e. we do not write Persistence logic we work with Dynamically generated Persistence logic of the Proxy class.

```
Student DB table in Oracle
-----------------------------------------
Student (db table)
        I---> sno (pk) (n)
        I---> sname (vc2)
        I---> sadd (vc2)
        I---> avg (float)
```

```
Entity class
----------------------
@Entity
public class Student implements Serializable {
        private Integer sno;
        private String sname;
        private String sadd;
        private float avg;
        //getters & getters
        .............
}
```

CurdRepository - So many method declrations for CURD operations
Student - Entity class name
<Integer> - ID Property type

```
Repository interface
-----------------------------------
package com.nt.repo;
public interface StudentRepo extends CrudRepository <Student, Integer> {

}
```

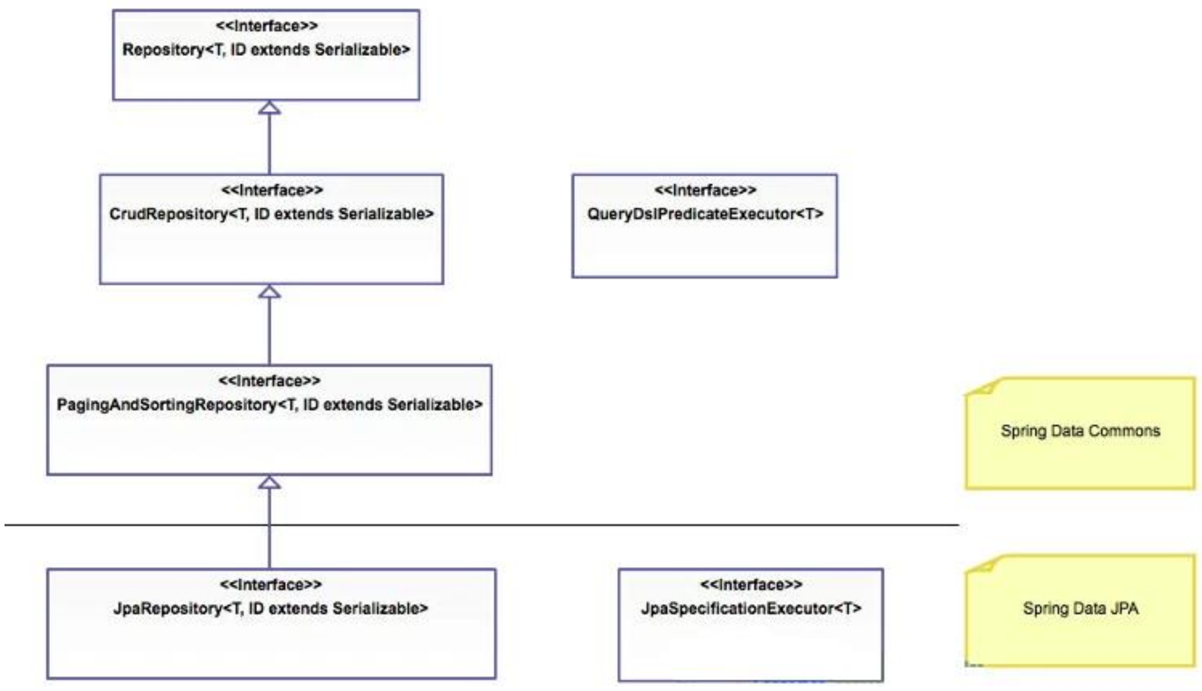
Should extends Repository(I) directory or indirectly

- The IOC container creates InMemory Proxy class at runtime implementing StudentRepo interface and provides implementation for the all methods that are inherited from CrudRepository Interface.

Before spring Data using Spring ORM or Plain ORM:
100 DB tables
    100 DAO interfaces
    100 DAO implementation classes having curd operation persistence logics methods (10 methods in each class) (bit heavy)

With Spring Data JAP:
100 DB tables
    100 repository interfaces (programmer part is over)
    Here 100 implementation classes are generated internally using Proxy DP having Persistence logic So here burden on the programmer is reduced.

Hierarchy of Spring Data:



<<Interface>>
Repository<T, ID extends Serializable>
<<Interface>>
CrudRepository<T, ID extends Serializable>
<<Interface>>
QueryDslPredicateExecutor<T>
<<Interface>>
PagingAndSortingRepository<T, ID extends Serializable>
Spring Data Commons
<<Interface>>
JpaRepository<T, ID extends Serializable>
<<Interface>>
JpaSpecificationExecutor<T>
Spring Data JPA

Using Spring Data Repository in Service class:

```
public interface StudentMgmtService {
        public String register (StudentDTO dto);
}

@Service("studService")
public class StudentMgmtServiceImpl implements StudentMgmtService{

        @Autowired
        private StudentRepo studRepo; //injects Generated proxy class obj

        public String register (StudentDTO dto) {
                //convert dto to entity
                …………
                //use repo
                Student stud=studRepo.save(entity);
                If (stud! = null)
                        return " student registered";
                else
                        return "student not registered";
        }
}
```

Difference between DAO Design Pattern and Repository Design Pattern:

DAO Pattern	Repository Pattern
a. Contains DAO (I) and DAO Implementation class developed by Programmer.	a. Contains Repository (I) given by the programmer and implementation class for Interface given underlying framework or container.
b. Does not use Proxy Pattern at all.	b. Repository(I) implementation class will come based on Proxy Design Pattern.
c. Writing Persistence logics is responsibility of the Programmer.	c. Here the underlying F/w or Container will take care.
d. In one DAO we can write persistence logics of multiple DB table.	d. One Repository(I) we can deal with only one DB table or Entity class.
e. In DAO we can write both JDBC style O-R mapping style persistence logic.	e. Here only O-R mapping style persistence logic possible.
f. DAO is DB table centric.	f. Repository is Entity class centric.

Note: Both are given to separate persistence logic from other logics of the Application.

In Spring Data JPA we can develop persistence logics by using:

a. Using the inherited methods of Repository Interface to the dynamically generated Proxy class (for CURD operations).
b. Using explicitly declared findXxx methods in our Repository Interface (select operations).
c. Using @Query(select), @Query with @Modify (non-select) methods based Custom HQL/JPQL queries Native SQL queries

Note: We can call Stored procedures using Spring Data JPA.

Different approaches of developing spring data Applications:

a. Using xml driven configuration
b. Using annotation driven configuration
c. Using 100% Code/ Java Config configuration
d. Using Spring Boot configuration

First application Using Spring Boot configuration CrudRepository

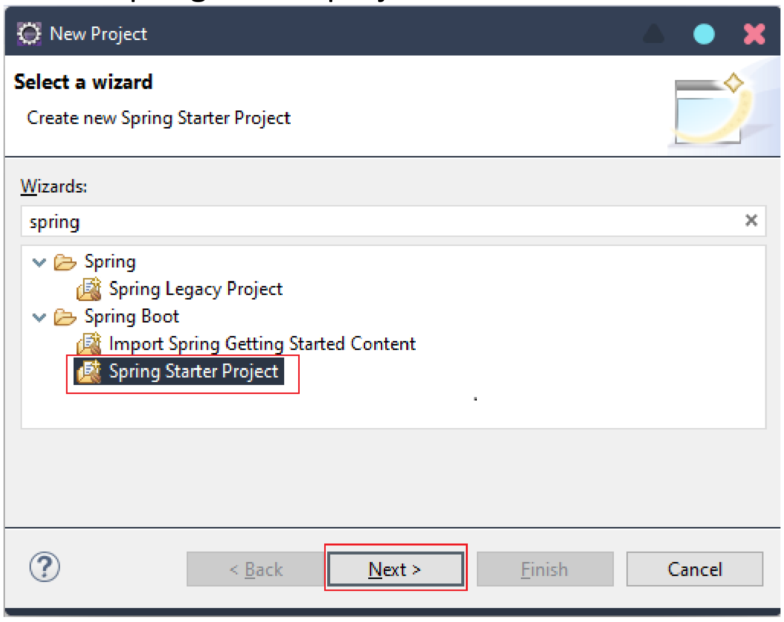
```
                  DTO                     BO/ Entity
Clint App ----------------> Service class --------------> Repository ----------> DB s/w
                            (b logic)                     (Persistence logic)

                  (Controller is ignored here)
```

Resources in Application Development:

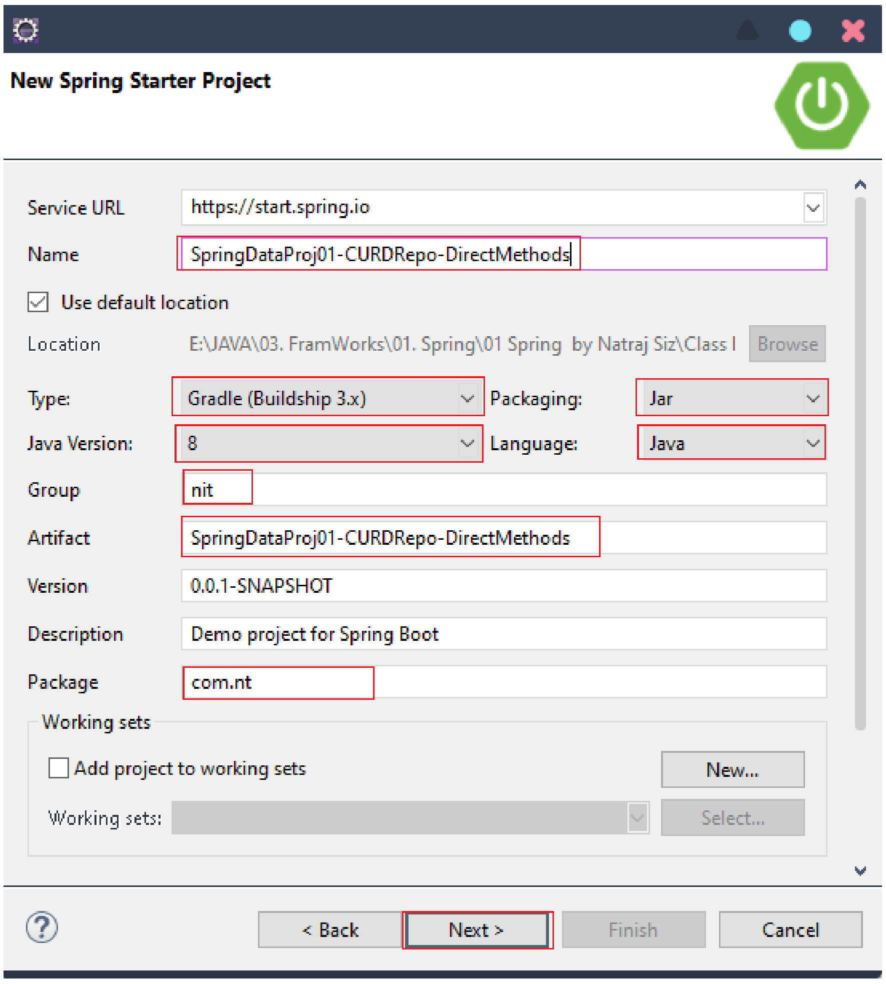
a. Customer.java Entity class @Entity
b. CustomerDTO.java DTO class
c. CustomerRepo.java Repository Interface
d. CustomerMgmtService.java Service Interface
e. CustomerMgmtSErvicelmpl.java Service Implementation class @Service
f. Stater class /Main class with @SpringBootApplicaiton
g. application.properties (opt)
   DataSource properties (driver class name, URL, username, password)
   ORM/JPA properties (dialect, hbm2ddl.auto, show_sql, format_sql)
   (opt) (opt)

Procedure to develop Spring Data application using Spring boot in eclipse:

Step #1: Create a Sprig starter project, Click on File then New then other, search Spring starter project then choose that click on Next.

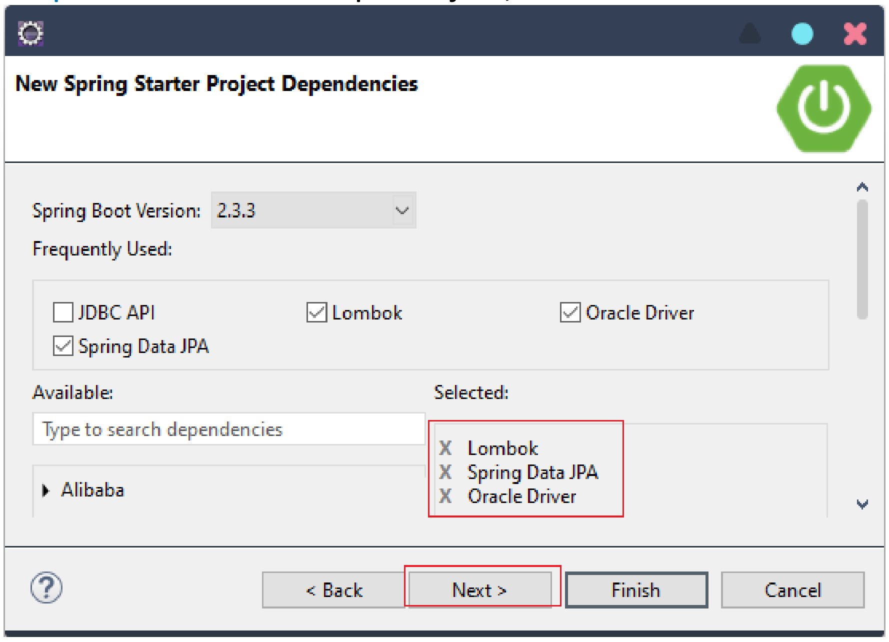


Step #2: Give the following details, then click on Next.



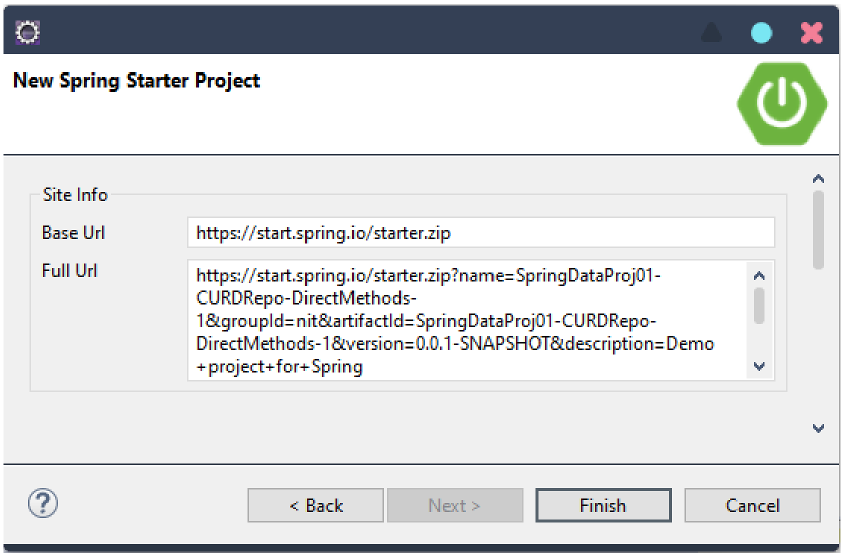
Service URL: https://start.spring.io
Name: SpringDataProj01-CURDRepo-DirectMethods
[✓] Use default location
Location: E:\JAVA\03. FramWorks\01. Spring\01 Spring by Natraj Siz\Class I
Type: Gradle (Buildship 3.x)
Packaging: Jar
Java Version: 8
Language: Java
Group: nit
Artifact: SpringDataProj01-CURDRepo-DirectMethods
Version: 0.0.1-SNAPSHOT
Description: Demo project for Spring Boot
Package: com.nt
Working sets
[ ] Add project to working sets
Working sets:

Step #3: choose the required jars, then click on next.



Spring Boot Version: 2.3.3
Frequently Used:
[ ] JDBC API
[✓] Lombok
[✓] Oracle Driver
[✓] Spring Data JPA
Available:
Type to search dependencies
Alibaba
Selected:
X Lombok
X Spring Data JPA
X Oracle Driver

Step #4: Then click on Finish


New Spring Starter Project

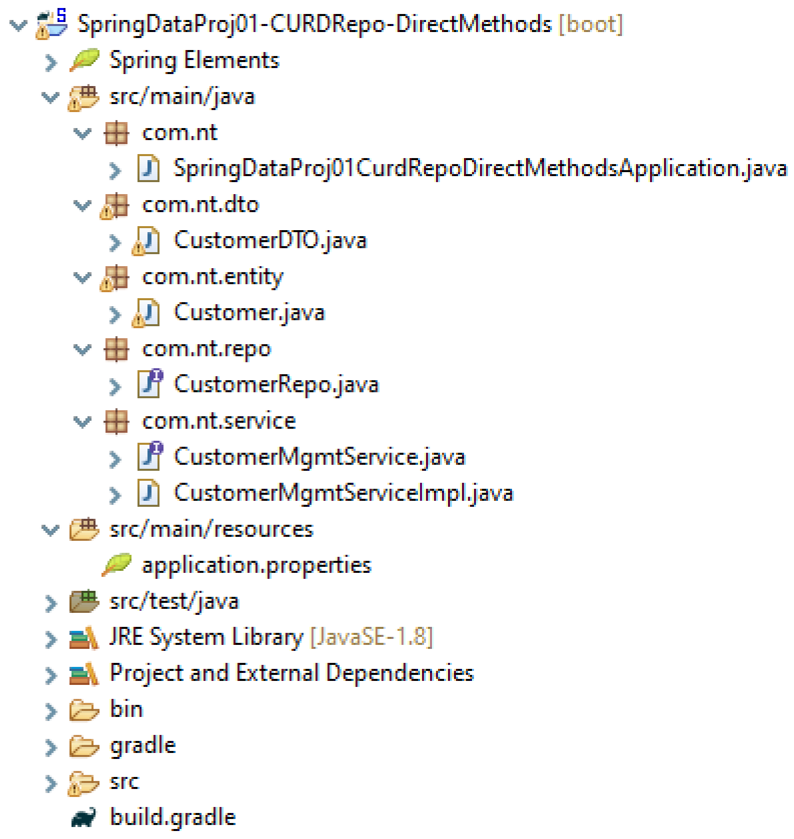
Site Info

Base Url: https://start.spring.io/starter.zip

Full Url: https://start.spring.io/starter.zip?name=SpringDataProj01-CURDRepo-DirectMethods-1&groupId=nit&artifactId=SpringDataProj01-CURDRepo-DirectMethods-1&version=0.0.1-SNAPSHOT&description=Demo+project+for+Spring

Now your Spring Boot project is ready.

