



INDEX

Spring JDBC/ DAO -----

1. Introduction	<u>04</u>
2. JdbcTemplate	<u>08</u>
3. Callback Interface	<u>27</u>
a. RowMapper<T>	<u>28</u>
b. ResultSetExtractor<T>	<u>37</u>
c. RowCallbackHandler	<u>41</u>
4. NamedParameterJdbcTemplate	<u>44</u>
5. SimpleJdbcInsert or Call	<u>53</u>
a. SimpleJdbcInsert	<u>53</u>
b. SimpleJdbcCall	<u>59</u>
6. Mapping SQL Operations as Sub classes	<u>64</u>
7. Working with properties file and yml/ yaml files in Spring/ Spring Boot	<u>75</u>
a. YML/ YAML	<u>81</u>
8. Profiles in Spring or Spring Boot	<u>89</u>
9. Spring Boot Flow for Standalone application	<u>121</u>

Spring JDBC or DAO

Introduction

Spring JDBC (Also known as Spring DAO):

- It provides abstraction on plain JDBC Technology and simplifies JDBC style persistence logic development by avoiding boiler plate code.

Plain JDBC code (Java JDBC code):

- Load JDBC driver class (To register JDBC driver with Driver Manager Service)
 - Establish the connection
 - Create JDBC Statement object
- (Common logics)
- Send and execute SQL query
 - Gather results and process results (If necessary, iterate through RS)
- (Application specific logics)
- Perform exception handling
 - Perform TxMgmt (optional)
- (Common logics)
- Close JDBC objects (including JDBC connection)

Note:

- ✓ Common logics these are same in all JDBC apps (boilerplate code), Application specific logics will change based on the Db s/w we use.
- ✓ The code that repeats across the multiple parts of Project /application either with no changes or with minor changes is called boilerplate code.

Spring JDBC Code:

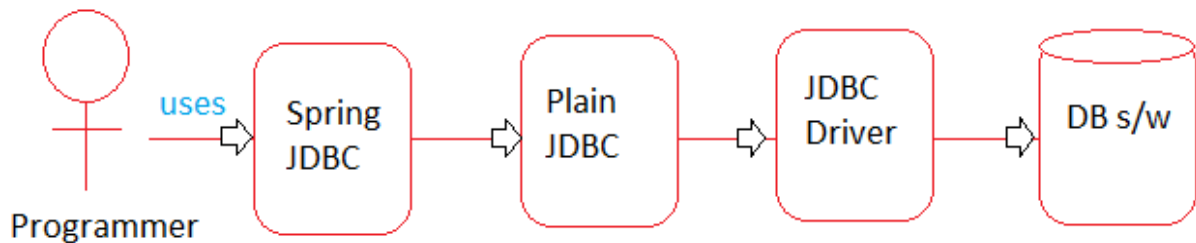
- Inject JdbcTemplate at class object having DataSource object (It internally takes care of boiler plate code (common logics)).
 - send and execute SQL query
 - Gather results and process results
- (Application specific logics)

Note: JdbcTemplate/ Spring JDBC is given based on Template Method Design Pattern. [This DP says provide template/ algorithm to perform series operation where common things will be taken care internally and specific things will be given to programmer to implement].

- Plain JDBC code = Java code + SQL queries

(DB s/w dependent Persistence logic because SQL queries are DB s/w dependent)

- Spring JDBC code = Spring code + JDBC code + SQL queries
(DB s/w dependent persistence logic)



Persistence: The process of saving and managing data for long time is called persistence.

Persistence store: The place where persistence takes place.

e.g. files, DB s/w (Best)

Persistence operations: insert, update, delete, select operations are called persistence operations, these are also called CURD/ CRUD operations

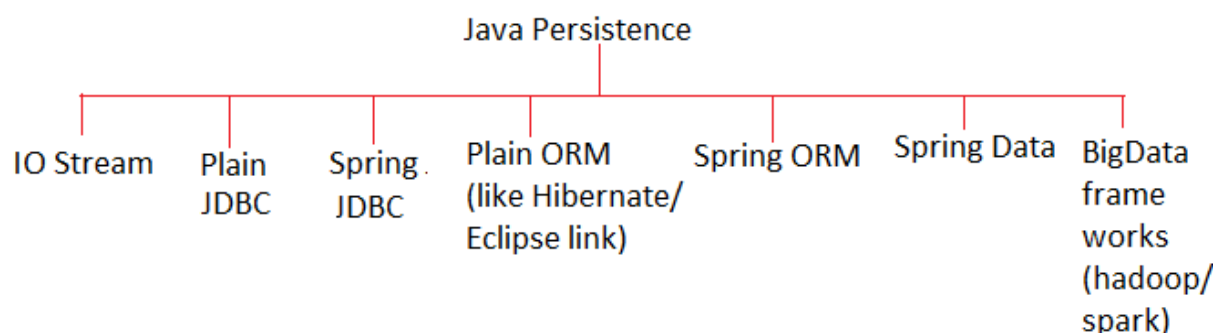
C -> create, U -> update, R -> Read, D->Delete

Persistence logic: The logic to perform curd operations

e.g. JDBC code, istreams code, hibernate code, Spring JDBC code and etc.

Q. Where should we use which Persistence API of Java?

Ans.



IO Streams:

- For small apps, small data like files.
- e.g. Desktop games, mobile games.

Plain JDBC and Spring JDBC:

- Large scale Apps getting huge amount data batch by batch for processing having DB s/w as persistence store.

- Here persistence logic is SQL queries, DB s/w dependent Persistence logic.
- e.g. Census App, e-commerce Apps

Plain ORM, Spring ORM and Spring Data JPA:

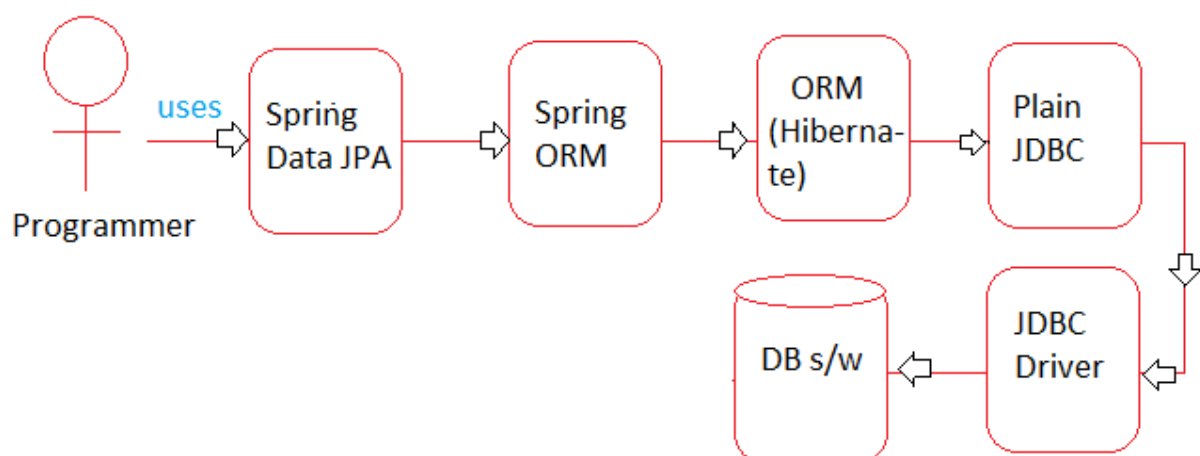
- For medium scale Apps that is having little amount of data to process at a time by taking DB s/w as persistence store.
- Here each record will be represented by 1 BO (java bean) class object.
- DB s/w independent persistence logic - we can take other advantages like caching, versioning, timestamping and etc.
- Here persistence logics are NoSQL queries.
- e.g. College Apps, University Apps, Job portal App, Hospital Apps and etc.

Bigdata Frameworks:

- If data is beyond storing and processing capacity of regular DB s/w then go for Bigdata Frameworks which says to store and process data by multiple ordinary computers called racks.
- e.g. Facebook data, Google data, YouTube data and etc.

Note:

- ✓ Spring Data module is having capability of generating 100% Basic CRUD operations code dynamically for the given DB tables.
- ✓ Spring JDBC internally uses plain JDBC and just simplifies JDBC style persistence logic i.e. (70% Spring JDBC will take care and 30% should be taken care by programmer).
- ✓ Spring ORM internally uses plain ORM and just simplifies ORM style objects-based persistence logic i.e. (70% Spring ORM will take care and 30% should be taken care by programmer)



Limitations with Plain JDBC:

- a. Uses DB s/w dependent SQL Queries in the development of Persistence logic, So the persistence logic DB s/w dependent.
- b. Supports only positional params (?) i.e. does not support named params.
- c. ResultSet object that represents the "SELECT SQL Query" execution is not Serializable object to send its data over the network.
- d. We need to write explicit logic to convert RS object records to different formats like List collection, Map collection, simple values and etc.
- e. Gives boiler plate code problem (i.e. we need write to common logics in every JDBC app).
- f. Throws SQLException which is checked Exception and limitations are,
 - i. For all problems of JDBC code same exception.
 - ii. We should explicitly catch and handle the exception.
 - iii. Does not support Exception Propagation naturally.
- g. Customization results is very complex.
and etc.

Spring JDBC advantages:

- a. Supports both positional (?) and named parameters.
- b. We can get "SELECT Query" Results in different formats directly with the support of query(), queryXxx() [queryForList(), queryForMap(), queryForObject() and etc.] methods.
- c. Customization results is bit easy with the support of Callback Interfaces.
- d. Provides abstraction on plain JDBC code and avoids the boiler plate code (common logics will be generated internally).
- e. Gives Detailed Exception classes hierarchy which is called DataAccessException classes hierarchy the advantages are,
 - i. These exceptions are unchecked exceptions.
 - ii. Exception handling is optional.
 - iii. Supports exception propagation by default.
 - iv. Raises different exceptions for different problems.
 - v. These are same exceptions for Spring JDBC, Spring ORM an Spring Data module.
 - vi. Spring JDBC internally uses Exception rethrowing concept to convert checked exceptions (SQLException) into Unchecked Exceptions (DataAccessException and its sub classes).

Note: The direct sub classes of java.lang.Exception class are called Checked Exception and The direct sub classes of java.lang.RuntimeException class are called Unchecked Exception.

JdbcTemplate class

```
public Object queryForObject(String query) throws DataAccessException{
    try{
        ..... //plain JDBC code
    }
    catch(SQLException se){
        throw new DataAccessException(se.getMessage());
    }
}
```

Exception rethrowing is happening here

- f. Simplifies the process of calling PL/SQL Procedures and functions.
- g. Gives great support to work with Generics, var args and etc. (Java5, 6 features).
- h. Allows to work with Java8, 9, 10 and etc. features.
- i. Can generate insert SQL query dynamically based on the given DB table name, column names and column values. and etc.

Note: Spring JDBC Persistence logic is still DB s/w dependent Persistence logic because its SQL queries-based Persistence logic.