Directory Structure of SpringDataProj01-CURDRepo-DirectMethods:



```
SpringDataProj01-CURDRepo-DirectMethods [boot]
  Spring Elements
  src/main/java
    com.nt
      SpringDataProj01CurdRepoDirectMethodsApplication.java
    com.nt.dto
      CustomerDTO.java
    com.nt.entity
      Customer.java
    com.nt.repo
      CustomerRepo.java
    com.nt.service
      CustomerMgmtService.java
      CustomerMgmtServiceImpl.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure using Spring Starter Project option and package and classes also.
- Many jars dependencies will came automatically in build.gradle because while developing Spring Starter Project we choose some jars and other required jar we will add in dependencies { } enclosure along with previous jars.

- Then use the following code with in their respective file.

application.properties

```
#DataSource configuration details
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
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100

#JPA/ Hibernate properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDialect
```

Customer.java

```
package com.nt.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 org.hibernate.annotations.Type;

import lombok.Data;

@Entity
@Data
public class Customer implements Serializable {

	@Id
	@Type(type = "int")
	@GeneratedValue(strategy = GenerationType.AUTO )
	private Integer cno;

	@Column(length = 15)
	@Type(type = "string")
	private String cname;

	@Column(length = 15)
	@Type(type = "string")
	private String cadd;

	@Type(type = "double")
	private Double billAmount;

}
```

CustomerDTO.java

```
package com.nt.dto;

import java.io.Serializable;

import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;
@Data
@RequiredArgsConstructor
@NoArgsConstructor
public class CustomerDTO implements Serializable {
	@NonNull
	private Integer cno;
	private String cname;
	private String cadd;
	private Double billAmount;
}
```

CustomerRepo.java

```
package com.nt.repo;

import org.springframework.data.repository.CrudRepository;
import com.nt.entity.Customer;

public interface CustomerRepo extends CrudRepository<Customer,
Integer> {
}
```

CustomerMgmtService.java

```
package com.nt.service;

import com.nt.dto.CustomerDTO;

public interface CustomerMgmtService {
	public String registerCustomer(CustomerDTO dto);
}
```

CustomerMgmtServiceImpl.java

```
package com.nt.service;

import java.util.Arrays;

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

import com.nt.dto.CustomerDTO;
import com.nt.entity.Customer;
import com.nt.repo.CustomerRepo;

@Service("custService")
public class CustomerMgmtServiceImpl implements CustomerMgmtService
{
	@Autowired
	private CustomerRepo custRepo;

	@Override
	public String registerCustomer(CustomerDTO dto) {
		System.out.println(custRepo.getClass()+" :
"+Arrays.toString(custRepo.getClass().getInterfaces()));
		Customer cust = null;
```

```
                    //convert dto to entity
                    cust = new Customer();
                    BeanUtils.copyProperties(dto, cust);
                    //use repo
                    cust = custRepo.save(cust);
                    return cust!=null?"Object is saved with id :
"+cust.getCno():"Object is not saved";
          }

}
```

SpringDataProj01CurdRepoDirectMethodsApplication.java

```
package com.nt;

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.nt.dto.CustomerDTO;
import com.nt.service.CustomerMgmtService;

@SpringBootApplication
public class SpringDataProj01CurdRepoDirectMethodsApplication {

          public static void main(String[] args) {
                    ApplicationContext ctx = null;
                    CustomerMgmtService service = null;
                    CustomerDTO custDTO = null;
                    //get AC IoC container
                    ctx =
SpringApplication.run(SpringDataProj01CurdRepoDirectMethodsApplication
.class, args);
                    //get service class object
                    service = ctx.getBean("custService",
CustomerMgmtService.class);
                    //invoke method
                    try {
```

```
                                //prepare DTO object
                                custDTO = new CustomerDTO();
                                custDTO.setCname("Hari");
                                custDTO.setCadd("hyd");
                                custDTO.setBillAmount(1000.0);
                                System.out.println(service.registerCustomer(custDTO));

                        } catch (DataAccessException dae) {
                                dae.printStackTrace();
                        }

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

        }
```

Note:

- ✓ <S extends T> S save (S entity); this takes entity object and returns another object entity class as saved object.
- ✓ save (-) given spring Data JPA performs save object operation if record is not available having given id property value in PK column otherwise it will perform update operations (when no Generator is configured, if generator is configured always performs save object operation).
- ✓ Generally. in Spring Data JPA we use @Qyery + @Modify annotation methods for update operations because we always take id property with Generator configuration.
- ✓ There are no merge(-), saveOrUpdate(-), update(-) methods in Spring Data JPA. We can bring these effects using with @Query+@Modify annotation methods.
- ✓ <S extends T> Iterable<S> saveAll(Iterable<S> entities)
  This method is very useful in batch insertion of records use cases like group ticket reservation, group ticket booking and etc.

Q. In batch process if get any exception comes for any one of object is it save remaining or not saved?
Ans. No, Transaction management is enabled by default.

Different methods for delete object operation in CrudRepository:

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

Always prefer load and delete or check delete operation to return success or failure message from service class method.

- Place the following code in their respective file for the following operations.

CustomerMgmtService.java

```
	public String registerGroupCustomer(List<CustomerDTO> listDTO);
	public String removeCustomerByID(int id);
	public String removeGivenCustomers(Iterable<CustomerDTO> itDTO);
	public Long fetchCustomerCount();
	public Iterable<CustomerDTO> fetchAllCustomer();
```

CustomerMgmtServiceImpl.java

```
	@Override
	public String registerGroupCustomer(List<CustomerDTO> listDTO) {
		List<Customer> listEntities = new ArrayList();
		List<Customer> listEntities1 = null;
		String ids = new String();
		//convert dto to entity
		listDTO.forEach(dto -> {
			Customer cust = new Customer();
			BeanUtils.copyProperties(dto, cust);
			listEntities.add(cust);
		});
		//user repo
		listEntities1 = (List<Customer>) custRepo.saveAll(listEntities);
		for (Customer customer : listEntities1) {
			ids = ids+", "+customer.getCno();
		}
		return listEntities1!=null?"Batch records are inserted with ids
"+ids:"Records are not inserted";
	}
```

```
        @Override
        public String removeCustomerByID(int id) {
                boolean flag = false;
                //use repo
                flag = custRepo.existsById(id);
                if (flag)
                        custRepo.deleteById(id);
                return flag?"Given Record has deleted":"Record not exist";
        }

        @Override
        public String removeGivenCustomers(Iterable<CustomerDTO> itDTO)
{
                Iterable<Customer> itEntity = new ArrayList();
                //use repos
                itDTO.forEach(dto-> {
                        Customer cust = new Customer();
                        BeanUtils.copyProperties(dto, cust);
                        ((List<Customer>) itEntity).add(cust);
                });
                //use repo
                custRepo.deleteAll(itEntity);
                return "Bulk records are deleted";
        }

        @Override
        public Long fetchCustomerCount() {
                //use repo
                return custRepo.count();
        }

        @Override
        public Iterable<CustomerDTO> fetchAllCustomer() {
                Iterable<Customer> itEntities = null;
                Iterable<CustomerDTO> itDTO = new ArrayList();
                //use repo
                itEntities = custRepo.findAll();
                //convet entity to dto
                itEntities.forEach(entity->{
                        CustomerDTO dto = new CustomerDTO();
```

```
                    BeanUtils.copyProperties(entity, dto);
                    ((List<CustomerDTO>) itDTO).add(dto);
            });
            return itDTO;
    }
```

SpringDataProj01CurdRepoDirectMethodsApplication.java

```
            System.out.println("-----------------------------");
            custDTO = new CustomerDTO();
            custDTO.setCname("Hari"); custDTO.setCadd("hyd");
custDTO.setBillAmount(1000.0);
            custDTO1 = new CustomerDTO();
            custDTO1.setCname("Ravi"); custDTO1.setCadd("hyd");
custDTO1.setBillAmount(1000.0);
            custDTO2 = new CustomerDTO();
            custDTO2.setCname("javi"); custDTO2.setCadd("hyd");
custDTO2.setBillAmount(1000.0);
            try {
        System.out.println(service.registerGroupCustomer(Arrays.asList(cust
DTO, custDTO1, custDTO2)));
            } catch (DataAccessException dae) {
                    dae.printStackTrace();
            }
            System.out.println("-----------------");
            System.out.println(service.removeCustomerByID(27));
            System.out.println("------------------");
        System.out.println(service.removeGivenCustomers(Arrays.asList(new
CustomerDTO(30), new CustomerDTO(24))));
            System.out.println("------------------");
            System.out.println(service.fetchCustomerCount());
            System.out.println("------------------");
            System.out.println(service.fetchAllCustomer());
```

- Execute one by one method by commenting other methods.

To check whether record is record is available or not:

    boolean existsById(ID id)

To perform save object operation:

```
<S extends T> S save(S entity)
<S extends T> Iterable<S> saveAll(Iterable<S> entities)
```

To select operations:

```
Iterable<T> findAll()
Iterable<T> findAllById(Iterable<ID> ids)
Optional<T> findById(ID id)
```

Note: Iterable is supper interface for all collections from java 5 (earlier it was Collection(I)).

```
Optional<Customer> opt;
Customer cust=null;
CustomerDTO dto=null;
Customer cust=custRepo.findById(no);
if(cust!=null){
        CustomerDTO dto=new CustomerDTO();
        BeanUtils.copyProperties(cust,dto);
}
```

CHILD CODE

```
Optional<Customer> opt=custRepo.findById(id);
if(opt.isPresent()) {
        Customer cust=opt.get();
        CustomerDTO dto=new CustomerDTO();
        BeanUtils.copyProperties(cust, dto);
}
```

ADULT CODE

Optional API is introduced from Java 8 to checks whether received object is null or not and to perform various when present when not present. This basically, avoid NullPointerException from java code.

CustomerMgmtService.java

```
        public Optional<CustomerDTO> fetchCustomerById(int id);
```

CustomerMgmtServiceImpl.java

```
        @Override
        public Optional<CustomerDTO> fetchCustomerById(int id) {
                Optional<Customer> optEntity;
                Optional<CustomerDTO> optDTO = null;
                //use repo
                optEntity = custRepo.findById(id);

                if (!optEntity.isEmpty()) {
                        optDTO = Optional.of(new CustomerDTO());
                        BeanUtils.copyProperties(optEntity.get(), optDTO.get());
                } else  {
                        optDTO = optDTO.empty();
```

```
            }
            return optDTO;
      }
```

SpringDataProj01CurdRepoDirectMethodsApplication.java

```
System.out.println("-------------------------------");
            try {
                  Optional<CustomerDTO> opt =
service.fetchCustomerById(31);
                  if (opt.isPresent() && !opt.isEmpty())
                        System.out.println(opt.get());
                  else
                        System.out.println("Record is not there");
            } catch (DataAccessException dae) {
                  dae.printStackTrace();
            }
```

PagingAndSortingRepository

- Sub Interface of CrudRepository (I)
- Super interface of JpaRepository(I)
- Given for sorting (ASE/ DESC) and pagination (displaying records page by page report generation) activities.

methods:

Iterable<T> findAll(Sort sort) // Only Sorting
Page<T> findAll(Pageable pageable) // for pagination with/ with our sorting

Iterable<T> findAll(Sort sort):

- itEntities=custRepo.findAll(Sort.by(asc?Direction.ASC:Direction. DESC, property));
- itEntities=custRepo.findAll(Sort.by(asc?Direction.ASC:Direction.DESC, properties));

Directory Structure of SpringDataProj02-PASRepo-DirectMethods:

- Copy paste the SpringDataProj01-CRUDRepo-DirectMethods application and change rootProject.name to SpringDataProj02-PASRepo-DirectMethods in settings.gradle file.
- Need not to add any new file same SpringDataProj01-CRUDRepo-DirectMethods. Use PagingAndSortingRepository here.

- Change the SpringBootApplication class name to SpringDataProj02PASRepoDirectMethodsApplication
- Add the following code in their respective files.

CustomerMgmtService.java

```
package com.nt.service;

import com.nt.dto.CustomerDTO;

public interface CustomerMgmtService {
	public Iterable<CustomerDTO>
fetchAllRecordsSortByProprty(boolean asc, String propertie);
	public Iterable<CustomerDTO>
fetchAllRecordsSortByProprties(boolean asc, String... properties);
}
```

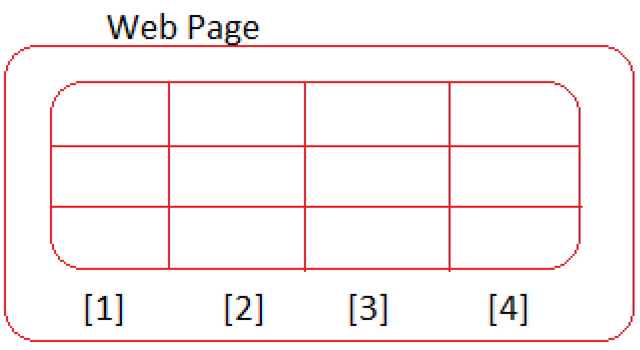
CustomerMgmtServiceImpl.java

```
package com.nt.service;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.BeanUtils;
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.nt.dto.CustomerDTO;
import com.nt.entity.Customer;
import com.nt.repo.CustomerRepo;

@Service("custService")
public class CustomerMgmtServiceImpl implements CustomerMgmtService
{
	@Autowired
	private CustomerRepo custRepo;

	@Override
	public Iterable<CustomerDTO>
fetchAllRecordsSortByProprty(boolean asc, String propertie) {
		Iterable<Customer> itEntity = null;
		Iterable<CustomerDTO> itDTO = new ArrayList();
		//use repo
		itEntity =
custRepo.findAll(Sort.by(asc?Direction.ASC:Direction.DESC, propertie));
		//convert itEnitiy to itDTO
		itEntity.forEach(entity-> {
			CustomerDTO dto = new CustomerDTO();
			BeanUtils.copyProperties(entity, dto);
			((List) itDTO).add(dto);
		});
		return itDTO;
	}

	@Override
	public Iterable<CustomerDTO>
fetchAllRecordsSortByProprties(boolean asc, String... properties) {
		Iterable<Customer> itEntity = null;
		Iterable<CustomerDTO> itDTO = new ArrayList();
		//use repo
		itEntity =
custRepo.findAll(Sort.by(asc?Direction.ASC:Direction.DESC, properties));
		//convert itEnitiy to itDTO
		itEntity.forEach(entity-> {
			CustomerDTO dto = new CustomerDTO();
			BeanUtils.copyProperties(entity, dto);
			((List) itDTO).add(dto);
		});
		return itDTO;
	}

}
```

SpringDataProj02PASRepoDirectMethodsApplication

```
package com.nt;

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

import com.nt.service.CustomerMgmtService;

@SpringBootApplication
public class SpringDataProj02PASRepoDirectMethodsApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		CustomerMgmtService service = null;
		//get AC IoC container
		ctx =
SpringApplication.run(SpringDataProj02PASRepoDirectMethodsApplication.
class, args);
		//get service class object
		service = ctx.getBean("custService",
CustomerMgmtService.class);
		//invoke method
		//service.fetchAllRecordsSortByProprty(true,
"cname").forEach(System.out::println);
		System.out.println("----------------------------");
		service.fetchAllRecordsSortByProprties(true, "cname",
"cadd").forEach(System.out::println);

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

}
```

Page<T> findAll(Pageable pageable):



Instead of displaying all records in a single page, display them page by page, that to based given Page No and Pagesize.

- We need inputs (Page No - 0 based, Pagesize - 1 based) in the Pageable object.

  // pageNo, pageSize PageRequest class implements Pageable
  Pageable pageable=PageRequest.of(3, 5);

- We get output from findAll(Pageable obj) either in the form Page object/ Slice obj having List<T> (list entities) other details like page number, total number of pages , count of records, next page Info and etc.

  Page<Customer> page=custRepo.findAll(pageable);

CustomerMgmtService.java

```
	public Iterable<CustomerDTO> fetchRecordsByPageNoAndSize(int pageNo, int pageSize);
}
```

CustomerMgmtServiceImpl.java

```
	@Override
	public Iterable<CustomerDTO> fetchRecordsByPageNoAndSize(int pageNo, int pageSize) {
		Pageable pageable = null;
		Page<Customer> page = null;
		Iterable<Customer> itEntity = null;
		Iterable<CustomerDTO> itDTO = new ArrayList();
		// Create Pageable object
		pageable = PageRequest.of(pageNo, pageSize);
		//get Page object
		page = custRepo.findAll(pageable);
		//convert page object into DTO
		itEntity = page.getContent();
		//convert itEnitiy to itDTO
		itEntity.forEach(entity-> {
			CustomerDTO dto = new CustomerDTO();
			BeanUtils.copyProperties(entity, dto);
			((List) itDTO).add(dto);
		});
		return itDTO;
	}
```

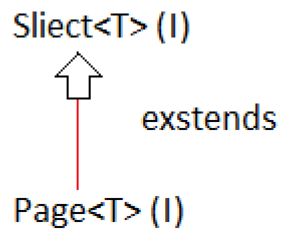
SpringDataProj02PASRepoDirectMethodsApplication

```
                    System.out.println("--------------------------");
                    service.fetchRecordsByPageNoAndSize(1,
3).forEach(System.out::println);
```

Q. What is the difference between Slice and Page object in PagingAndSortingRepository?

Ans. Slice obj holds info about the current report page data information not about other report pages data. Using this object, we can get info about only current report page data by calling various getXxx() but we cannot get total number of records total number of pages information.

- Page obj holds info about the all the report pages data „ including current port pages data. Using this object, we can get info about current report page data by calling various getXxx() and we can also get total number of records, total number of pages information



Note: Page object is bit heavy object compare to Slice object because it contains multiple List objects/ Slice objects internally.