Different Approaches of developing Persistence logic in Spring JDBC:

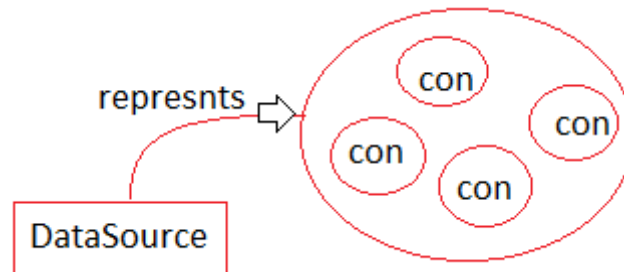
- a. Using JdbcTemplate
- b. Using NamedParameterJdbcTemplate
- c. Using SimpleJdbcTemplate (deprecated in Spring 4.x and removed in Spring 5.x)
- d. Using SimpleJdbcInsert, SimpleJdbcCall
- e. MappingSQLOperations as sub classes

JdbcTemplate

- ✚ It is central class/ API class for entire Spring JDBC i.e. remaining approaches of Spring JDBC programming internally uses this JdbcTemplate.
- ✚ Designed based on Template method design pattern which says define an algorithm to complete a task where common aspects will be taken care by Spring JDBC and lets the programmer to take care of only specific activities.
- ✚ Need DataSource obj as Dependent object.
- ✚ Gives query(-) and queryForXxx(-) for select query execution and gives update(-) method for non-select query execution.
- ✚ JdbcTemplate supports only Positional params.

Q. What is the JDBC connection pool/ DataSource that you used in Spring Project?

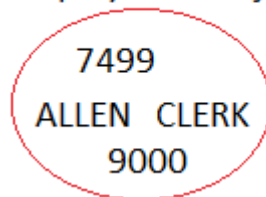
Ans. DataSource object represents JDBC connection pool i.e. all operations on JDBC connection pool can be done through DataSource object.



- If the project or application is standalone then use HikariCP(best) or Apache DBCP or c3p0 or ViburCP or TomcatCP and etc.
- If the project or application is web application and deployable in the server like Tomcat, WebLogic, glassfish and etc. then use Server managed JDBC connection pool like Tomcat managed JDBC connection pool, WebLogic JDBC connection pool and etc.

Different query (), queryForXxx() of JdbcTemplate:

1. To get Single value or single object use queryForObject()
e.g. 1: select count(*) from emp
e.g. 2: select ename from emp where empno=?
e.g. 3: select empno, ename, job, sal from emp where empno=?
to get these values of a record into Employee Bo class object
EmployeeBO object

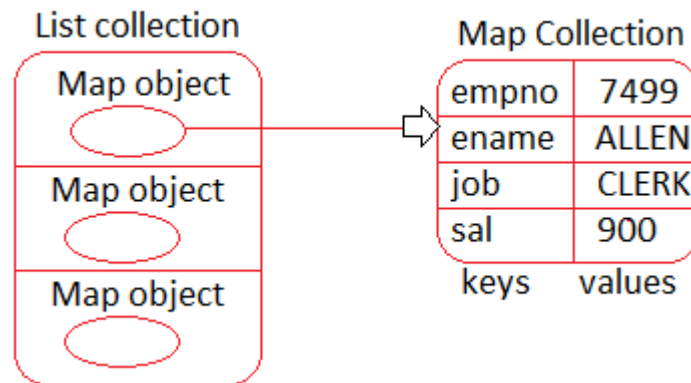


2. To get single record into Map Collection use queryForMap()
e.g. 1: select empno, ename, job, sal from emp where empno=?

Map Collection

empno	7499
ename	ALLEN
job	CLERK
sal	900
keys	values

3. To get multiple records into List Collection use queryForList() method
e.g. 1: select empno, ename, job, sal from emp where job=?



We can use JdbcTemplate in two ways:

1. Using Direct methods without callback interfaces
 - To get results as given by methods like object, Map, list and etc.
2. Using methods having callback interfaces
 - To get customize results as needed by writing partial JDBC code by utilizing the internally create JDBC objects.

JdbcTemplate Example:

[persistence-beans.xml](#)

```
<beans>
    <bean id="hkDs" class="pgk.HikariDataSource">
        .....
    </bean>

    <bean id="template" class="pkg.JdbcTemplate">
        <constructor-arg ref="hkDs"/>
    </bean>
    <bean id="empDAO" class="pkg.EmployeeDAOImpl">
        <constructor-arg ref="template"/>
    </bean>
</beans>
```

[applicationContext.xml](#)

```
<import resource="persistence-beans.xml"/>
<import resource="service-beans.xml"/>
```

```
public class EmployeeDAOImpl implements EmployeeDAO {
    private JdbcTemplate jt;
```

```

    public EmployeeDAOImpl(JdbcTemplate jt){
        this.jt=jt;
    }
    //methods with persistnece logic
    .....
}

```

Client App ----> Service class ----> DAO class ----> DB s/w

(Presentation logic) (b.logic) (Persistence logic)

DS
/ JdbcTemplate
/

e.g. JdbcTemplateTest --> EmployeeMgmtServiceImpl --> EmployeeDAOImpl --> DB

```
int count = jt.queryForObject("SELECT COUNT (*) FROM EMP, Integer.class);
```

- queryForObject(-) gets the Injected DS from jt -> DS collects one JDBC connection object from JDBC connection pool -> creates PS (PreparedStatement object) having given SQL query as pre-compiled SQL query -> executes query using ps.executeQuery() and get RS (ResultSet) object -> calls rs.next() and rs.getInt(1) method to get result as int value because of required type Integer.class -> gives result to DAO method -> DAO method gives to the caller service class method.

Directory Structure of DAOProj01-JdbcTemplate-DirectMethods:

```

v DAOProj01-JdbcTemplate-DirectMethods
  > Spring Elements
  v src/main/java
    v com.nt.cfgs
      applicationContext.xml
      persistence-beans.xml
      service-beans.xml
    v com.nt.dao
      EmployeeDAOImpl.java
      IEmployeeDAO.java
    v com.nt.service
      EmployeeMgmtService.java
      IEmployeeMgmtService.java
    v com.nt.test
      JdbcTemplateTest.java
  > src/test/java
  > JRE System Library [JavaSE-1.8]
  > Maven Dependencies
  > src
  > target
  pom.xml

```

- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file.
- Change to Java 1.8 or its higher version.

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>5.2.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.2.9.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>com.oracle.database.jdbc</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0.4</version>
  </dependency>
  <dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>3.4.5</version>
  </dependency>
</dependencies>
```

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

  <import resource="persistence-beans.xml"/>
  <import resource="service-beans.xml"/>

</beans>
```

persistence-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- HikariCP DS configuration -->
    <bean id="hkDs" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver"/>
        <property name="jdbcUrl"
value="jdbc:oracle:thin:@localhost:1521:xe"/>
        <property name="username" value="system"/>
        <property name="password" value="manager"/>
        <property name="minimumIdle" value="10"/>
        <property name="maximumPoolSize" value="30"/>
    </bean>

    <!-- JdbcTemplate configuration -->
    <bean id="template"
class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="hkDs"/>
    </bean>

    <!-- DAO configuration -->
    <bean id="empDAO" class="com.nt.dao.EmployeeDAOImpl">
        <constructor-arg ref="template"/>
    </bean>
</beans>
```

service-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- Service configuration -->
    <bean id="empService"
class="com.nt.service.EmployeeMgmtService">
        <constructor-arg ref="empDAO"/>
    </bean>
</beans>
```

IEmployeeDAO.java

```
package com.nt.dao;

public interface IEmployeeDAO {
    public int getEmployeeCount();
}
```

EmployeeDAOImpl.java

```
package com.nt.dao;

import org.springframework.jdbc.core.JdbcTemplate;

public class EmployeeDAOImpl implements IEmployeeDAO {

    private static final String GET_EMPLOYEES_COUNT = "SELECT
COUNT(*) FROM EMP";

    private JdbcTemplate jt;

    public EmployeeDAOImpl(JdbcTemplate jt) {
        this.jt = jt;
    }

    @Override
    public int getEmployeeCount() {
        int count = 0;
        count = jt.queryForObject(GET_EMPLOYEES_COUNT,
Integer.class);
        return count;
    }
}
```

IEmployeeMgmtService.java

```
package com.nt.service;

public interface IEmployeeMgmtService {
    public int fetchEmpsCount();
}
```

EmployeeMgmtServiceImpl.java

```
package com.nt.service;

import com.nt.dao.IEmployeeDAO;
```

```

public class EmployeeMgmtService implements IEmployeeMgmtService {

    private IEmployeeDAO dao;

    public EmployeeMgmtService(IEmployeeDAO dao) {
        super();
        this.dao = dao;
    }

    @Override
    public int fetchEmpsCount() {
        //use dao
        return dao.getEmployeeCount();
    }

}

```

IEmployeeMgmtService.java

```

package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;

import com.nt.service.IEmployeeMgmtService;

public class JdbcTemplateTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        IEmployeeMgmtService service = null;
        //create IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get service class object
        service = ctx.getBean("empService",
IEmployeeMgmtService.class);
        //invoke method
        try {
            System.out.println("Employee count is :
"+service.fetchEmpsCount());
        }
    }
}

```

```

        catch (DataAccessException dae) {
            dae.printStackTrace();
        }
        ((AbstractApplicationContext) ctx).close()
    }
}

```

Note: JdbcTemplate internally uses SimpleStatement object to execute the given SQL if the query is not having any positional (?) params otherwise it uses PreparedStatement object.

```
int count = jt.queryForObject("SELET COUNT(*) FROM EMP", Integer.class);
```

It internally uses SimpleStatement object (static query)

```
String name = jt.queryForObject("SELECT ENAME FROM EMP WHERE EMPNO=?", String.class, eno);
```

Supplies query param values
It internally uses PreparedStatement obj because the SQL query is dynamic SQL query (query with parameter)

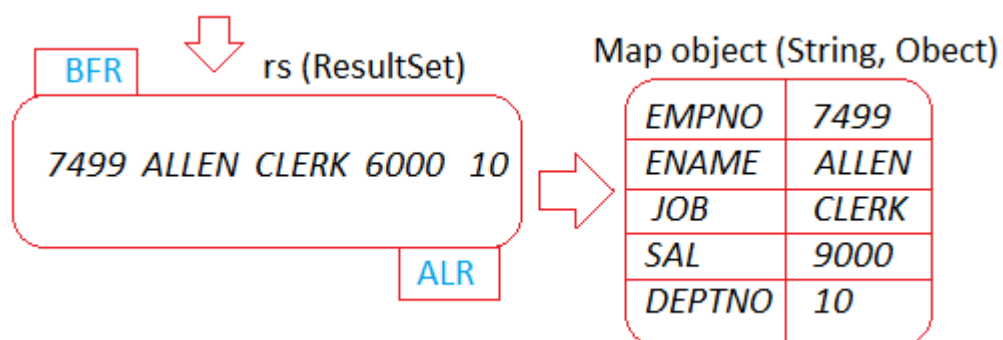
@Override

```

public Map<String, Object> getEmployeeDetailsByNo(int eno) {
    Map<String, Object> map = null;
    map = jt.queryForMap("SELECT EMPNO, ENAME, JOB, SAL,
                        DEPTNO FROM EMP WHERE EMPNO=?", eno);

    return map;
}