```
                    Page<Customer> page = custRepo.findAll(pageable);
                    Slice<Customer> slice = custRepo.findAll(pageable);
                    System.out.println(page.getNumber()+"
"+page.getNumberOfElements()+" "+page.hasContent()+"
"+page.isEmpty()+" "+page.isFirst()+" "+page.getTotalPages()+"
"+page.getTotalElements());
                    System.out.println(slice.getNumber()+"
"+slice.getNumberOfElements()+" "+slice.hasContent()+" "+slice.isEmpty()+"
"+slice.isFirst());
                    //convert page object into DTO
                    itEntity = page.getContent();
                    itEntity = slice.getContent();
```

Real Pagination Example – place the following code in their respective file.

CustomMgmtService.java

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

CustomerMgmtServiceImpl.java

```
	@Override
	public void fetchRecordByPagination(int pageSize) {
		long recordsCount = 0;
		long pagesCount = 0;
		Pageable pageable = null;
		Page<Customer> page = null;
		//get total no. of record
		recordsCount = custRepo.count();
		pagesCount = recordsCount/pageSize;
		pagesCount = 
recordsCount%pageSize==0?pagesCount:pagesCount++;
		//display records through pagenation
		for (int i = 0; i < pagesCount; i++) {
			pageable = PageRequest.of(i, pageSize);
			page = custRepo.findAll(pageable);
			page.getContent().forEach(System.out::println);
			System.out.println("Page "+(i+1)+" of 
"+page.getTotalPages());
		}
	}
```

SpringDataProj02PASRepoDirectMethodsApplication

```
		ySystem.out.println("---------------------------");
		service.fetchRecordByPageination(3);
```

JpaRepository

- Most of the methods inherited from CrudRepository, PagingAndSortingRepository and redefined as per JPA specification. But also having some direct methods
  <S extends T> List<S> findAll(Example<S> example)
  <S extends T> List<S> findAll(Example<S> example, Sort sort)
  void deleteAllInBatch() - Deletes batch of records by generating single delete query
  void deleteInBatch(Iterable<T> entities)
  T getOne(ID id)
  void flush() - without waiting for TxManager to commit the Tx, it will write changes to Db s/w.

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

- Example obj wrapper object around given Object. It given in hibernate API having similar behaviour of Optional (java 8).
- <S extends T> - Collects entity object from wrapper Example object and uses all non-null value properties int the SQL query preparation having and clause (prefer taking wrapper data types).

Directory Structure of SpringDataProj03-JpaRepo-DirectMethods:

- Copy paste the SpringDataProj01-CRUDRepo-DirectMethods application and change rootProject.name to SpringDataProj03-JpaRepo-DirectMethods in settings.gradle file.
- Need not to add any new file same SpringDataProj01-CRUDRepo-DirectMethods.
- Change the SpringBootApplication class name to SpringDataProj03JpaRepoDirectMethodsApplication
- Add the following code in their respective files.

CustomerRepo.java

```java
package com.nt.repo;

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

import com.nt.entity.Customer;

public interface CustomerRepo extends JpaRepository<Customer, Integer>
{
}
```

CustomerMgmtService.java

```java
package com.nt.service;

import java.util.List;

import com.nt.dto.CustomerDTO;

public interface CustomerMgmtService {
	public List<CustomerDTO>
fetchAllRecordsByGivenExampleDTO(CustomerDTO dto);
}
```

CustomerMgmtServiceImpl.java

```
package com.nt.service;

import java.util.ArrayList;
import java.util.List;

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

import com.nt.dto.CustomerDTO;
import com.nt.entity.Customer;
import com.nt.repo.CustomerRepo;

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

	@Autowired
	private CustomerRepo custRepo;

	@Override
	public List<CustomerDTO>
fetchAllRecordsByGivenExampleDTO(CustomerDTO dto) {
		Customer entity = null;
		Example<Customer> example = null;
		List<Customer> listEntity = null;
		List<CustomerDTO> listDTO = new  ArrayList<>();
		//convert DTO to entity
		entity = new Customer();
		BeanUtils.copyProperties(dto, entity);
		//perpare Examplple object
		example = Example.of(entity);
		//use repo
		listEntity = custRepo.findAll(example);
		//covert listEntity to listDTO
		listEntity.forEach(entity1-> {
			CustomerDTO dto1 = new CustomerDTO();
			BeanUtils.copyProperties(entity1, dto1);
```

```
                              listDTO.add(dto1);
                    });
                    return listDTO;
          }

}
```

SpringDataProj03JpaRepoDirectMethodsApplication.java

```
package com.nt;

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

import com.nt.dto.CustomerDTO;
import com.nt.service.CustomerMgmtService;

@SpringBootApplication
public class SpringDataProj03JpaRepoDirectMethodsApplication {

          public static void main(String[] args) {
                    ApplicationContext ctx = null;
                    CustomerMgmtService service = null;
                    CustomerDTO dto = null;
                    //get AC IoC container
                    ctx =
SpringApplication.run(SpringDataProj03JpaRepoDirectMethodsApplication.c
lass, args);
                    //get service class object
                    service = ctx.getBean("custService",
CustomerMgmtService.class);
                    //invoke method
                    dto = new CustomerDTO();
                    dto.setCadd("hyd");

          service.fetchAllRecordsByGivenExampleDTO(dto).forEach(System.out
::println);
```

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

}
```

Q. what is difference among findAll() methods available 3 Repositories?
Ans.

- findAll() of CrudRepository does not support pagination and sorting, returns Iterable(S extends T) object and doesn't allow Example objects as arguments.
- findAll() of PagingAndSortingRepository supports pagination and sorting, returns Iterable(S extends T) object and doesn't allow Example objects as arguments.
- findAll() of JpaRepository returns List<S extends T> collection support is there for Sorting no support pagination, allows Example objects as arguments.

- public void deleteAllInBatch() - deletes the records in a batch by generating single delete SQL query.

CustomerMgmtService.java

```
        public String removeAllCustomer();
```

CustomerMgmtServiceImpl.java

```
        @Override
        public String removeAllCustomer() {
                boolean flag = false;
                // TODO Auto-generated method stub
                if(custRepo.count()>=1) {
                        custRepo.deleteAllInBatch();
                        flag = true;
                } else {
                        flag = false;
                }
                return flag?"All records are deleted":"No records are exist";
        }
```

SpringDataProj03JpaRepoDirectMethodsApplication.java

```
System.out.println("------------------------");
System.out.println(service.removeAllCustomer());
```

Q. Why there is no update(-) and saveOrUpdate(-) in Spring Data JPA?
Ans. Since save(-) method internally persist() or merge() to perform save object or update object operation So there is no need of separate update(-) method. save(-) method calls persist(-)(JPA) if records already not available otherwise calls merge(-) (JPA) to update the existing record.

Internal Code of save(-) of Data JPA:

```
@Transactional
@Override
public <S extends T> S save(S entity) {
        if (entityInformation.isNew(entity)) {
                em.persist(entity);
                return entity;
        } else {
                return em.merge(entity);
        }
}
```

Custom Persistence logic using Spring Data JPA

a. Using findByXxx() methods declaration in Repository interface
b. Using @Query methods (HQL/JPQL, SQL queries)

Using findByXxx() methods declaration in Repository interface

- Converts findByXxx() abstract method declared in Repository into SQL Query dynamically at runtime.
- Syntax:
  <Return type> findBy<Property Name:><Condition>(<params>);
- Supports only Select operations.
- For non-select operations go for @Query methods.
- Supports both Entity Query Operations (selecting all column values) and Scalar Que Operations (Also called Projections) (select specific col values aggregate results).

Note:

- ✓ On static Properties Dependency Injections are not possible even autowiring also, not possible.
- ✓ We can get Bean class obj from Spring container by passing object type or reference type.
  //get Bean class object
  custRepo=ctx.getBean(CustomerRepo.class);
- ✓ Container generated Proxy class for Data Repository internally becomes spring bean automatically.

Entity Queries (getting all column values by Condition):

- ➢ If no condition is taken default condition is Where with " is " or "=".
- ➢ If wrong property name is given in findByXxx method then we get Invalid derived query! No property cadd1 found for type Customer! Did you mean 'Xxx'?

Directory Structure of SpringDataProj04-JpaRepo-FinderMethods:

- Copy paste the SpringDataProj01-CRUDRepo-DirectMethods application and change rootProject.name to SpringDataProj04-JpaRepo-FinderMethods in settings.gradle file.
- Remove the service package [com.nt.service], we will direct call Repository method in client application.
- Change the SpringBootApplication class name to SpringDataProj04JpaRepoFinderMethodsApplication.
- Add the following code in their respective files.

CustomerRepo.java

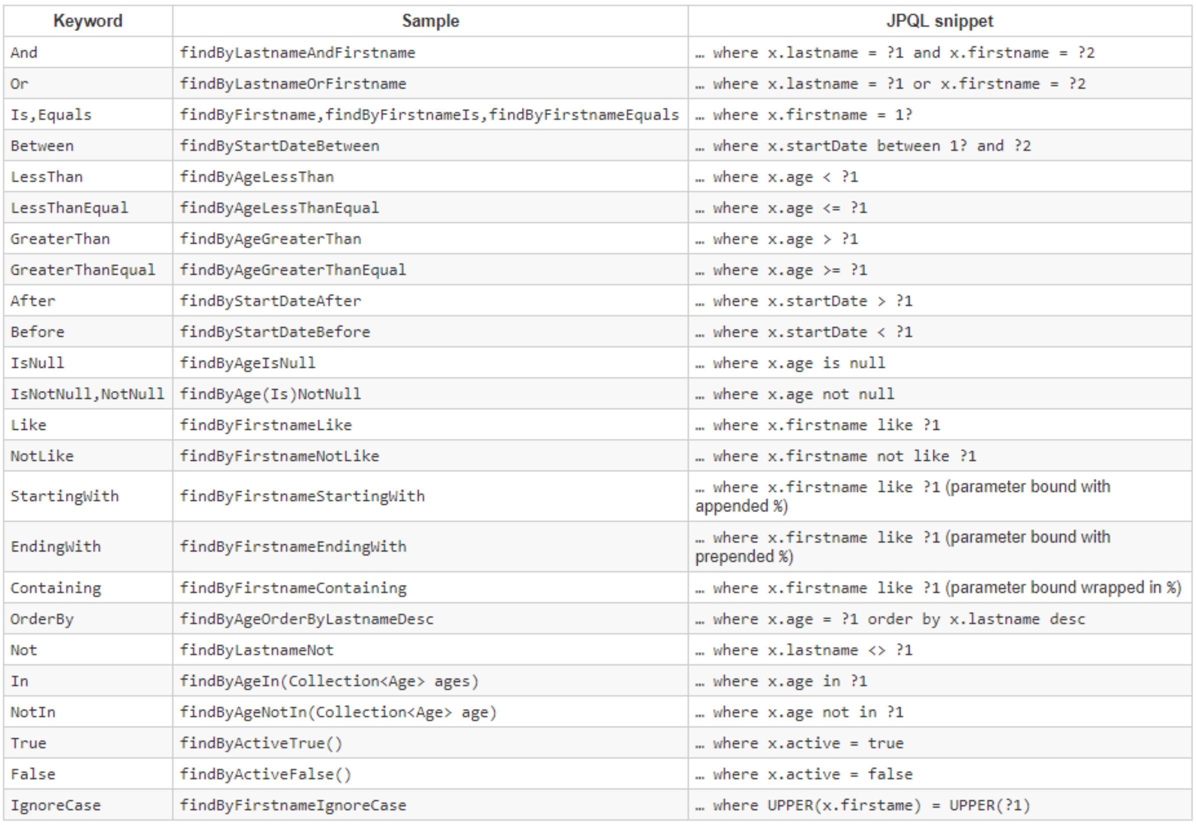
```
package com.nt.repo;

import java.util.List;

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

import com.nt.entity.Customer;

public interface CustomerRepo extends JpaRepository<Customer, Integer>
{

        //SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CADD=?
```

```
	List<Customer> findByCadd(String address);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CNAME=?
	List<Customer> findByCname(String name);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
BILLAMT>?
	List<Customer> findByBillAmountGreaterThan(double amount);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
BILLAMT<?
	List<Customer> findByBillAmountLessThan(double amount);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CNAME LIKE 'r%'
	List<Customer> findByCnameLike(String initChars);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CNAME LIKE 'r%'
	List<Customer> findByCnameStartingWith(String initChars);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CNAME LIKE '%h'
	List<Customer> findByCnameEndingWith(String lastChars);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CNAME LIKE '%j%'
	List<Customer> findByCnameContaining(String Chars);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CADD IS NULL
	Iterable<Customer> findByCaddIsNull();
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE 
CADD NOt NULL
	Iterable<Customer> findByCaddIsNotNull();

}
```

SpringDataProj04JpaRepoFinderMethodsApplication.java

```
package com.nt;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;
import com.nt.repo.CustomerRepo;

@SpringBootApplication
public class SpringDataProj04JpaRepoFinderMethodsApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		CustomerRepo custRepo = null;
		//get AC IoC container
		ctx =
SpringApplication.run(SpringDataProj04JpaRepoFinderMethodsApplication.
class, args);
		//Get Repo bean
		custRepo = ctx.getBean(CustomerRepo.class);
		//invoke the methods
		custRepo.findByCadd("hyd").forEach(System.out::println);
		custRepo.findByCname("Raja").forEach(System.out::println);

	custRepo.findByBillAmountGreaterThan(3000).forEach(System.out::p
rintln);

	custRepo.findByBillAmountLessThan(2000).forEach(System.out::printl
n);

	custRepo.findByCnameLike("raj%").forEach(System.out::println);

	custRepo.findByCnameStartingWith("r").forEach(System.out::println);

	custRepo.findByCnameEndingWith("h").forEach(System.out::println);

	custRepo.findByCnameContaining("raj").forEach(System.out::println);
		custRepo.findByCaddIsNull().forEach(System.out::println);
		custRepo.findByCaddIsNotNull().forEach(System.out::println);

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

}
```

Keyword	Sample	Logical result
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"$gt" : from, "$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"$in" : [ages…]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"$nin" : [ages…]}}
IsNotNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}
Like, StartingWith, EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike, IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "$not" : name }} (name as regex)
Containing on String	findByFirstnameContaining(String name)	{"firstname" : name} (name as regex)
NotContaining on String	findByFirstnameNotContaining(String name)	{"firstname" : { "$not" : name}} (name as regex)
Containing on Collection	findByAddressesContaining(Address address)	{"addresses" : { "$in" : address}}
NotContaining on Collection	findByAddressesNotContaining(Address address)	{"addresses" : { "$not" : { "$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"$regex" : firstname }}
(No keyword)	findByFirstname(String name)	{"firstname" : name}
Not	findByFirstnameNot(String name)	{"firstname" : {"$ne" : name}}
Near	findByLocationNear(Point point)	{"location" : {"$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"$near" : [x,y], "$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"$geoWithin" : {"$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"$geoWithin" : {"$box" : [[x1, y1], x2, y2]}}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"$exists" : exists }}



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.firstame) = UPPER(?1)

Some more Methods:

CustomerRepo.java

```
//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE CNAME
LIKE '_____'
	Iterable<Customer> findByCnameLike(String chars);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CADD LIKE '_____%'
	Iterable<Customer> findByCaddLike(String chars);

	//--------------Working with More than one property Condition

	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
BILLAMT>400 AND BILLAMT<5000
	Iterable<Customer>
findByBillAmountGreaterThanAndBillAmountLessThan(double min, double
max);
	//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
BILLAMT BETWEEN (4000, 5000)
	Iterable<Customer> findByBillAmountBetween(double min, double
max);
```

```
//SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CNAME='raja' OR CADD='hyd'
        Iterable<Customer> findByCnameEqualsOrCaddEquals(String name,
String address);
        //SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
BILLAMT BETWEEN (4000, 5000) ORDER BY CNAME DESC
        Iterable<Customer>
findByBillAmountBetweenOrderByCnameDesc(double min, double max);
        //SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CADD <> 'hyd'
        //SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CADd != 'hyd'
        Iterable<Customer> findByCaddNot(String name);
        //SELECT CNO, CNAME, CADD, BILLAMT FROM CUSTOMER WHERE
CADD IN ('hyd', 'vizg, 'delihi');
        Iterable<Customer> findByCaddIn(Collection<String> citis);
```

SpringData04JpaRepoFinderMethodsApplication.java

```
System.out.println("--------------------------");

        custRepo.findByCnameLike("___").forEach(System.out::println);

        custRepo.findByCaddLike("___%").forEach(System.out::println);

        custRepo.findByBillAmountGreaterThanAndBillAmountLessThan(400
0, 5000).forEach(System.out::println);
                custRepo.findByBillAmountBetween(4000,
5000).forEach(System.out::println);
                custRepo.findByCnameEqualsOrCaddEquals("raja",
"hyd").forEach(System.out::println);
                custRepo.findByBillAmountBetweenOrderByCnameDesc(4000,
5000).forEach(System.out::println);
                custRepo.findByCaddIn(List.of("hyd", "vizg",
"delhi")).forEach(System.out::println);
```

Note: if findByXxx (-) is returning single Record having all col values then we can take just <T> as the return type instead of List<T> or Iterable<T>

```
        Customer findByCname(String name);
```

Using findByXxx() method for Scalar Operations using Projections

- findByXxx (-) are abstract methods generating select SQL queries either giving all column values (Entities queries) or specific col values (scalar queries Using Projections) by applying where clause condition.

findByXxx methods with Projections:

- Projections are columns in SQL getting records by specifying col names is called Projections concept.
    - Static Projections (Always gives fixed specific col values)
    - Dynamic Projections (we can get varying column values)

Static Projections (scalar queries):

1. Take an interface as inner interface repository interface and declare getter methods by choosing the properties of entity class.
```
interface ResultsView1{
        Integer getCno();
        String getCname();
}
```
2. Design findByXxx having the above Type View interface as part of return type or as return type.
```
//SELECT CNO, CNAME FROM CUSTOMER WHERE CADD=?
List<ResultsView1> findByCadd(String addrs);
```
3. Call the above method in service class/client App to get the result and to process the result.
```
//invoke method
List<ResultsView1> viewList = custRepo.findByCadd(“hyd");
ResultsView1.forEach(rv -> System.out.println(v.getCno()+" "
                              +v.getCname()+" "+v.getClass());
```

Note:

- ✓ Here we cannot change view type dynamically in the return of findByXxx (-) method.
- ✓ Entity operations - Getting all column values.
- ✓ Scalar operations - Getting specific column values support.

Directory Structure of SpringDataProj05-JpaRepo-FinderMethods-ScalarQuery-Projection:

- Copy paste the SpringDataProj04-JapRepoFinderMethods application and change rootProject.name to SpringDataProj05-JpaRepo-FinderMethods-ScalarQuery-Projection in settings.gradle file.

- Change the SpringBootApplication class name to SpringDataProj05JpaRepoFinderMethodsProjectionApplication.
- Add the following code in their respective files.

CutomerRepo.java

```
package com.nt.repo;

import java.util.List;

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

import com.nt.entity.Customer;

public interface CustomerRepo extends JpaRepository<Customer, Integer> {

	interface ResultView1 {
		Integer getCno();
		String getCname();
	}

	//SELECT CNO, CNAME FROM CUSTOMER WHERE CADD =?;
	List<ResultView1> findByCadd(String address);

	interface ResultView2 {
		String getCname();
		Double getBillAmount();
	}

	//SELECT CNO, CNAME FROM CUSTOMER WHERE CNO BETWEEN (?,
?);
	List<ResultView2> findByCnoBetween(int start, int end);

}
```

SpringDataProj05JpaRepoFinderMethodsProjectionApplication.java

```
package com.nt;

import java.util.List;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ConfigurableApplicationContext;

import com.nt.repo.CustomerRepo;
import com.nt.repo.CustomerRepo.ResultView1;

@SpringBootApplication
public class SpringDataProj05JpaRepoFinderMethodsProjectionApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		CustomerRepo custRepo = null;
		//get AC IoC container
		ctx = 
SpringApplication.run(SpringDataProj05JpaRepoFinderMethodsProjectionA
pplication.class, args);
		//Get Repo bean
		custRepo = ctx.getBean(CustomerRepo.class);
		//invoke the methods
		List<ResultView1> listRV = custRepo.findByCadd("hyd");
		listRV.forEach(rv -> System.out.println(rv.getCno()+"
"+rv.getCname()));
		System.out.println("------------------------");
		custRepo.findByCnoBetween(2, 4).forEach(rv ->
System.out.println(rv.getCname()+" "+rv.getBillAmount()));
		//close container
		((ConfigurableApplicationContext) ctx).close();
	}

}
```

Core Java Recap:

```
class Person {
	............
}
```

```
class Employee extends Person {
	...........
}
```

```
class Customer exstends Person {
        ...........
}
```

```
class Student extends Person {
        .............
}
```

Design method having ability to display any object data:

Option 1: public void display (Object obj) - Not good practice

- typecasting is required, may give ClassCastException

Option 2: public void display (Person obj) - Not good practice

- typecasting is required, may give ClassCastException

Option3: public void display (Class<T> clazz) (jdk5 Generics)

Good practice and avoids ClassCastException

public $\underbrace{\text{<T extends Person>}}_{*}$ void display (Class<T> clazz)

(More Good Practice)

* is not return type of method, it is Generic type declaration. So that can be used throughout method signature and definition in return type, parameter types.