```



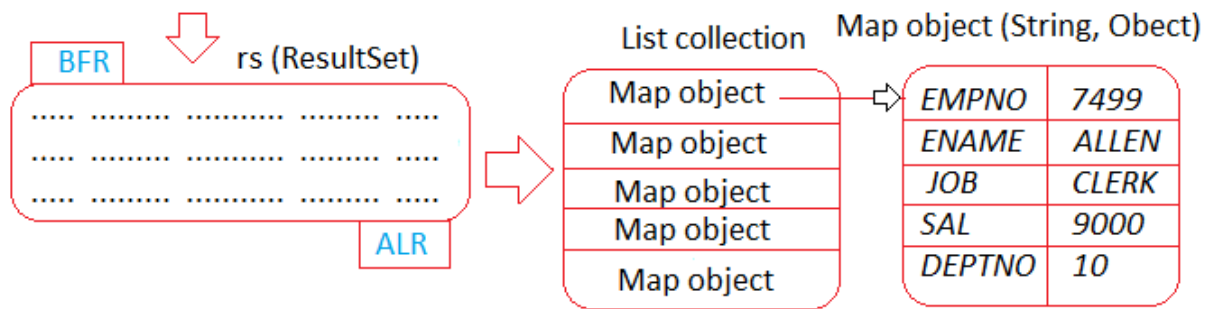
@Override

```

public List<Map<String, Object>> getEmployeesDetailsByDesg(String
                                desg1, String desg2) {

    return jt.queryForList("SELECT EMPNO, ENAME, SAL, JOB,
                        DEPTNO FROM EMP WHERE JOB IN (?, ?) ORDER BY JOB",
                        desg1, desg2);
}

```

IEmployeeDAO.java

```

public String getEmployeeNameByNo(int eno);
public Map<String, Object> getEmployeeDetailsByNo(int eno);
public List<Map<String, Object>> getEmployeesDetailsByDesg(String
desg1, String desg2);
public int insert(String name, String desg, float sal);
public int addBonusToEmployeeByDesg(String desg, float bonus);

```

EmployeeDAOImpl.java

```

private static final String GET_EMPLOYEE_NAME_BY_ID = "SELECT ENAME
FROM EMP WHERE EMPNO=?";
private static final String GET_EMPLOYEE_DETAILS_BY_ID = "SELECT
EMPNO, ENAME, SAL, JOB, DEPTNO FROM EMP WHERE EMPNO=?";
private static final String GET_EMPLOYEES_DETAILS_BY_JOB =
"SELECT EMPNO, ENAME, SAL, JOB, DEPTNO FROM EMP WHERE JOB IN (?,
?) ORDER BY JOB";
private static final String INSERT_EMPLOYEE = "INSERT INTO EMP
(EMPNO, ENAME, JOB, SAL) VALUES (ENO_SEQ.NEXTVAL, ?, ?, ?)";
private static final String ADD_BONUS_BY_JOB = "UPDATE EMP SET
SAL=SAL+? WHERE JOB=?";

```

```

@Override
public String getEmployeeNameByNo(int eno) {
    String name = null;
    name = jt.queryForObject(GET_EMPLOYEE_NAME_BY_ID,
String.class, eno);
    return name;
}

```

```

@Override
public Map<String, Object> getEmployeeDetailsByNo(int eno) {
    Map<String, Object> map = null;

```

```

        map = jt.queryForMap(GET_EMPLOYEE_DETAILS_BY_ID, eno);
        return map;
    }

    @Override
    public List<Map<String, Object>> getEmployeesDetailsByDesg(String
desg1, String desg2) {
        return jt.queryForList(GET_EMPLOYEES_DETAILS_BY_JOB,
desg1, desg2);
    }

    @Override
    public int insert(String name, String desg, float sal) {
        return jt.update(INSERT_EMPLOYEE, name, desg, sal);
    }

    @Override
    public int addBonusToEmployeeByDesg(String desg, float bonus) {
        return jt.update(ADD_BONUS_BY_JOB, bonus, desg);
    }
}

```

EmployeeMgmtService.java

```

public String fetchEmployeeNameByNo(int eno);
public Map<String, Object> fetchEmployeeDetailsByNo(int eno);
public List<Map<String, Object>>
fetchEmployeesDetailsByDesg(String desg1, String desg2);
public String resgisteEmployee(String name, String desg, float sal);
public String putBonusToEmployeeByDesg(String desg, float bonus);

```

EmployeeMgmtServiceImpl.java

```

@Override
public String fetchEmployeeNameByNo(int eno) {
    return dao.getEmployeeNameByNo(en0);
}

@Override
public Map<String, Object> fetchEmployeeDetailsByNo(int eno) {
    return dao.getEmployeeDetailsByNo(en0);
}

```

```

@Override
public List<Map<String, Object>>
fetchEmployeesDetailsByDesg(String desg1, String desg2) {
    return dao.getEmployeesDetailsByDesg(desg1, desg2);
}

@Override
public String registeEmployee(String name, String desg, float sal) {
    int count = 0;
    //use dao
    count = dao.insert(name, desg, sal);
    return count==0?"Employee is not registered":"Employee is
registered";
}

@Override
public String putBonusToEmployeeByDesg(String desg, float bonus) {
    int count = 0;
    //use dao
    count = dao.addBonusToEmployeeByDesg(desg, bonus);
    return count==0?desg+" Employee record found for add
bouns":count+" No. of records are added with bouns";
}

```

JdbcTemplateTest.java

```

System.out.println("-----");
try {
    System.out.println("7499 Employee Name is :
"+service.fetchEmployeeNameByNo(7499));
} catch (DataAccessException dae) {
    dae.printStackTrace();
}
System.out.println("-----");
try {
    System.out.println("7499 Employee Name is :
"+service.fetchEmployeeDetailsByNo(7499));
} catch (DataAccessException dae) {
    dae.printStackTrace();
}
System.out.println("-----");

```

```

        try {
            //System.out.println("CLERK, MANAGER Employee
            Details : "+service.fetchEmployeesDetailsByDesg("CLERK", "MANAGER"));
            service.fetchEmployeesDetailsByDesg("CLERK",
            "MANAGER").forEach(System.out::println);

        } catch (DataAccessException dae) {
            dae.printStackTrace();
        }
        System.out.println("-----");
        try {
            System.out.println(service.resgisteEmployee("SYAM",
            "MANAGER", 45000));
        } catch (DataAccessException dae) {
            dae.printStackTrace();
        }
        System.out.println("-----");
        try {
            System.out.println(service.putBonusToEmployeeByDesg("MANAGER"
            , 1000));
        } catch (DataAccessException dae) {
            dae.printStackTrace();
        }
    }

```

Converting spring JDBC App into annotation driven configuration-based App: Thumb rule:

- Configure Pre-defined classes as Spring beans using <bean> tags.
- Configure User-defined classes as Spring beans using stereo type annotations and link them with configuration file (xml file) using <context: component-scan> tag.

Directory Structure of DAOAnnoProj01-JdbcTemplate-DirectMethods:

- Copy paste the DAOProj01-JdbcTemplate-DirectMethods application.
- Need Not to add anything same DAOProj01-JdbcTemplate-DirectMethods.
- Add the following code in their respective files.

EmployeeDAOImpl.java

```
Repository("empDAO")  
public class EmployeeDAOImpl implements IEmployeeDAO {  
    @Autowired  
    private JdbcTemplate jt;
```

EmployeeMgmtServiceImpl.java

```
@Service("empService")  
public class EmployeeMgmtService implements IEmployeeMgmtService {  
    @Autowired  
    private IEmployeeDAO dao;
```

service-beans.xml

```
<context:component-scan base-package="com.nt.service"/>
```







persistence-beans.xml

```
<!-- HikariCP DS configuration -->  
<bean id="hkDs" class="com.zaxxer.hikari.HikariDataSource">  
    <property name="driverClassName"  
value="oracle.jdbc.driver.OracleDriver"/>  
    <property name="jdbcUrl"  
value="jdbc:oracle:thin:@localhost:1521:xe"/>  
    <property name="username" value="system"/>  
    <property name="password" value="manager"/>  
    <property name="minimumIdle" value="10"/>  
    <property name="maximumPoolSize" value="30"/>  
</bean>  
  
<!-- JdbcTemplate configuration -->  
<bean id="template"  
class="org.springframework.jdbc.core.JdbcTemplate">  
    <constructor-arg ref="hkDs"/>  
</bean>  
  
<context:component-scan base-package="com.nt.dao"/>
```

Converting Spring JDBC App into 100%Code Driven configuration-based App: Thumb rule:

- Configure User-defined classes as Spring beans using stereo type annotations and link them with @Configuration class using @ComponentScan Annotation.
- Configure Pre-defined classes as Spring beans using @Bean methods in @Configuration classes
- Use AnnotationConfigApplicationContext class to create IoC container.

Directory Structure of DAO100pProj01-JdbcTemplate-DirectMethods:

- Copy paste the DAOAnnoProj01-JdbcTemplate-DirectMethods application.
- Add the following package and class and files
 - ▼  com.nt.commons
 -  jdbc.properties
 - ▼  com.nt.config
 - >  AppConfig.java
 - >  PersistenceConfig.java
 - >  ServiceConfig.java
- Add the following code in their respective files and change the container in Test class.

jdbc.properties

```
#jdbc properties
jdbc.driver=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.user=system
jdbc.pwd=manager
jdbc.hcp.minisize=10
jdbc.hcp.maxsize=100
```

AppConfig.java

```
package com.nt.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({PersistenceConfig.class, ServiceConfig.class})
public class AppConfig {
}
```

ServiceConfig.java

```
package com.nt.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.nt.service")
public class ServiceConfig {

}
```

PersistenceConfig.java

```
package com.nt.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.core.JdbcTemplate;

import com.zaxxer.hikari.HikariDataSource;

@Configuration
@ComponentScan(basePackages = "com.nt.dao")
@PropertySource("com/nt/commons/jdbc.properties")
public class PersistenceConfig {

    @Autowired
    private Environment env;

    @Bean(name="hkDs")
    public HikariDataSource createDataSource() {
        HikariDataSource hkDs = null;
        hkDs = new HikariDataSource();
        hkDs.setDriverClassName(env.getRequiredProperty("jdbc.driver"));
        hkDs.setJdbcUrl(env.getRequiredProperty("jdbc.url"));
        hkDs.setUsername(env.getRequiredProperty("jdbc.user"));
        hkDs.setPassword(env.getRequiredProperty("jdbc.pwd"));
        hkDs.setMinimumIdle(Integer.parseInt(env.getRequiredProperty("jdbc.hcp.minisize")));
    }
}
```

```

        hkDs.setMaximumPoolSize(Integer.parseInt(env.getRequiredProperty
("jdbc.hcp.maxsize")));
        return hkDs;
    }

    @Bean(name="jt")
    public JdbcTemplate createJT() {
        return new JdbcTemplate(createDataSource());
    }
}

```























Converting Spring JDBC App into Spring Boot App:

Thumb rule:

- User-defined classes as Spring beans using stereo type annotations.
- Make sure that all packages are placed under starter/main class package as sub packages.
- Configure pre-defined classes as Spring beans using @Bean methods in @Configuration classes only if they are not coming through AutoConfiguration.
- Get IoC container using SpringApplication.run(-) method.
- if add spring-boot-starter-jdbc to spring boot project, we following classes as spring beans through AutoConfiguration.
 - a. HikariDataSource
 - b. JdbcTemplate
 - c. NamedParameterJdbcTemplate
 - d. DataSourceTransactionManager
 - and etc.

Directory Structure of DAOBootProj01-JdbcTemplate-DirectMethods:

- Develop the below directory structure using Spring Starter Project option and package and classes also.
- Many jars' dependencies will come automatically in pom.xml because while developing Spring Starter Project we choose some jars. They are
 - JDBC API
 - Oracle Driver
- Then use the following code with in their respective file.
- And rest of code copy from Previous project because they are same.