Dynamic Projections (Scalar operations):

- Allows to view type dynamically at runtime, by designing the method with the support of java5 Generics.

1. Take multiple view type interfaces having declaration of getter methods.
```
interface View {

}
interface ResultsView1 extends View {
        Integer getCno();
        String getCname();
}
interface ResultsView2 extends View {
        String getCname();
        double getBillAmt();
}
```
2. Design the method with generics with dynamic Projections
```
// List<T> findByCadd(String addrs Class<T> clzz);
<T extends View> List<T> findByCadd(String addrs, Class<T> clzz);
```
3. Invoke methods in client App [service class]
```
List<ResultsView viewList - custRepo.findByCadd("hyd",
ResultsView1.class);
```

```
            for (ResultsView1 v : view1List) {
                        System.out.println(v.getcname()+" "+v.getBillAmtt();
            }
```

CutomerRepo.java

```
            //--------- Dynamic Projections -------------------
            //SELECT CNO, CNAME FROM CUSTOMER WHERE CADD =?;
            //<T> List<T>  findByCadd(String address, Class<T> clazz);
            <T extends View> List<T>  findByCadd(String address, Class<T> clazz);
```

SpringDataProj05JpaRepoFinderMethodsProjectionApplication.java

```
                custRepo.findByCadd("hyd", ResultView1.class).forEach(rv ->
System.out.println(rv.getCno()+" "+rv.getCname()));
                custRepo.findByCadd("hyd", ResultView2.class).forEach(rv ->
System.out.println(rv.getCname()+" "+rv.getBillAmount()));
                custRepo.findByCadd("hyd", ResultView2.class).forEach(rv ->
System.out.println(rv.getCname()+" "+rv.getBillAmount()));
```

Limitations of findByXxx () while wring custom Persistence logics in Spring data JPA:

a. Supports only select operations.
b. Writing findByXxx (-) method with multiple properties and multiple conditions makes method name very big and complex.
c. Working with Projections (Scalar Operations) bit complex. (Especially taking view type interfaces for different combinations we need to take different view type interfaces).
d. Not So readable for programmers.
e. Does not support aggregate operations.

Conclusion: Use findByXxx (-) methods only for simple select operations with simple conditions. To overcome these problems and to write Custom persistence logic in all possible angels go for @Query methods.

Q. Why @Query methods do not support single record operation?
Ans. Query based insertion cannot use generators to generate the id values. So use ses.save(-) for that.

@Query methods

- ➢ Allows to write persistence logic using custom HQL/ JPQL and Native SQL Queries.
- ➢ Support both select and non-select operations (except insert operation - use ses.save(-) for insert operation).
- ➢ Supports both single row and bulk operations with our choice conditions.
- ➢ Query can have both positional (?1, ?2, ?3, ...) and named parameters (:<var1>, :<var2>, :<max>) (named params are good).
- ➢ Allows to invoke PL/ SQL procedures and functions.
- ➢ Supports joins.
- ➢ Allows both Entity and scalar query operations.

Syntax:

In Repository interface.

@Query (" query with params ?, ?, ?) even place named params
public <Return Type> <method> (param1, param2, param3);

- ➢ Java method param values will be mapped with query positional params (?) automatically.
- ➢ Java method param values will be mapped with query name params(?) automatically if both names are matching. If not matching we can bind them using @Param annotation.

HQL/ JPQL:

- Will be written based on Entity class name and property names (not based on DB table names and col names).
- These queries are DB s/w independent because these object-based SQL queries.
- These query partial case -sensitive i.e. HQL/ JPQL keyworks are not case-sensitive but entity class name, property names are case-sensitive.
- These queries learning curve is very small.
- These queries not support JDBC style positional params like ?, ? , ? but supports JPA style positional params ?1, ?2, ?3 because hibernate 5.2 onwards support for JDBC style positional params have been removed.

```
SQL> SELECT * FROM CUSTOMER (DB table name)
                                        Entity class name
HQL/ JPQL> SELECT cust FROM com.nt.entity.Customer cust
HQL/ JPQL> FROM Customer cust
```

HQL/ JPQL> FROM Customer

Alias name

SQL> SELECT * FROM CUSTOMER WHERE AND CNO>=? AND CNO>=?
HQL/ JPQL> FROM Customer WHERE AND cno>=?1 AND cno>=?2

- All HQL/ JPQL queries internally converted to SQL queries with positional params before sending them DB s/w. Because DB s/w understands only SQL queries.

Directory Structure of SpringDataProj06-JpaRepo-@QueryMethods:

- Copy paste the SpringDataProj05-JpaRepo-FinderMethods-ScalarQuery-Projection application and change rootProject.name to SpringDataProj06-JpaRepo-@QueryMethods in settings.gradle file.
- Change the SpringBootApplication class name to SpringDataProj06JpaRepoQueryMethodsApplication.
- Add the following code in their respective files.

CutomerRepo.java

```
package com.nt.repo;

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

import com.nt.entity.Customer;

public interface CustomerRepo extends JpaRepository<Customer, Integer> {

	// Select bulk operation with positional param (Entity query)
	@Query("FROM Customer")
	Iterable<Customer> getAllCustomers();

	@Query("FROM Customer WHERE cadd=?1")
	Iterable<Customer> getCustomerByCity(String city);

	@Query("FROM Customer WHERE billAmount>=?1 AND 
billAmount<=?2")
	Iterable<Customer> getCustomerByBillAmountRange(double start, 
double end);

}
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
package com.nt;

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

import com.nt.repo.CustomerRepo;

@SpringBootApplication
public class SpringDataProj06JpaRepoQueryMethodsApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		CustomerRepo custRepo = null;
		//get AC IoC container
		ctx = 
SpringApplication.run(SpringDataProj06JpaRepoQueryMethodsApplication.
class, args);
		//Get Repo bean
		custRepo = ctx.getBean(CustomerRepo.class);
		//invoke the methods
		custRepo.getAllCustomers().forEach(System.out::println);
	custRepo.getCustomerByCity("hyd").forEach(System.out::println);
	custRepo.getCustomerByCity("hyd").forEach(System.out::println);
		custRepo.getCustomerByBillAmountRange(200000, 
400000).forEach(System.out::println);

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

}
```

```
@Query("FROM Customer WHERE billAmount>=?1 AND billAmount<=?2")
Iterable<Customer> getCustomerByBillAmountRange(double start, double
end);
```

- When multiple positional params we can change their order but there should not be any gap in the numbering, we need take params in the method according to that changed order.
- Giving index to more than 3 or 4 positional params is very complex to overcome this problem use named params (:<name>) (parameter with name).
- If you use JDBC style plain positional parameters (?) in HQL/ JPQL queries then we get org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'customerRepo': FactoryBean threw exception on object creation; nested exception is java.lang.IllegalArgumentException: JDBC style parameters (?) are not supported for JPA queries.

CutomerRepo.java

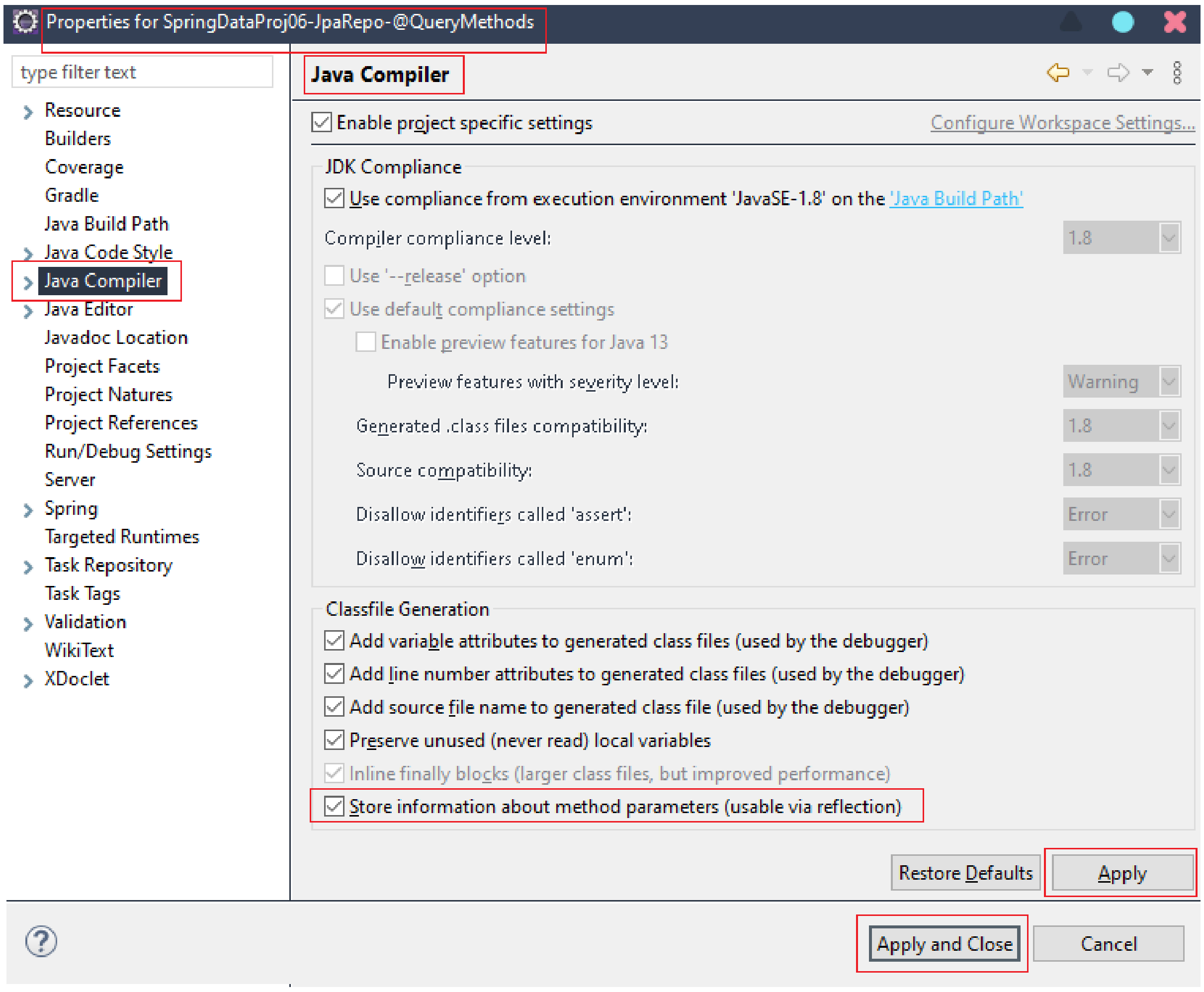
```
	//Select bulk operation with Named param (Entity query)
	/*@Query("FROM Customer WHERE cadd IN(:cityOne, :cityTwo, :cityThree)")
	Iterable<Customer> getCustomerByCityNames(@Param("cityOne")String city1,

	@Param("cityTwo")String city2,

	@Param("cityThree")String city3);*/

	@Query("FROM Customer WHERE cadd IN(:cityOne, :cityTwo, :cityThree)")
	Iterable<Customer> getCustomerByCityNames(String cityOne, String cityTwo, String cityThree);

	@Query("FROM Customer WHERE cname=:name")
	Iterable<Customer> getCustomerByName(String name);
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
		custRepo.getCustomerByCityNames("hyd", "vizg", "delih").forEach(System.out::println);

	custRepo.getCustomerByName("hari").forEach(System.out::println);
```

Note:

- ✓ @Param is used to java method param values to HQL/JPQL named param value.
- ✓ If named param name and java method param name are matching then no need of giving @Param.
- ✓ If you want to work with named param without @param then you have to change the following settings in your particular project.
  Right click on you project, then go to Properties then click on Java compiler then enable the following check box then Apply after that Apply and close.


Properties for SpringDataProj06-JpaRepo-@QueryMethods

Java Compiler

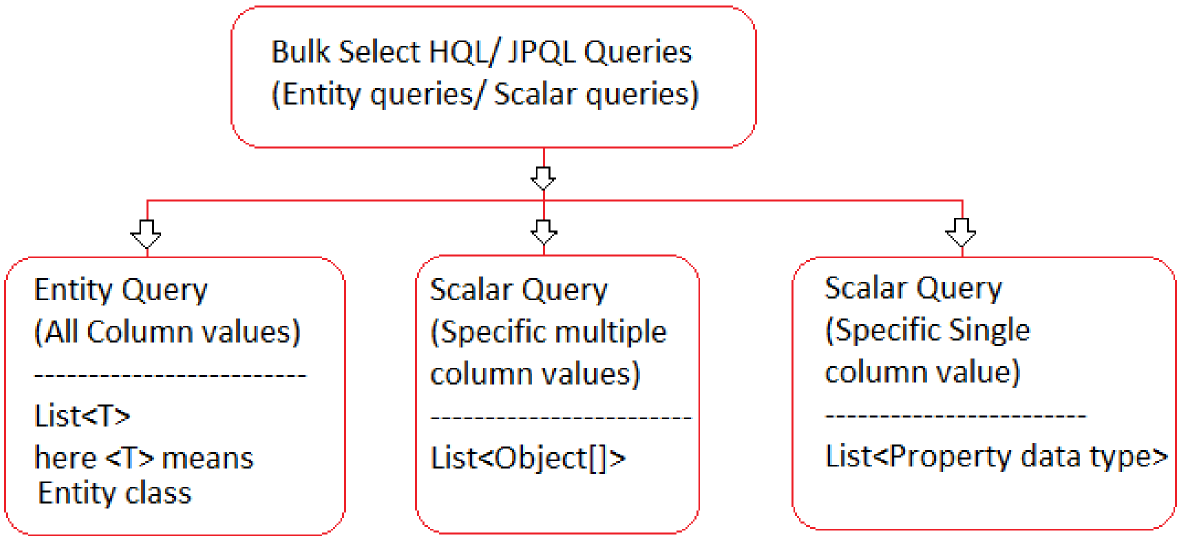
☑ Enable project specific settings — Configure Workspace Settings...

JDK Compliance
☑ Use compliance from execution environment 'JavaSE-1.8' on the 'Java Build Path'
Compiler compliance level: 1.8
☐ Use '--release' option
☑ Use default compliance settings
☐ Enable preview features for Java 13
Preview features with severity level: Warning
Generated .class files compatibility: 1.8
Source compatibility: 1.8
Disallow identifiers called 'assert': Error
Disallow identifiers called 'enum': Error

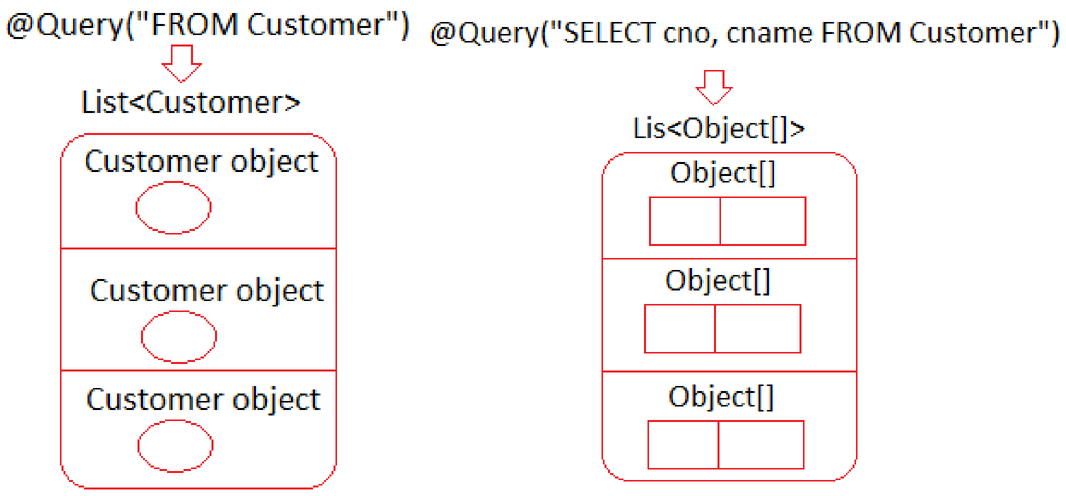
Classfile Generation
☑ Add variable attributes to generated class files (used by the debugger)
☑ Add line number attributes to generated class files (used by the debugger)
☑ Add source file name to generated class file (used by the debugger)
☑ Preserve unused (never read) local variables
☑ Inline finally blocks (larger class files, but improved performance)
☑ Store information about method parameters (usable via reflection)

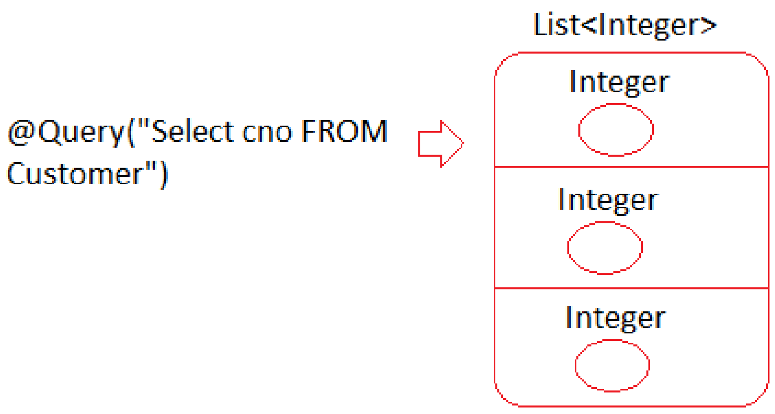
We cannot place both positional and named parameters in single @Query method JPQL/ HQL query, it throws org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'customerRepo': FactoryBean threw exception on object creation; nested exception is java.lang.IllegalStateException: Using named parameters for method public abstract java.lang.Iterable com.nt.repo.CustomerRepo.getCustomerByCnoRange(double,double) but parameter 'Optional[start]' not found in annotated query 'FROM Customer WHERE cno>=?1 AND cno<=:end'!

```
        @Query("FROM Customer WHERE cno>=?1 AND cno<=:end")
         Iterable<Customer> getCustomerByCnoRange(double start, double
end);
```



Bulk Select HQL/ JPQL Queries (Entity queries/ Scalar queries)
Entity Query (All Column values)
List<T> here <T> means Entity class
Scalar Query (Specific multiple column values)
List<Object[]>
Scalar Query (Specific Single column value)
List<Property data type>



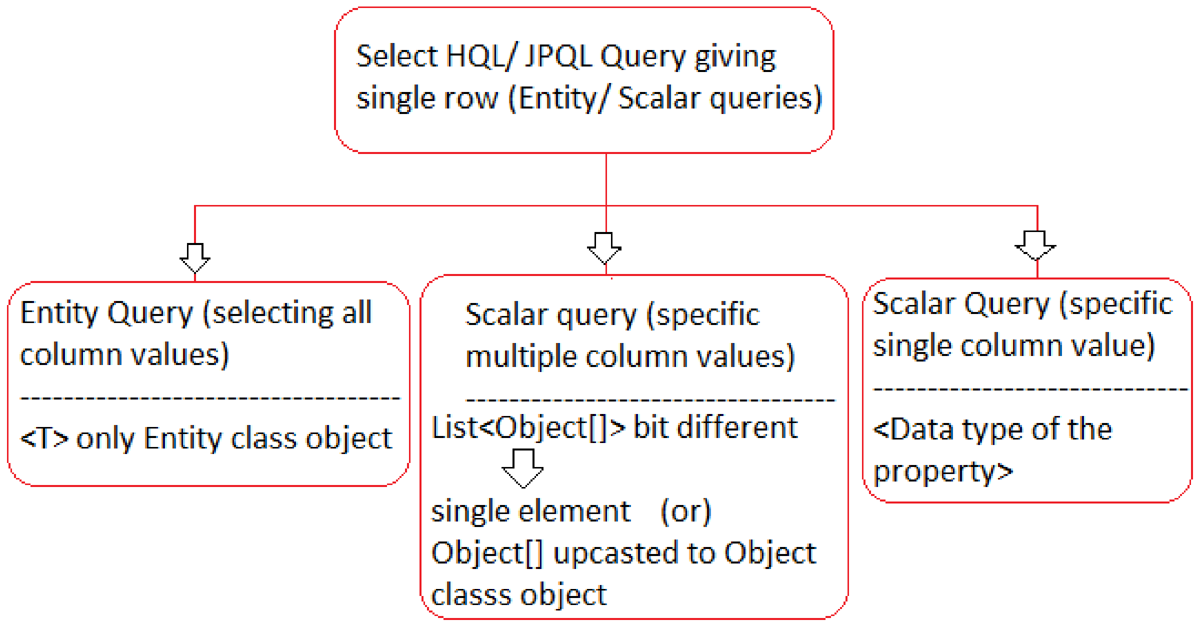


CutomerRepo.java

```
	//Select bulk operation with Named param (Scalar query) Multiple columns
	@Query("SELECT cno, cname FROM Customer WHERE cadd=:city")
	Iterable<Object[]> getCustomerValuesByCity(String city);

	//Select bulk operation with Named param (Scalar query) specific single columns
	@Query("SELECT billAmount FROM Customer WHERE cadd in (:city1, :city2)")
	Iterable<Double> getBillAmountByCities(String city1, String city2);
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
		custRepo.getCustomerValuesByCity("hyd").forEach(row -> {
			//System.out.println(row[0]+" "+row[1]);
			for (Object val:row) {
				System.out.print(val+" ");
			}
			System.out.println();
		});
			custRepo.getBillAmountByCities("hyd",
	"vizg").forEach(System.out::println);
```



Select HQL/ JPQL Query giving single row (Entity/ Scalar queries)
Entity Query (selecting all column values)
<T> only Entity class object
Scalar query (specific multiple column values)
List<Object[]> bit different
single element (or) Object[] upcasted to Object classs object
Scalar Query (specific single column value)
<Data type of the property>

CutomerRepo.java

```
	//Single Row Select operation HQL/ JPQL (Entity query) specific single
columns
	@Query("FROM Customer WHERE cname= :name")
	Customer getCustomerByName(String name);

	//Single Row Select operation HQL/ JPQL (Scalar query) Multiple
columns
	@Query("SELECT cno, cname FROM Customer WHERE cname=
:name")
	//List<Object[]> getDataValueByName(String name);
	Object getDataValueByName(String name);

	//Single Row Select operation HQL/ JPQL (Scalar query) single
columns
	@Query("SELECT billAmount FROM Customer WHERE cname=
:name")
	Object getBillAmountByName(String name);


	//Select operation Aggregate functions
	@Query("SELECT MAX(billAmount) FROM Customer")
	Double findMaxBillAmount();

	//Select operation Aggregate functions
	@Query("SELECT MAX(billAmount), SUM(billAmount),
AVG(billAmount), COUNT(*) FROM Customer")
	Object findAggregateResults();
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
		System.out.println(custRepo.getCustomerByName("hari"));
		/*List<Object[]> list = custRepo.getDataValueByName("hari");
		Object value[] = list.get(0);
		System.out.println(value[0]+" "+value[1]);*/
		Object obj[] = (Object[])
custRepo.getDataValueByName("hari");
		System.out.println(obj[0]+" "+obj[1]);

		System.out.println(custRepo.getBillAmountByName("hari"));
```

```
                System.out.println("Max Bill amount is
:"+custRepo.findMaxBillAmount());
                Object result[] = (Object[]) custRepo.findAggregateResults();
                System.out.println("Max : "+result[0]);
                System.out.println("Sum : "+result[1]);
                System.out.println("Avg : "+result[2]);
                System.out.println("Count : "+result[3]);
```

Non-Select Operations (both bulk and single row operations):

- Here we can take our choice conditions queries.
- Only update and delete queries are possible for insert use repo.save(-) method.
    1. To take advantage of generators.
    2. Generally, insert query does not need any condition
    3. HQL/ JPQL is not having any insert Query.
- Use @Query, @Modify together on the methods.
- If service class is not taken, we should @Transactional explicitly (Add in the Repository interface), otherwise not required.

CutomerRepo.java

```
        //Update operation
        @Modifying
        @Query("UPDATE Customer SET
billAmount=billAmount+:extraAmount WHERE cadd= :city")
        int modifyCustomeByCity(String city, double extraAmount);

        //Delete operation
        @Modifying
        @Query("DELETE Customer WHERE cadd IS NULL")
        int deleteCustomeIfCadIsNull();
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
                System.out.println("Number of records are updated :
        "+custRepo.modifyCustomeByCity("hyd", 100));
                System.out.println("Number of records are deleted :
"+custRepo.deleteCustomeIfCadIsNull());
```

Native SQL/ Original SQL Queries

- These DB s/w are dependent queries so makes persistence logic as DB s/w dependent.
- Use this when HQL/ JPQL does not support certain operation (like insert operation, calling PL/ SQL procedure function calling DB s/w specific aggregate functions like sysDate(oracle), now () (MySQL) and etc.).
- It allows both named, positional params (JPA style [?1, ?2, ...], JDBC style [?, ?,……].
- Use Native SQL to perform those operations that are not possible with HQL/ JPQL like using specific aggregate functions, insert query, calling PL/SQL procedures and functions.

Note:

✓ Native SQL supports JDBC style positional params (?), JPA style positional params (?1) and named params (:name).

✓ Native SQL quires based persistence logics are bad because it makes persistence logic as DB s/w dependent persistence logic.

CutomerRepo.java

```
	//Execute Native SQL select queries
	//@Query(nativeQuery = true, value = "SELECT CNO, CNAME, CADD, 
BILL_AMOUNT FROM CUSTOMER WHERE CADD=?")
	//@Query(nativeQuery = true, value = "SELECT CNO, CNAME, CADD, 
BILL_AMOUNT FROM CUSTOMER WHERE CADD=?1")
	@Query(nativeQuery = true, value = "SELECT CNO, CNAME, CADD, 
BILL_AMOUNT FROM CUSTOMER WHERE CADD=:address")
	Iterable<Customer> getCustomersByAddress(String address);

	//get System date
	@Query(nativeQuery = true, value="SELECT SYSDATE FROM DUAL")
	java.util.Date getSystemDate();

	//Insert operation
	@Query(nativeQuery = true, value="INSERT INTO CUSTOMER VALUES 
(CNO_SEQ.NEXTVAL, ?, ?,?)")
	@Modifying
	int insertCustomer(double billAmount, String address, String name);
```

SpringDataProj06JpaRepoQueryMethodsApplication.java

```
		custRepo.getCustomersByAddress("hyd").forEach(System.out::
	println);
		System.out.println(custRepo.getSystemDate());
		int count = custRepo.insertCustomer(234.3, "katu", "kalia");
		System.out.println(count==0?"Record is not inserted":"Record
	is inserted");
```

Calling PL/SQL Procedure and Function

- PL/SQL procedure or function is like a java method to execute bunch of statement together in a single block.
- PL/SQL procedure does not return a value where function returns a value.
- Industry uses more of PL/SQL procedures because we can result through, out params.
- Java method contains param name and type whereas PL/SQL procedure or function params will have param name, type and mode (IN, OUT, INOUT).
  - IN->INPUT
  - OUT->OUTPUT
  - INOUT ->INPUT and OUTPUT

  y: = x*x; y is out param, x is IN param
  x: = x*x (x is INOUT param)

  In oracle PL/SQL
  = (for comparison)
  := (for assignment)

Note: PL/SQL programming is specific to each DB s/w.

- Instead of writing same persistence logic in multiple modules/ Apps of a Project in the form of SQL or HQL/JPQL queries, it recommended to write only for 1 time as PL/SQL procedure or function and use it multiple Apps or modules.
  a. Authentication logic
  b. Attendance calculation logic and etc.
- Java Projects 60 to 80% => SQL or HQL/JPQL based Queries.
  20 to 40% => PL/SQL procedures.

In Spring Data JPA we call PL/SQL procedures in 3 ways:

a. Using @Query (Native SQL query approach) we can call only IN params, no params procedures
b. Using @Procedure
c. Using EntityManager (** Best)

We can call any procedure having in params, out params or no params procedures

Calling PL/SQL Procedure using @Query:

Step 1: Create PL/SQL procedure in MySQL having IN Params in using Workbench

Workbench -> go to your ntsp612db -> right click stored procedures -> create procedure

Name: GET CUSTOMERS BY ADDS -> type the code replacing null

Step 2: Prepare @Query method in the Repository interface.

Step 3: Invoke the method in the client App

Directory Structure of SpringDataProj07-CallingProcedures:

- Copy paste the SpringDataProj06-JpaRepo-@QueryMethods application and change rootProject.name to SpringDataProj07-CallingProcedures in settings.gradle file.
- Change the SpringBootApplication class name to SpringDataProj07CallingProceduresApplication.
- Add the following code in their respective files.

GET_CUSTOMERS_BY_ADDS Procedure details

```
USE `nshb413`;

DROP procedure IF EXISTS `GET_CUSTOMERS_BY_ADDS`;

DELIMITER $$

USE `nshb413`$$

CREATE DEFINER=`root`@`localhost` PROCEDURE
`GET_CUSTOMERS_BY_ADDS` (IN addrs varchar (10))

BEGIN

select cno, cname, cadd, bill_Amount from customer where cadd =addrs;

END$$

DELIMITER;
```

build.gradle

```
        // https://mvnrepository.com/artifact/mysql/mysql-connector-java
        implementation group: 'mysql', name: 'mysql-connector-java',
version: '8.0.21'
```

application.properties

```
#MySQL
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///nshb413
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100


spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDia
lect
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialec
t
```

CustomerRepo.java

```
        //Calling Procedure and function
        //@Query(nativeQuery = true, value = "{call
GET_CUSTOMERS_BY_ADDS(?)}")
        //@Query(nativeQuery = true, value = "{call
GET_CUSTOMERS_BY_ADDS(?1)}")
        @Query(nativeQuery = true, value = "{call
GET_CUSTOMERS_BY_ADDS(:addrs)}")
        Iterable<Customer> featchCustomerDataByAddrs(String addrs);
```

SpringDataProj07CallingProceduresApplication.java

```
                //call PL/SQL procedure
        custRepo.featchCustomerDataByAddrs("hyd").forEach(System.out::pr
intln);
```

Calling PL/SQL Procedure using EntityManager:

- EntityManager object manages entity objects Lifecyle and uses them for persistence operations.
- It is JPA object hold Session, SessionFactory objects if the underlying ORM is hibernate.
- It is like HibernateTemplate class object but it is common for all ORM frameworks.

Note:

- ✓ If add Spring-Boot-Data-JPA-starter to libraries The EntityManager obj comes through AutoConfiguration.
- ✓ In service class we inject using @Autowired EntityManager manager.
- ✓ In Client App we can get
  EntityManager manager = ctx.getBean(EntityManager.class);

Step 1: Make sure that above MySQL PL/SQL is running property (having entity query with IN param).
Step 2: In client App write following code to call PL/SQL procedure using EntityManager.

Note:

- ✓ Using EntityManager, no restriction to call any type of PL/SQL procedure.
- ✓ @Procedure we cannot deal with Entity Query based PL/SQL procedures, we can deal with Scalar query based that to single out param PL/SQL procedures.

SpringDataProj07CallingProceduresApplication.java

```
		//calling PL/SQL procedure using EntityManager
		EntityManager manager = ctx.getBean(EntityManager.class);
		//create StoreProcedureQuery object
		StoredProcedureQuery procedure = 
manager.createStoredProcedureQuery("GET_CUSTOMERS_BY_ADDS", 
Customer.class);
		procedure.registerStoredProcedureParameter(1, String.class, 
ParameterMode.IN);
		//set Value to In param
		procedure.setParameter(1, "hyd");
		//call Pl/SQL procedure
		procedure.getResultList().forEach(System.out::println);
```

Calling PL/SQL Procedure from MySQL that is having Scalar query with IN, OUT Params:

a. Using @Quer not possible
b. Using @Procedure possible
c. Using EntityManager possible (anything is possible)

Using @Procedure:

Step 1: Make sure PL/SQL procedure is ready in MySQL having scalar query with one OUT param and one or more IN params.
Step 2: Add the following method in Repository Interface having @Procedure
Step 3: Invoke method in client App by accessing Repository object.

GET_CUSTOMERS_COUNT_BY_ADDRS Procedure Details

```
USE `nshb413`;

DROP procedure IF EXISTS `GET_CUSTOMERS_COUNT_BY_ADDRS`;

DELIMITER $$


USE `nshb413`$$

CREATE PROCEDURE `GET_CUSTOMERS_COUNT_BY_ADDRS` (IN ADDRS
varchar(10), OUT CNT int)

BEGIN

	SELECT COUNT(*) into CNT FROM CUSTOMER WHERE CADD=ADDRS;

END$$


DELIMITER ;
```

CustomerRepo.java

```
	//Using @Procedure
	@Procedure(procedureName =
"GET_CUSTOMERS_COUNT_BY_ADDRS")
	int featchCustomersCountByAddress(String address);

	@Procedure
	int GET_CUSTOMERS_COUNT_BY_ADDRS(String address);
```

SpringDataProj07CallingProceduresApplication.java

```
                    // invoke the metho
                    System.out.println("No of records are : 
"+custRepo.featchCustomersCountByAddress("hyd"));
                    System.out.println("No of records are : 
"+custRepo.GET_CUSTOMERS_COUNT_BY_ADDRS("vizg"));
```

Note:

- ✓ If PL/SQL procedure name is taken as java method name So need of specifying procedure name separately.
- ✓ @Procedure is not industry standard.

Using EntityManager Approach for the above PL/SQL procedure:

Step 1: Make sure that above PL/SQL procedure in stable

Step 2: Writing following in client App

SpringDataProj07CallingProceduresApplication.java

```
                    //Call IN, OUT param using EntityManagere
                    //get EntityManager object
                    EntityManager manager = ctx.getBean(EntityManager.class);
                    //Create StoreProcedureQuery object
                    StoredProcedureQuery procedure = 
manager.createStoredProcedureQuery("GET_CUSTOMERS_COUNT_BY_ADD
RS");
                    //register params
                    procedure.registerStoredProcedureParameter(1, String.class, 
ParameterMode.IN);
                    procedure.registerStoredProcedureParameter(2, Integer.class, 
ParameterMode.OUT);
                    //set value to in params
                    procedure.setParameter(1, "hyd");
                    //get result
                    int count = (int) procedure.getOutputParameterValue(2);
                    System.out.println("No of records are : "+count);
```

Note:

- ✓ If result is List<T> then use getResultList () method.
- ✓ If result is List<Object []> then use getResultList () method.

- ✓ If result is in out params holding single values then use getOutParameterValue (-).

Note: In MySQL if we want to get Select query result there is no need of taking OUT param.

Calling PL/SQL Procedure of Oracle using EntityManager of Spring Data JPA:

Step 1: Create PL/SQL procedure in Oracle.
Cursor holds the select query result.

Step 2: Make sure that u r application is pointing to Oracle DB s/w.

Step 3: Use EntityManager in Client App as shown below.

Note:

- ✓ Cursor in Memory variable of oracle PL/SQL programming having capability to hold bunch of records it is like JDBC ResultSet object.
- ✓ SYS_REFCURSOR is a built-in cursor data type in oracle PL/SQL programmer.
- ✓ Oracle PL/SQL programming does no any results without OUT params. So, we cannot use @Query to call PL/SQL procedures of oracle.
- ✓ @Procedure cannot deal with entity Queries (select all column values) So we cannot use it here, we need to use only EntityManager option.

GET_CUSTOMERS_DETAILS_BY_ADDS Procedure details

```
CREATE OR REPLACE PROCEDURE GET_CUSTOMERS_DETAILS_BY_ADDRS

(

  ADDR IN VARCHAR2

, DETAILS OUT SYS_REFCURSOR

) AS

BEGIN

    OPEN DETAILS FOR

        SELECT CNO, CNAME, CADD, BILL_AMOUNT FROM CUSTOMER WHERE
CADD=ADDR;


END GET_CUSTOMERS_DETAILS_BY_ADDRS;
```

application.properties

```
#DataSource configuration details
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
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100

#JPA/ Hibernate properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDial
ect
```

SpringDataProj07CallingProceduresApplication.java

```
		// Call IN, OUT param using EntityManagere (ORACLE)
		// get EntityManager object
		EntityManager manager = ctx.getBean(EntityManager.class);
		// Create StoreProcedureQuery object
		StoredProcedureQuery query = 
manager.createStoredProcedureQuery("GET_CUSTOMERS_DETAILS_BY_AD
DRS", Customer.class);
		// register params
		query.registerStoredProcedureParameter(1, String.class, 
ParameterMode.IN);
		query.registerStoredProcedureParameter(2, Class.class, 
ParameterMode.REF_CURSOR);
		// set value to in params
		query.setParameter(1, "hyd");
		// get result
		List<Customer> list = query.getResultList();
		list.forEach(System.out::println);
```

Calling PL/SQL Procedure of Oracle that having Scalar query multiple column values) using EntityManager:

Step 1: Create PL/SQL procedure using SQL Developer.