- ▼  DAOBootProj01-JdbcTemplate-DirectMethods [boot]
 - >  Spring Elements
 - ▼  src/main/java
 - ▼  com.nt
 - >  DaoBootProj01JdbcTemplateDirectMethodsApplication.java
 - ▼  com.nt.dao
 - >  EmployeeDAOImpl.java
 - >  IEmployeeDAO.java
 - ▼  com.nt.service
 - >  EmployeeMgmtService.java
 - >  IEmployeeMgmtService.java
 - ▼  src/main/resources
 -  application.properties
 - >  src/test/java
 - >  JRE System Library [JavaSE-11]
 - >  Maven Dependencies
 - >  src
 - >  target
 -  HELP.md
 -  mvnw
 -  mvnw.cmd
 -  pom.xml

application.properties

```
#JDBC properties
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
```

DaoBootProj01JdbcTemplateDirectMethodApplication.java

```
package com.nt;

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

import com.nt.service.IEmployeeMgmtService;

@SpringBootApplication
public class DaoBootProj01JdbcTemplateDirectMethodsApplication {
```

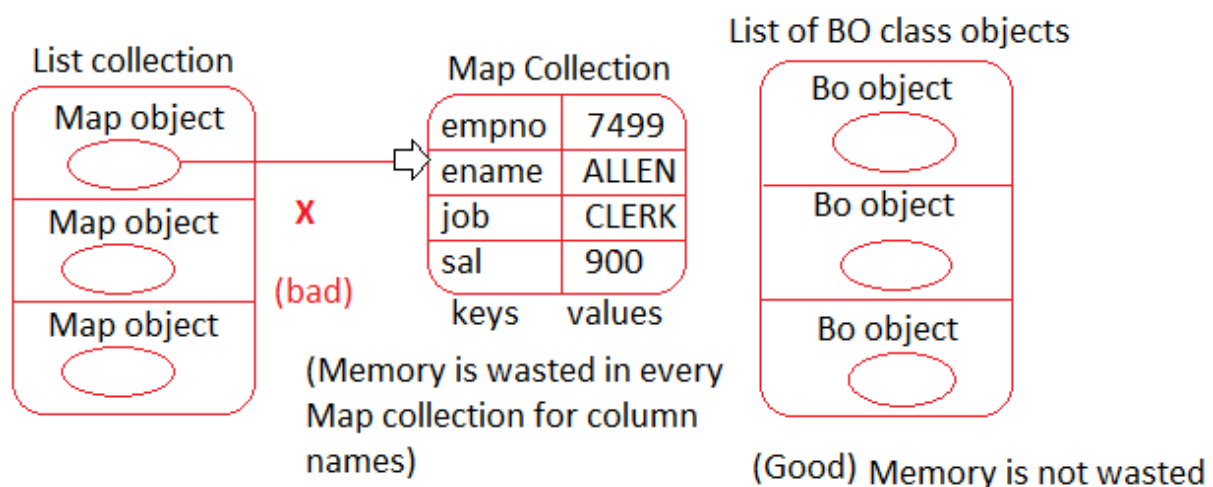
```

public static void main(String[] args) {
    ApplicationContext ctx = null;
    IEmployeeMgmtService service = null;
    //get IoC container
    ctx =
SpringApplication.run(DaoBootProj01JdbcTemplateDirectMethodsApplicati
on.class, args);
    //get service class object
    service = ctx.getBean("empService",
IEmployeeMgmtService.class);
    //invoke methods
    try {
        System.out.println("Employees
count:"+service.fetchEmpsCount());
    } catch (DataAccessException dae) {
        dae.printStackTrace();
    }
}
}

```

Limitations with JdbcTemplate Direct methods:

- a. query(), queryForXxx(-,-) are giving "SELECT" SQL query results in different formats but still they are Industry standard because of memory wastage.
 - queryForMap(-) gives single record as Map object having column names as keys and column values as values here column names wasting the memory. The industry standard is getting record into BO class object.



- queryForList() gives multiple records as Map objects stored into the List Collection. But column names in all Map objects are

wasting the memory. The Industry standard is List Collection with BO class objects.

- b. No ability to use our choice JDBC statement objects to send and execute SQL queries in DB s/w.
- c. Customization of Results (SQL Query results) is more required.

Note: To overcome the above problems use JdbcTemplate methods with Callback Interfaces.

JdbcTemplate Direct methods without callback Interfaces:

- No Boilerplate codes.
- Provides abstraction on plain JDBC.
- Fixed Custom Results like record as Map object, records as List of Map objects which are not industry standard.
- E.g. Staying in hostel with hostel food.

JdbcTemplate direct methods with Callback Interfaces:

- No Boilerplate code problem.
- Provides abstraction on plain JDBC code.
- Exposes required JDBC objects as the parameters of callback methods by creating them internally to customize the results as needed i.e., we can get industry standard results like BO, ListBO and etc.
- E.g. Staying in hostel taking hostel food and kitchen facility to prepare our own food.

Plain JDBC code:

- Boilerplate code problem.
- No Abstraction.
- Pain to programmer to do everything
- Results Customization is completely in our choice. So, we can get results as per industry standard like BO object, ListBO objects and etc.
- E.g. staying in Flat with self-made food.

Callback Interface

Callback method:

- ✚ The method that executes automatically is called Callback method i.e. this method will be called by underlying environment like Spring JDBC or Container or F/w or server automatically.
- ✚ Servlet life cycle methods are called Container callback methods because

we do not call them they will be called ServletContainer automatically for different life cycle events.

Callback Interface:

- + The interface that contains the declaration of callback methods is called Callback Interface.
- + Spring JDBC is providing multiple callback Interfaces, they are
 - a. RowMapper: Gives RS object to customize single record/ row (like BO)
 - b. ResultSetExtractor: Gives RS object to customize multiple records (like ListBO) -> stateless
 - c. RowCallbackHandler: Gives RS object to customize multiple records (like ListBO)-> stateful
 - d. PreparedStatementCreator: Gives JDBC connection object to create PreparedStatement object
 - e. PreparedStatementSetter: Gives JDBC PreparedStatement object to set values to query params and to execute Query
 - f. StatementCallback
 - g. PreparedStatementCallback
 - h. CallableStatementCallback
 - i. PreparedStatementBatchSetter and etc.