Step 2: Write code in client app using EntityManager.

FEATCH_CUSTOMERS_DETAILS_BY_ADDS Procedure details

```
CREATE OR REPLACE PROCEDURE FEATCH_CUSTOMER_DEATILS_BY_ADD

(ADDR IN VARCHAR2,

DETAILS OUT SYS_REFCURSOR

) AS

BEGIN

   OPEN DETAILS FOR

         SELECT CNAME, BILL_AMOUNT FROM CUSTOMER WHERE
CADD=ADDR;

END FEATCH_CUSTOMER_DEATILS_BY_ADD;
```

SpringDataProj07CallingProceduresApplication.java

```
		// Call IN, OUT param using EntityManagere (ORACLE SPECIFIC
COLUMN)
		// get EntityManager object
		EntityManager manager = ctx.getBean(EntityManager.class);
		// Create StoreProcedureQuery object
		StoredProcedureQuery query =
manager.createStoredProcedureQuery("FEATCH_CUSTOMER_DEATILS_BY_
ADD");
		// register params
		query.registerStoredProcedureParameter(1, String.class,
ParameterMode.IN);
		query.registerStoredProcedureParameter(2, Class.class,
ParameterMode.REF_CURSOR);
		// set value to in params
		query.setParameter(1, "hyd");
		// get result
		List<Object[]> list = query.getResultList();
		list.forEach(row-> {
			for (Object obj : row) {
				System.out.print(obj+" ");
			}
			System.out.println();
		});
```

Calling PL/SQL Procedure of Oracle that perform Authentication using EntityManager:

Step 1: Create DB table in Oracle having users and passwords.
Step 2: Create PL/SQL procedure having Authentication logic.
Step 3: Write following code in client app using EntityManager.

P_AUTHENTICATION Procedure details

```
CREATE OR REPLACE PROCEDURE P_AUTHENTICATION

(

  UNAME IN VARCHAR2

, PASS IN VARCHAR2

, RESULT OUT VARCHAR2

) AS

   CNT NUMBER(4);

BEGIN

    SELECT COUNT(*) INTO CNT FROM USERSINFO WHERE
USERNAME=UNAME AND PASSWORD=PASS;

    IF CNT <> 0 THEN

        RESULT:='VALID CREDENTIAL';

    ELSE

        RESULT:='INVALID CREDENTIAL';

    END IF;

END P_AUTHENTICATION;
```

USERSINFO table details

```
  CREATE TABLE "SYSTEM"."USERSINFO"

   ("USERNAME" VARCHAR2(20 BYTE) NOT NULL ENABLE,

        "PASSWORD" VARCHAR2(20 BYTE),

         CONSTRAINT "USERSINFO_PK" PRIMARY KEY ("USERNAME")
```

SpringDataProj07CallingProceduresApplication.java

```
                // Call Authentication Procedure using EntityManagere
(ORACLE)
                // get EntityManager object
                EntityManager manager = ctx.getBean(EntityManager.class);
                // Create StoreProcedureQuery object
                StoredProcedureQuery query =
manager.createStoredProcedureQuery("P_AUTHENTICATION");
                // register params
                query.registerStoredProcedureParameter(1, String.class,
ParameterMode.IN);
                query.registerStoredProcedureParameter(2, String.class,
ParameterMode.IN);
                query.registerStoredProcedureParameter(3, String.class,
ParameterMode.OUT);
                // set value to in params
                query.setParameter(1, "nimu");
                query.setParameter(2, "nimu@123");
                // get result
                String result = (String) query.getOutputParameterValue(3);
                System.out.println("Result : "+result);
```

Note: While working with @procedure we can have only one IN param and only one OUT param, so there are multiple limitations to use it.

- PL/SQL Procedure does not return a value where as PL/SQL Function returns a value.
- If PL/SQL Procedure want to return 10 outputs then we have to take 10 out params.
- If PL/SQL Function want to return 10 outputs then we have to take 9 out params and 1 return value.

Note:

- ✓ In Spring Data JPA there is no direct provision to call PL/SQL function, but we can add the plain JDBC code by unwrapping Session, JDBC connection, CallableStatement objects through EntityManager
- ✓ In this approach we need not to have Entity class and repository matching to the SQL queries of PL/SQL procedures or function.

Step 1: Keep PL/SQL function ready with having any link with your repository and entity class.
Step 2: Write the following code in client app using EntityManager.

- Here we have to go for the Service class for that you have to create com.nt.service package with CustomerMgmtService.java (I) and CustomerMgmtServiceImpl.java (IC) and place the following code in their respective file.

GET_CUSTOMERS_BY_ADDS Procedure details

```
create or replace FUNCTION FX_GET_EMP_DETAILS_BY_NO
( NO IN NUMBER
, NAME OUT VARCHAR2
, DESG OUT VARCHAR2
, DNO OUT NUMBER
) RETURN FLOAT AS
   BSAL FLOAT;
BEGIN
   SELECT ENAME, JOB, SAL, DEPTNO INTO NAME, DESG, BSAL, DNO FROM
EMP WHERE EMPNO=NO;
   RETURN BSAL;

 END FX_GET_EMP_DETAILS_BY_NO;
```

CustomerMgmtService.java

```
package com.nt.service;

public interface CustomerMgmtService {

        public void getEmployById(int no);

}
```

CustomerMgmtServiceImpl.java

```
package com.nt.service;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;

import javax.persistence.EntityManager;
import javax.transaction.Transactional;

import org.hibernate.Session;
import org.hibernate.jdbc.ReturningWork;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("custService")
@Transactional
public class CustomerMgmtServiceImpl implements CustomerMgmtService
{

        @Autowired
        private EntityManager manager;

        @Override
        public void getEmployById(int no) {
                //get seesion object
                Session ses = manager.unwrap(Session.class);
                float sal = ses.doReturningWork(new ReturningWork<Float>() {

                        @Override
                        public Float execute(Connection con) throws
SQLException {
                                //create CallableStatement
                                CallableStatement cs = con.prepareCall("{?=call
FX_GET_EMP_DETAILS_BY_NO(?,?,?,?)}");
                                //register Return, OUT params with JDBC types
                                cs.registerOutParameter(1, java.sql.Types.FLOAT);
                                cs.registerOutParameter(3,
java.sql.Types.VARCHAR);
```

```
                                        cs.registerOutParameter(4,
java.sql.Types.VARCHAR);
                                        cs.registerOutParameter(5,
java.sql.Types.INTEGER);
                                        //set value to IN param
                                        cs.setInt(2, no);
                                        //call PL/SQL function
                                        cs.execute();
                                        //gether resuts from OUT params and Return
Param
                                        System.out.println("Emp NAME :
"+cs.getString(3));
                                        System.out.println("EMP DESG : "+cs.getString(4));
                                        System.out.println("DEPT NO : "+cs.getInt(5));
                                        return cs.getFloat(1);
                              }

                    });
                    System.out.println("Emp Salary : "+sal);
          }
}
```

SpringDataProj07CallingProceduresApplication.java

```
                    service = ctx.getBean("custService",
          CustomerMgmtService.class);
                    service.getEmployById(7369);
```

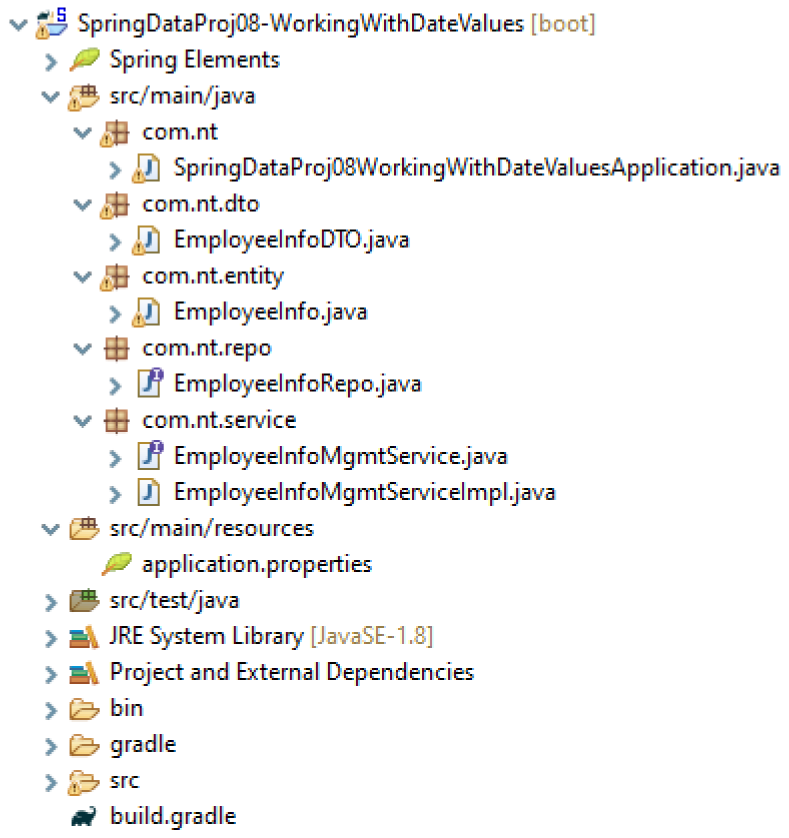
Working with Date vales

- @Temporal should be taken while working with java.util.Date property.
  @Temporal(type = TemporalType.DATE/ TIME/ TIMESTAMP)
- If you using Java 8 date and time API there is no need of taking @Temporal type.
  - LocalDate
  - LocalTime
  - LocalDateTime

Q. Can we interchange @Repository and @Service?

Ans. Yes, we can do but not recommended because @Service gives built-in Tx support. which is required in Service class, not in DAO. @Repository takes care Exception translation which is required in DAO but not in Service. (So not recommended to interchange).

Directory Structure of SpringDataProj08-WorkingWithDateValue:



```
SpringDataProj08-WorkingWithDateValues [boot]
  Spring Elements
  src/main/java
    com.nt
      SpringDataProj08WorkingWithDateValuesApplication.java
    com.nt.dto
      EmployeeInfoDTO.java
    com.nt.entity
      EmployeeInfo.java
    com.nt.repo
      EmployeeInfoRepo.java
    com.nt.service
      EmployeeInfoMgmtService.java
      EmployeeInfoMgmtServiceImpl.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure using Spring Starter Project option and package and classes also.
- During development use the following jars
  a. Lombok
  b. Spring Data JPA
  c. MySQL Driver
  d. Oracle Driver
- Add the following code in their respective files.

application.properties

```
#DataSource configuration details
#spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
#spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
#spring.datasource.username=scott
```

```
#spring.datasource.password=tiger
#spring.datasource.hikari.minimum-idle=10
#spring.datasource.hikari.maximum-pool-size=100

#MySQL
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///nshb413
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100

#JPA/ Hibernate properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDia
lect
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialec
t
```

EmployeeInfoDTO.java

```java
package com.nt.dto;

import java.io.Serializable;
import java.util.Date;

import lombok.Data;

@Data
public class EmployeeInfoDTO implements Serializable {

	private Integer eid;
	private String ename;
	private String eadd;
	private Date dob;
	private Date doj;
	private Date batchTime;

}
```

EmployeeInfo.java

```java
package com.nt.entity;

import java.io.Serializable;
import java.util.Date;

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

import org.hibernate.annotations.Type;

import lombok.Data;

@Data
@Entity
public class EmployeeInfo implements Serializable {
	@GeneratedValue(strategy = GenerationType.AUTO)
	@Id
	@Type(type = "int")
	private Integer eid;

	@Column(length = 20)
	@Type(type = "string")
	private String ename;

	@Column(length = 20)
	@Type(type = "string")
	private String eadd;

	@Temporal(value = TemporalType.TIMESTAMP)
	private Date dob;

	@Temporal(value = TemporalType.DATE)
	private Date doj;

	@Temporal(value = TemporalType.TIME)
	private Date batchTime;
}
```

EmployeeInfoRepo.java

```
package com.nt.repo;

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

import com.nt.entity.EmployeeInfo;

public interface EmployeeInfoRepo extends JpaRepository<EmployeeInfo,
Integer> {

}
```

EmployeeInfoMgmtService.java

```
package com.nt.service;

import com.nt.dto.EmployeeInfoDTO;

public interface EmployeeInfoMgmtService {

	public Integer registerEmployee(EmployeeInfoDTO dto);
	Iterable<EmployeeInfoDTO> getAllEmployeeInformation();

}
```

EmployeeInfoMgmtServiceImpl.java

```
package com.nt.service;

import java.util.ArrayList;

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

import com.nt.dto.EmployeeInfoDTO;
import com.nt.entity.EmployeeInfo;
import com.nt.repo.EmployeeInfoRepo;

@Service("empService")
public class EmployeeInfoMgmtServiceImpl implements
```

```
EmployeeInfoMgmtService {

        @Autowired
        private EmployeeInfoRepo empRepo;

        @Override
        public Integer registerEmployee(EmployeeInfoDTO dto) {
                EmployeeInfo entity = null;
                //Convert DTO to entity
                entity = new EmployeeInfo();
                BeanUtils.copyProperties(dto, entity);
                //use empRepo
                return empRepo.save(entity).getEid();
        }

        @Override
        public Iterable<EmployeeInfoDTO> getAllEmployeeInformation() {
                Iterable<EmployeeInfo> itEntity = null;
                Iterable<EmployeeInfoDTO> itDTO = new ArrayList<>();
                //use empReop
                itEntity = empRepo.findAll();
                //convert itEntity to itDTO
                itEntity.forEach(entity->{
                        EmployeeInfoDTO dto = new EmployeeInfoDTO();
                        BeanUtils.copyProperties(entity, dto);
                        ((ArrayList<EmployeeInfoDTO>) itDTO).add(dto);
                });
                return itDTO;
        }

}
```

SpringDataProj08WorkingWithDateValuesApplication.java

```
package com.nt;

import java.util.Date;

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

import com.nt.dto.EmployeeInfoDTO;
import com.nt.service.EmployeeInfoMgmtService;

@SpringBootApplication
public class SpringDataProj08WorkingWithDateValuesApplication {

        public static void main(String[] args) {
                ApplicationContext ctx = null;
                EmployeeInfoMgmtService service = null;
                EmployeeInfoDTO dto = null;
                //create Container
                ctx =
SpringApplication.run(SpringDataProj08WorkingWithDateValuesApplication
.class, args);
                //get Service object
                service = ctx.getBean("empService",
EmployeeInfoMgmtService.class);
                //invoke the method
                try {
                        //pepare DTO
                        dto = new EmployeeInfoDTO();
                        dto.setEname("Harish");
                        dto.setEadd("hyd");
                        dto.setDob(new ~~Date~~(90, 04, 06, 12, 35, 05));
                        dto.setDoj(new ~~Date~~(114, 07, 23));
                        dto.setBatchTime(new Date());
                        System.out.println("Register Employee Id :
"+service.registerEmployee(dto));
                } catch (Exception e) {
                        System.out.println("Problem in Employee Registration");
                        e.printStackTrace();
                }
                System.out.println("-------------------------------");
        service.getAllEmployeeInformation().forEach(System.out::println);

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

- Oracle does not support "time" datatype it supports only date, timestamp datatypes.
- MySQL does support date, time, datetime datatypes.

Using Java 8 Date and Time API

- Most of the methods in java.util.Date class are deprecated. So, it is recommended to use java.util.Calendar or Java8 Date & time API .

Directory Structure of SpringDataProj09-WorkingWithDateValues-Java8:

- Copy paste the SpringDataProj08-WorkingWithDateValues application and change rootProject.name to SpringDataProj09-WorkingWithDateValues-Java8 in settings.gradle file.
- Need not to add any new file same SpringDataProj08-WorkingWithDateValues.
- Change the SpringBootApplication class name to SpringDataProj09WorkingWithDateValuesJava8Application.
- Add the following code in their respective files.

EmployeeInfoDTO.java

```
package com.nt.dto;

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

import lombok.Data;

@Data
public class EmployeeInfoDTO implements Serializable {

	private Integer eid;
	private String ename;
	private String eadd;
	private LocalDateTime dob;
	private LocalDate doj;
	private LocalTime batchTime;

}
```

EmployeeInfo.java

```
package com.nt.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 org.hibernate.annotations.Type;

import lombok.Data;

@Data
@Entity
public class EmployeeInfo implements Serializable {

	@GeneratedValue(strategy = GenerationType.AUTO)
	@Id
	@Type(type = "int")
	private Integer eid;

	@Column(length = 20)
	@Type(type = "string")
	private String ename;

	@Column(length = 20)
	@Type(type = "string")
	private String eadd;

	private LocalDateTime dob;
	private LocalDate doj;
	private LocalTime batchTime;

}
```

SpringDataProj09WorkingWithDateValuesJava8Application.java

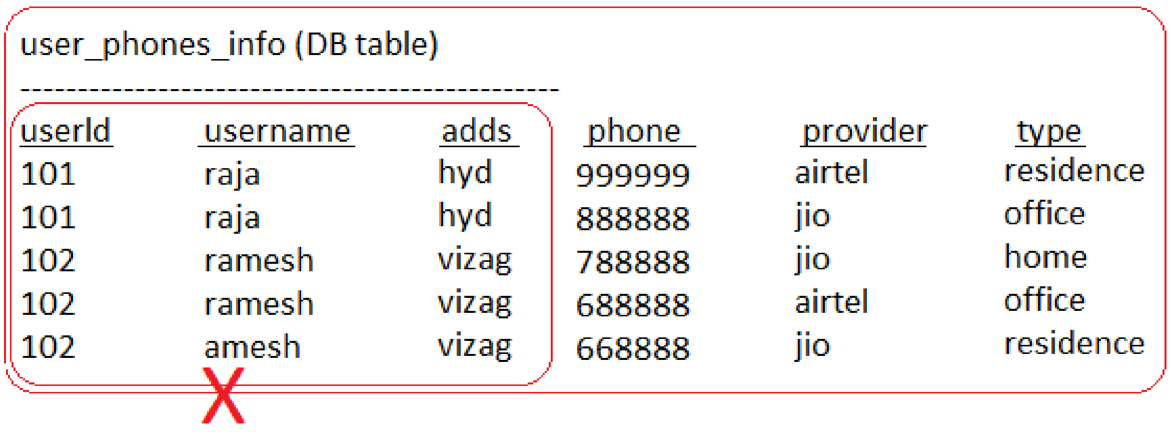
```
                try {
                        //pepare DTO
                        dto = new EmployeeInfoDTO();
                        dto.setEname("Harish");
                        dto.setEadd("hyd");
                        dto.setDob(LocalDateTime.of(1990, 03, 26, 12, 35, 05));
                        dto.setDoj(LocalDate.of(2015, 07, 23));
                        dto.setBatchTime(LocalTime.now());

                        System.out.println("Register Employee Id : 
"+service.registerEmployee(dto));
                } catch (Exception e) {
                        System.out.println("Problem in Employee Registration");
                        e.printStackTrace();
                }
                System.out.println("-------------------");
        service.getAllEmployeeInformation().forEach(System.out::println);
```

Association Mapping

- Association mapping is also known as Relationships and multiplicity.

Need:

- Keeping multiple entities/ parties' data in single DB table is bad practice because it gives,
    - Data Redundancy Problem (duplicate values may come).
    - Data Management Problem (managing huge amount columns in single DB table is bad).



user_phones_info (DB table)

----

userId	username	adds	phone	provider	type
101	raja	hyd	999999	airtel	residence
101	raja	hyd	888888	jio	office
102	ramesh	vizag	788888	jio	home
102	ramesh	vizag	688888	airtel	office
102	amesh	vizag	668888	jio	residence

X

(Data redundancy, management problems are here)

Solution: Take two entities data in two different DB tables and keep them relationship (always prefer adding foreign key column in child table).

DB tables having one to many relationship (One user having multiple phone numbers):

user_tab (Parent DB table)

userId(pk)	username	adds
101	raja	hyd
102	ramesh	vizag

phone_number_tab(child db table)

phone(pk)	provider	type	userId(FK)
999999	airtel	residence	101
888888	jio	office	101
788888	jio	home	102
688888	airtel	office	102
668888	jio	residence	102

- When DV table are in relationship/ multiplicity we need to keep the relevant Entity classes also in relationship by taking support composition [HAS-A] (parent object having one more objects of child).
    - One to One
    - Many to One

  Non-collection, based association (with single reference variables)

    - One to Many
    - Many to Many

  Collection based association (with collection of reference variables)

Association mapping:

|-> Uni-Directional: Parent to child or child to parent access is possible.
|-> Bi-Directional: Child to parent and parent to child access is possible.

- Using Single Foreign key column, we can build both Uni-Directional and Bi-Directional association between two DB tables.
- Java does not foreign key column concepts. So, we need build entity classes with relationship using References or Array or Collection of references. Moreover, we need design entity classes for Uni-Directional association, and Bi-Directional association differently.

One to Many Uni-Directional Association (Parent to Child) Entity classes:

```
@Entity            //Parent class
@Table(name="user_tab")
```

```
public class User {
            @Id
            @GenerateValue(strategy=AUTO)
            private Integer userId;
            private String name;
            private String addrs;
            @OneToMany(targetEntity=PhoneNumber.class, cascade-
            CascadeType.ALL)
            @JoinColumn(name="unid",referencedColumn=”userId")
            private Set<PhoneNumber> phones;
            //setters & getters
            ……………………
            ………….
}

@Entity             //Child class
@Table("Phone Numbers")
public class PhoneNumber {
            @Id
            private Long phone;
            private String type;
            private String provider;
            //setters && getters
            ………………..
            ……………
}
```

Note: Taking property for Foreign Key column in child class is optional.

Many to One or One to Many Uni-Directional Association Entity classes:

```
@Entity             //Parent class
@Table(name="user_tab")
public class User {
            @Id
            @GenerateValue(strategy=AUTO)
            private Integer userId;
            private String name;
            private String addrs;
            @OneToMany(targetEntity=PhoneNumber.class, cascade-
            CascadeType.ALL)
            @JoinColumn(name="unid",referencedColumn=”userId")
            private Set<PhoneNumber> phones;
            //setters & getters
            ……………………
            ………….
}

@Entity             //Child class
@Table("Phone Numbers")
public class PhoneNumber {
            @Id
            private Long phone;
            private String type;
            private String provider;
            @ManyToOne(targetEntity=User.class, cascade-CascadeType.ALL)
            @JoinColumn(name="unid",referencedColumn=”userId")
            private User parent;
            //setters && getters
            ………………..
            …………..
}
```

Note: cascade=CascadeType.ALL means any non-select persistence operation performed on the main object will reflect or cascade to associated objects.

Developing Using Spring Data:

a. Entity classes
    i. child class
    ii. parent class
b. Repository interfaces
    i. 1 for child class/ table
    ii. 1 for parent class/ table
c. Service Interface, Service Impl classes
    i. Inject both repositories
d. Develop application.properties
e. Develop client App and helper classes (DTO)

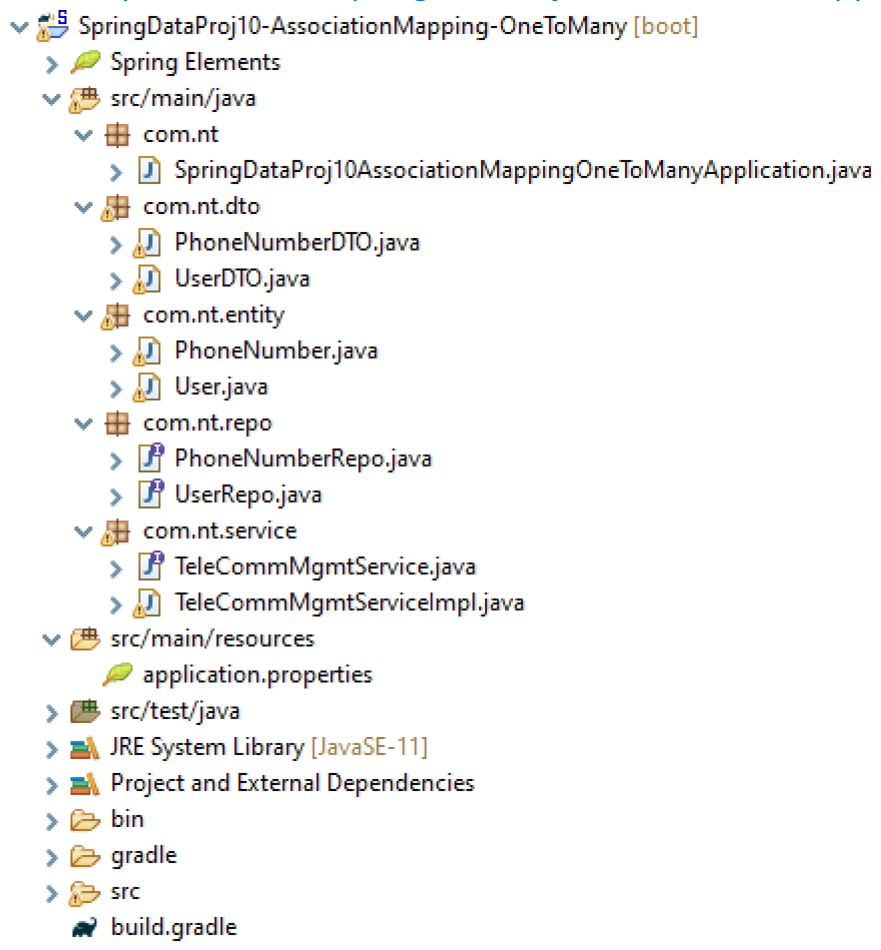
In general,

- 1 Entity class 1 DB table
- 1 Entity class 1 Repository Interface

- 1 object of entity class 1 record in DB table

Note: Be careful with BeanUtils.copyProperties(-,-) while dealing with inner collection properties.

Directory Structure of SpringDataProj10-AssociationMapping-OneToMany:



```
SpringDataProj10-AssociationMapping-OneToMany [boot]
  Spring Elements
  src/main/java
    com.nt
      SpringDataProj10AssociationMappingOneToManyApplication.java
    com.nt.dto
      PhoneNumberDTO.java
      UserDTO.java
    com.nt.entity
      PhoneNumber.java
      User.java
    com.nt.repo
      PhoneNumberRepo.java
      UserRepo.java
    com.nt.service
      TeleCommMgmtService.java
      TeleCommMgmtServiceImpl.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-11]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure using Spring Starter Project option and package and classes also.
- During development use the following jars
  a. Lombok
  b. Spring Data JPA
  c. MySQL Driver
  d. Oracle Driver
- Add the following code in their respective files.
- Copy and paste the application.properties from previous project because we are using same setup.

Use java higher version, to use all features of different java versions.

UserDTO.java

```java
package com.nt.dto;

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

import lombok.Data;

@Data
public class UserDTO implements Serializable {
	private Integer userId;
	private String userName;
	private String address;
	private Set<PhoneNumberDTO> phones;
}
```

PhoneNumberDTO.java

```java
package com.nt.dto;

import java.io.Serializable;

import lombok.Data;

@Data
public class PhoneNumberDTO implements Serializable {
	private long mobileNo;
	private String type;
	private String provider;
}
```

PhoneNumber.java

```java
package com.nt.entity;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import org.hibernate.annotations.Type;

import lombok.Data;

@Entity
@Table(name="DATA_PHONENUMBER")
@Data
public class PhoneNumber implements Serializable {

	@Id
	private long mobileNo;

	@Column(length = 10)
	@Type(type = "string")
	private String type;

	@Column(length = 10)
	@Type(type = "string")
	private String provider;

}
```

User.java

```
package com.nt.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.JoinColumn;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import org.hibernate.annotations.Type;
import lombok.Data;

@Entity
@Table(name = "DATA_USER")
@Data
public class User implements Serializable {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Type(type = "int")
        private Integer userId;

        @Column(length = 20)
        @Type(type = "string")
        private String userName;

        @Column(length = 20)
        @Type(type = "string")
        private String address;

        @OneToMany(targetEntity = PhoneNumber.class, cascade =
CascadeType.ALL)
        @JoinColumn(name = "fk_userId", referencedColumnName =
"userId")
        private Set<PhoneNumber> phones;

}
```

UserRepo.java

```
package com.nt.repo;

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

import com.nt.entity.User;

public interface UserRepo extends JpaRepository<User, Integer> {

}
```

PhoneNumberRepo.java

```
package com.nt.repo;

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

import com.nt.entity.PhoneNumber;

public interface PhoneNumberRepo extends JpaRepository<PhoneNumber,
Long> {

}
```

TeleCommMgmtService.java

```
package com.nt.service;

import com.nt.dto.UserDTO;

public interface TeleCommMgmtService {

        public String registerCustomer(UserDTO userDTO);
}
```

TeleCommMgmtServiceImpl.java

```
package com.nt.service;

import java.util.HashSet;
import java.util.Set;

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

import com.nt.dto.PhoneNumberDTO;
import com.nt.dto.UserDTO;
import com.nt.entity.PhoneNumber;
import com.nt.entity.User;
import com.nt.repo.PhoneNumberRepo;
import com.nt.repo.UserRepo;
```

```
@Service("teleComService")
public class TeleCommMgmtServiceImpl implements
TeleCommMgmtService {

	@Autowired
	private UserRepo userRepo;
	@Autowired
	private PhoneNumberRepo phnoRepo;

	@Override
	public String registerCustomer(UserDTO userDTO) {
		User userEntity = null;
		Set<PhoneNumberDTO> childDTO = null;
		Set<PhoneNumber> childEntity = new HashSet<>();
		//convert userDTO to userEntity
		userEntity = new User();
		BeanUtils.copyProperties(userDTO, userEntity);
		childDTO = userDTO.getPhones();
		childDTO.forEach(phDTO -> {
			PhoneNumber phEntity = new PhoneNumber();
			BeanUtils.copyProperties(phDTO, phEntity);
			childEntity.add(phEntity);
		});
		userEntity.setPhones(childEntity);
		return "Customer is registered having User Id :
"+userRepo.save(userEntity).getUserId();
	}

}
```

TeleCommMgmtService.java

```
package com.nt;

import java.util.Set;

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

import com.nt.dto.PhoneNumberDTO;
import com.nt.dto.UserDTO;
import com.nt.service.TeleCommMgmtService;

@SpringBootApplication
public class SpringDataProj10AssociationMappingOneToManyApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		TeleCommMgmtService service = null;
		PhoneNumberDTO phDTO1 = null, phDTO2 = null;
		UserDTO userDTO = null;
		///create container
		ctx =
SpringApplication.run(SpringDataProj10AssociationMappingOneToManyAp
plication.class, args);
		//get Service class object
		service
=ctx.getBean("teleComService",TeleCommMgmtService.class);
		//child object
		phDTO1 = new PhoneNumberDTO();
		phDTO1.setMobileNo(8018149478L);
		phDTO1.setType("residence");
		phDTO1.setProvider("airtel");

		phDTO2 = new PhoneNumberDTO();
		phDTO2.setMobileNo(9337043730L);
		phDTO2.setType("Personal");
		phDTO2.setProvider("jio");
		//parent Object
		userDTO = new UserDTO();
		userDTO.setUserName("Harish");
		userDTO.setAddress("UP");
		userDTO.setPhones(Set.of(phDTO1, phDTO2));
		//invoke the method
		try {
			System.out.println(service.registerCustomer(userDTO));
		} catch (DataAccessException dae) {
			dae.printStackTrace();
		}
	}
}
```

JOINS

- Join are to collect data from two DB tables having implicit conditions and also allow to some explicit conditions.
- To work SQL joins we need not to keep DB tables in relationship but to work with JPQL/ HQL joins two DB tables and their relevant Entity classes must be there in relationship.
- HQL/ JPQL supports 4 types of joins
  a. inner join --> gives common data of both DB tables.
  b. left join/ left outer join --> gives common data and also gives uncommon data of left side DB table.
  c. right join /right outer join gives common data and also gives uncommon data of right-side DB table.
  d. Full Join gives common and un common data of both DB tables.

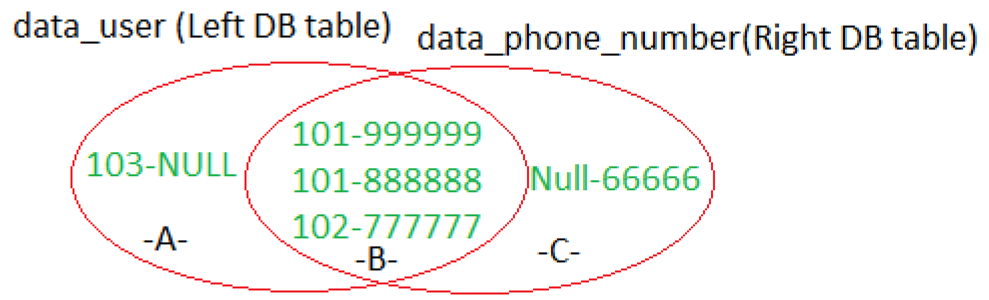
data _ user (Parent DB table)

userId (pk)	name	addrs
101	raja	hyd
102	rarnesh	viza
103	suresh	delhi

data_phone_number(Child DB table)

mobileNo	type	provider	fk_userId
999999	residen	airtel	101
888888	person	jio	101
777777	home	jio	102
666666	office	vodafon	

Venn diagram:



Inner join: -B-
Left join/ left outer join: -B- + -A-
Right join/ right outer join: -B- + -C-
Full join: -B- + -A- + -C-

Parent to child syntax:

```
select p.<property(s)>, c. <property(s)> from <parent entity> p
        [Join type]
        p.<Has-A variable in parent entity> c where <condition>
```

Note: [Join type] inner join (or) join / left join or left outer join / right join or right outer join/ full join.

Child to parent syntax:

select c.<property(s)>, p.<property(s)> from <child Entity> c
            [Join type]
            c.<has a variable in child entity> p where <condition>

Directory Structure of SpringDataProj11-JPQL-Joins:

- Copy paste the SpringDataProj10-AssociationMapping-OneToMany application and change rootProject.name to SpringDataProj11-JPQL-Joins in settings.gradle file.
- Need not to add any new file same SpringDataProj10-AssociationMapping-OneToMany
- Change the SpringBootApplication class name to SpringDataProj11JPQLJoinsApplication.
- Add the following code in their respective files.
- Make sure you have some common and uncommon data in both the table.

UserRepo.java

```
	//@Query("SELECT p.userId, p.userName, c.mobileNo, c.provider FROM User p INNER JOIN p.phones c")
	//@Query("SELECT p.userId, p.userName, c.mobileNo, c.provider FROM User p LEFT JOIN p.phones c")
	//@Query("SELECT p.userId, p.userName, c.mobileNo, c.provider FROM User p RIGHT JOIN p.phones c")
	@Query("SELECT p.userId, p.userName, c.mobileNo, c.provider FROM User p FULL JOIN p.phones c")
	List<Object[]> fetchDataByJoin();

	@Query("SELECT p.userId, p.userName, c.mobileNo, c.provider FROM User p FULL JOIN p.phones c WHERE p.address=?1")
	List<Object[]> fetchDataByJoinUsingAddress(String address);
```

TeleCommMgmtService.java

```
	public List<Object[]> getDataByJoin();

	public List<Object[]> getDataByJoinUsingAddress(String address);
```

TeleCommMgmtServiceImpl.java

```
	@Override
	public List<Object[]> getDataByJoin() {
		//use userREpo
		return userRepo.fetchDataByJoin();
	}

	@Override
	public List<Object[]> getDataByJoinUsingAddress(String address) {
		//use userREpo
		return userRepo.fetchDataByJoinUsingAddress(address);
	}
```

SpringDataProj11JPQLJoinsApplication.java

```
		//get Service class object
		service
=ctx.getBean("teleComService",TeleCommMgmtService.class);
		service.getDataByJoin().forEach(row->{
			for (Object val: row)
				System.out.print(val+" ");
			System.out.println();
		});
		System.out.println("-----------------------");
		service.getDataByJoinUsingAddress("MP").forEach(row->{
			for (Object val: row)
				System.out.print(val+" ");
			System.out.println();
		});
```

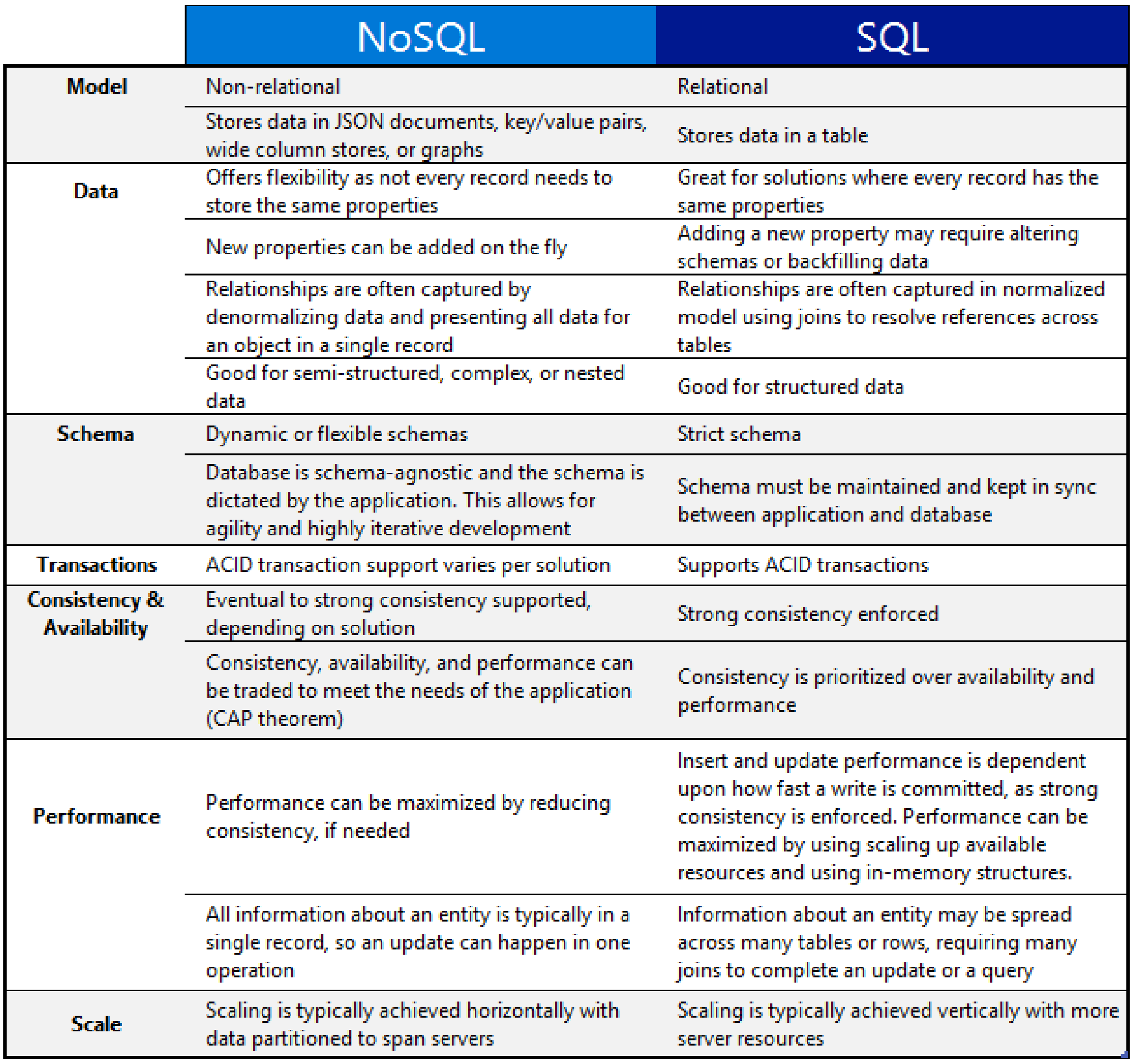
Working with MongoDB

- If data is structured data having fixed schema, then go for SQL DB s/w.
- If data is unstructured data ...having dynamic schema, then go for No-SQL DB s/w.

```
MongoDB (Physical No SQL DB s/w)
        |--> Logical DB1
                |--> Collection (Like DB table)
                        |--> {"<key>":<value>, ....} (Document)
```

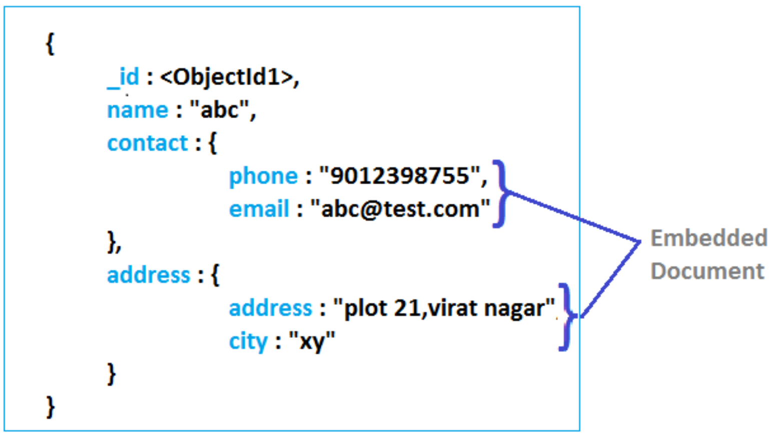
Note: MongoDB stores data in the form of BSON format. Binary JSON (JSON with more data type).

Difference between NOSQL and SQL:

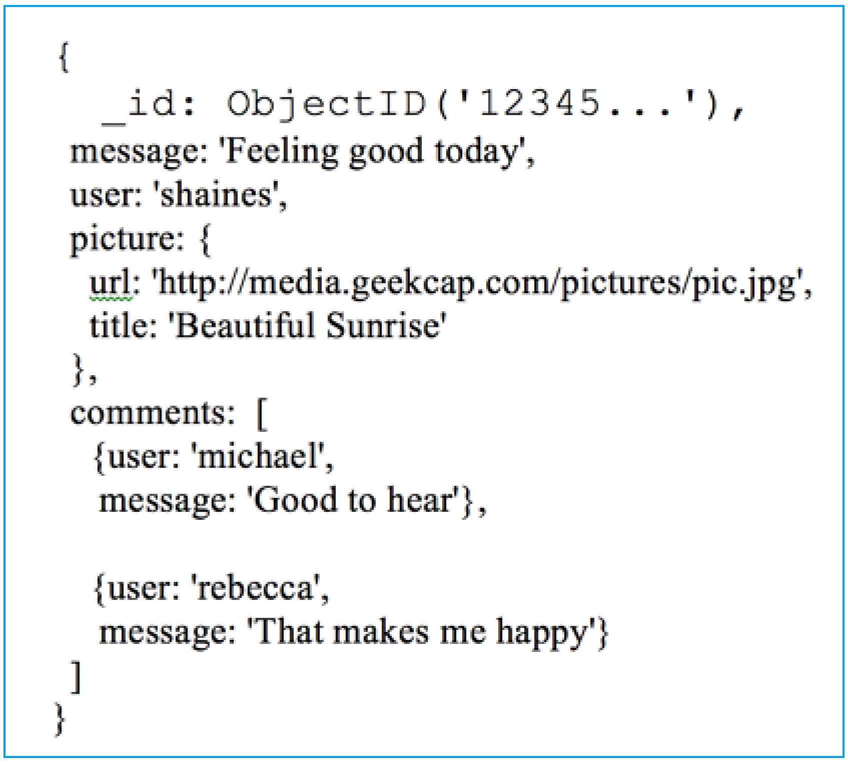


	NoSQL	SQL
Model	Non-relational	Relational
	Stores data in JSON documents, key/value pairs, wide column stores, or graphs	Stores data in a table
Data	Offers flexibility as not every record needs to store the same properties	Great for solutions where every record has the same properties
	New properties can be added on the fly	Adding a new property may require altering schemas or backfilling data
	Relationships are often captured by denormalizing data and presenting all data for an object in a single record	Relationships are often captured in normalized model using joins to resolve references across tables
	Good for semi-structured, complex, or nested data	Good for structured data
Schema	Dynamic or flexible schemas	Strict schema
	Database is schema-agnostic and the schema is dictated by the application. This allows for agility and highly iterative development	Schema must be maintained and kept in sync between application and database
Transactions	ACID transaction support varies per solution	Supports ACID transactions
Consistency & Availability	Eventual to strong consistency supported, depending on solution	Strong consistency enforced
	Consistency, availability, and performance can be traded to meet the needs of the application (CAP theorem)	Consistency is prioritized over availability and performance
Performance	Performance can be maximized by reducing consistency, if needed	Insert and update performance is dependent upon how fast a write is committed, as strong consistency is enforced. Performance can be maximized by using scaling up available resources and using in-memory structures.
	All information about an entity is typically in a single record, so an update can happen in one operation	Information about an entity may be spread across many tables or rows, requiring many joins to complete an update or a query
Scale	Scaling is typically achieved horizontally with data partitioned to span servers	Scaling is typically achieved vertically with more server resources

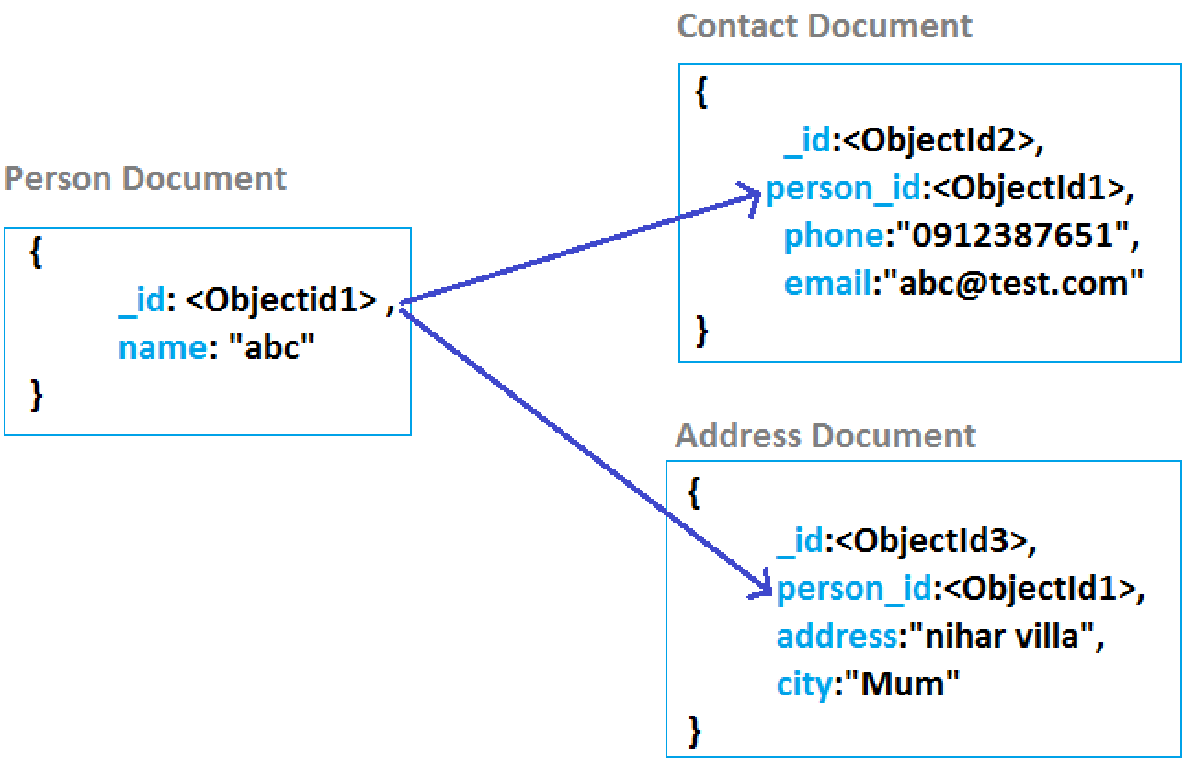
Examples of MongoDB Documents:



```
{
        _id : <ObjectId1>,
        name : "abc",
        contact : {
                phone : "9012398755",
                email : "abc@test.com"
        },
        address : {
                address : "plot 21,virat nagar"
                city : "xy"
        }
}
```



```
{
   _id: ObjectID('12345...'),
 message: 'Feeling good today',
 user: 'shaines',
 picture: {
   url: 'http://media.geekcap.com/pictures/pic.jpg',
   title: 'Beautiful Sunrise'
 },
 comments:  [
   {user: 'michael',
    message: 'Good to hear'},

   {user: 'rebecca',
    message: 'That makes me happy'}
  ]
}
```



Person Document

```
{
        _id: <Objectid1> ,
        name: "abc"
}
```

Contact Document

```
{
        _id:<ObjectId2>,
      person_id:<ObjectId1>,
        phone:"0912387651",
        email:"abc@test.com"
}
```

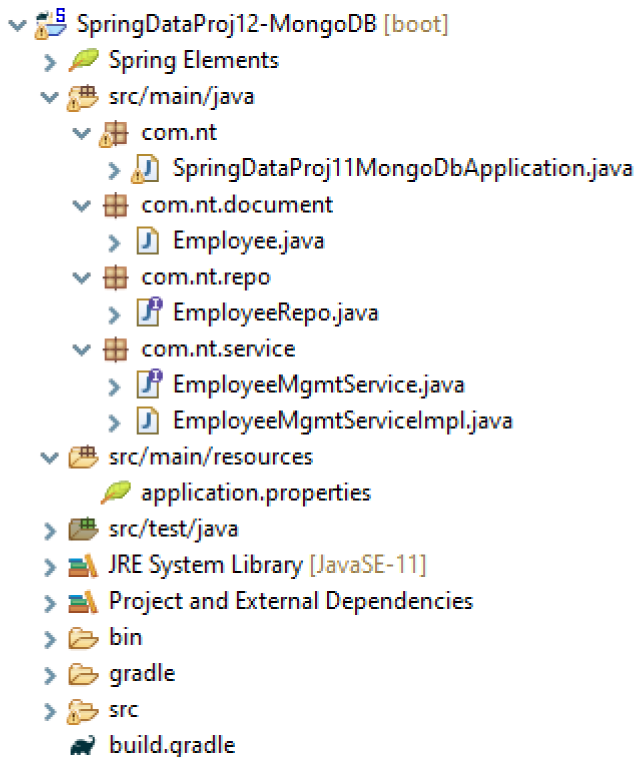
Address Document

```
{
        _id:<ObjectId3>,
        person_id:<ObjectId1>,
        address:"nihar villa",
        city:"Mum"
}
```

Procedure to create Logical DB having collection with docs in MongoDB:
Step #1: Install MongoDB [Download]
Step #2: Open MongoDB Compass.
Step #3: For new Connection Click on Fill in connection fields individually option.

Step #4: Leave all the things default then click on CONNECT.
Step #5: Now you can see your connection has established and then default database has come here, for create a new Database click on the CREATE DATABASE option.
Step #6: Give the Database Name and you have to create the Collection as well so you have to pass a Collection name during create the Database, then click on CREATE DATABASE.
Step #7: Now you can see your Database click on that.
Step #8: Then click on you Collection.
Step #9: There you get a Button ADD DATA, click on that then you will get two option click on Insert Document.
Step #10: Then you will get the following type of console there you can fill you details like below format and you should follow the format otherwise you will get error. After fill the all data click on INSERT button.
Step #11: Then you can see you inserted data.

Directory Structure of SpringDataProj10-AssociationMapping-OneToMany:



```
SpringDataProj12-MongoDB [boot]
  Spring Elements
  src/main/java
    com.nt
      SpringDataProj11MongoDbApplication.java
    com.nt.document
      Employee.java
    com.nt.repo
      EmployeeRepo.java
    com.nt.service
      EmployeeMgmtService.java
      EmployeeMgmtServiceImpl.java
  src/main/resources
    application.properties
  src/test/java
  JRE System Library [JavaSE-11]
  Project and External Dependencies
  bin
  gradle
  src
  build.gradle
```

- Develop the above directory structure using Spring Starter Project option and package and classes also.
- During development use the following jars
    a. Lombok
    b. Spring Data MongoDB
- Add the following code in their respective files.

application.properties

```
#To connect MongoDB
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=nitmgdblocal
```

Employee.java

```
package com.nt.document;

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

import lombok.Data;

@Document
@Data
public class Employee {

	@Id
	private Integer eid;
	private String ename;
	private String eadd;
	private Double salary;
	private String company;
	private String[] oldCompanies;

}
```

EmployeeRepo.java

```
package com.nt.repo;

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

import com.nt.document.Employee;

public interface EmployeeRepo extends MongoRepository<Employee,
Integer> {
}
```

EmployeeMgmtService.java

```java
package com.nt.service;

import java.util.List;

import com.nt.document.Employee;

public interface EmployeeMgmtService {

	public String registerEmployee(Employee doc);

}
```

EmployeeMgmtServiceImpl.java

```java
package com.nt.service;

import java.util.List;
import java.util.Optional;

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

import com.nt.document.Employee;
import com.nt.repo.EmployeeRepo;

@Service("empService")
public class EmployeeMgmtServiceImpl implements EmployeeMgmtService
{

	@Autowired
	private EmployeeRepo empRepo;

	@Override
	public String registerEmployee(Employee doc) {
		//use empRepo
		return "Document is saved with Id :
"+empRepo.save(doc).getEid();
	}
}
```

SpringDataProj11MongoDbApplication.java

```
package com.nt;

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

import com.nt.document.Employee;
import com.nt.service.EmployeeMgmtService;

@SpringBootApplication
public class SpringDataProj11MongoDbApplication {

	public static void main(String[] args) {
		ApplicationContext ctx = null;
		EmployeeMgmtService service = null;
		Employee doc1 = null;
		//get Container
		ctx = 
SpringApplication.run(SpringDataProj11MongoDbApplication.class, args);
		//get Service object
		service = ctx.getBean("empService", 
EmployeeMgmtService.class);
		//invoke the method
		try {
			doc1 = new Employee();
			doc1.setEid(104); doc1.setEname("yadv");
			doc1.setEadd("delhi"); doc1.setSalary(565674.0);
			doc1.setCompany("Polarish");
			doc1.setOldCompanies(new String[] {"HCL", "DELL", 
"WIPRO"});
			System.out.println(service.registerEmployee(doc1));
		} catch (Exception e) {
			e.printStackTrace();
		}
		//close container
		((ConfigurableApplicationContext) ctx).close();
	}
}
```

Note:

- ✓ No Generators while working with MongoDB property in Document class will become "_id" key value in the real document and becomes identity value for document (record). it must be unique value.
- ✓ While creating spring boot project we need to select only MongoDB libraries other libraries like Spring data JPA are not required.
- ✓ Spring data provides unified environment to work with both SQL and No SQL DB s/w.

EmployeeMgmtService.java

```
    public List<Employee> findAllEmployees();
    public Employee findEmpById(int id);
    public String updateEmployeeSalary(int id, double bonus);
    public String removeEmployee(int id);
```

EmployeeMgmtServiceImpl.java

```
    @Override
    public List<Employee> findAllEmployees() {
        //use empRepo
        return empRepo.findAll();
    }

    @Override
    public Employee findEmpById(int id) {
        Optional<Employee> optional = null;
        optional = empRepo.findById(id);
        return optional.get();
    }

    @Override
    public String updateEmployeeSalary(int id, double bonus) {
        Optional<Employee> optional = null;
        Employee doc = null;
        optional = empRepo.findById(id);
        if (optional.isPresent()) {
            doc = optional.get();
            doc.setSalary(doc.getSalary()+bonus);
            return doc.getEname()+" your salary is hiked by
```

```
"+bonus+" new salary is : "+empRepo.save(doc).getSalary();
                }
                else
                        return "Employee record not found";
        }

        @Override
        public String removeEmployee(int id) {
                if (empRepo.findById(id).isPresent()) {
                        empRepo.delete(empRepo.findById(id).get());
                        return id+" Employee is deleted";
                }
                else
                        return "Employee record not found";
        }
```

SpringDataProj11MongoDbApplication.java

```
                System.out.println("---------------------");
                //service.findAllEmployees().forEach(System.out::println);
                System.out.println("------------------------");
                //System.out.println(service.findEmpById(102));
                System.out.println("------------------------");
                //System.out.println(service.updateEmployeeSalary(102,
1000));
                System.out.println("------------------------");
                System.out.println(service.removeEmployee(103));
```

-- The END --