RowMapper<T>

- callback method is
`public <T> mapRow (ResultSet rs, int index)`
- Very useful to convert RS object single record to BO class object.
- queryForObject (-, -, -) is having overloaded forms having RowMapper as the parameter type.
 - a. @Nullable **for query with params (?)**
`public <T> T queryForObject (String sql, RowMapper<T> rowMapper, @Nullable Object... args) throws DataAccessException`
 - b. @Nullable **for query without params (?)**
`public <T> T queryForObject (String sql, RowMapper<T> rowMapper) throws DataAccessException`
 - c. @Nullable **for query with params (?) same as (a)**
`public <T> T queryForObject (String sql, Object [] args, int [] argTypes, RowMapper<T> rowMapper) throws DataAccessException`

Note:

- The above methods must be called RowMapper (I) implementation class object as the argument value because if java method parameter type is an interface, we should call method having implementation class obj of that interface as an argument value.
- public Student process (String data) -> Method returns Student class object always (Bad).
- public Person process (String data) Method returns Person or its sub class object (Good).
- To make process(data) method returning any object randomly then we should take Object as return type.
public Object process (String data)
(BAD: while calling we should go for type casting So, there is a chance of getting ClassCastException.)
Student st=(Student)process(...); **code is not type safe**
Employee emp=(Employee) process(..);
- To avoid type casting, go for Generics based method designing.
Public <T>T process (String data, Class<T> clazz);
Student st=process ("...", Student.class); **Code is type safe**
Employee emp=process ("...", Employee.class);

Example on queryForObject (-, -, -) having RowMapper to get record as StudentBO class object:

```
StudentBO bo = jt.queryForObject ("SELECT * FROM STUDENT WHERE  
SNO=?", new StudentMapper (), 101);
```

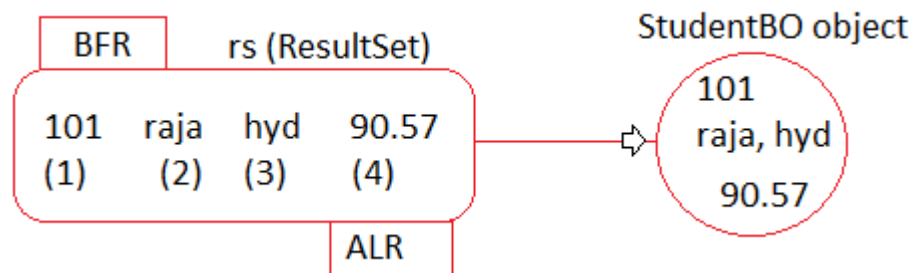
//nested inner class

```
private static class StudentMapper implements RowMapper<StudentBO> {  
    public StudentBO mapRow (ResultSet rs, int index) throws  
        SQLException {  
        //copy ResultSet object record to StudentBO class object  
        StudentBO bo=new StudentBO ();  
        bo.setSno(rs.getInt(1));  
        bo.setSname(rs.getString(2));  
        bo.setSadd(rs.getString(3));  
        bo.setAvg(rs.getFloat(4));  
        return bo;  
    }  
} //inner class
```

```

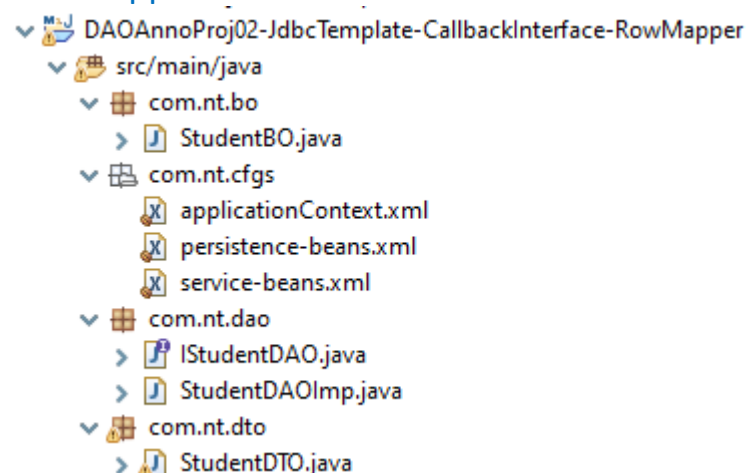
public class StudentBO {
    private int sno;
    private String sname;
    private String sadd;
    private float avg;
    //setters && getters
    .....
}

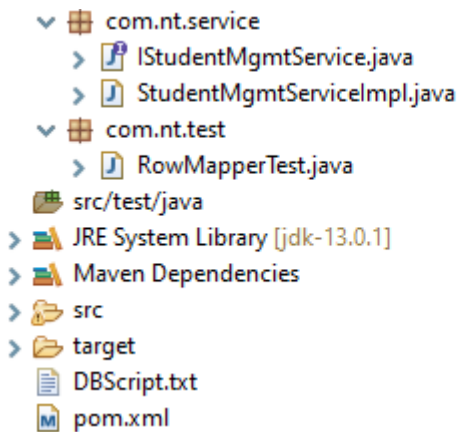
```



Code Flow: queryForObject (-) method gets the injected DS from jt (JdbcTemplate object) -> Using that DS gets one JDBC connection object from JDBC connection pool -> Using that con object creates PreparedStatement object having given SQL query as the pre-compiled SQL query -> Set given var args as query param values -> Executes the query and gets RS object with one record -> calls rs.next() method, gets retrieved record index from DB table -> Takes second argument (StudentMapper object -> RowMapper object) -> calls mapRow(-,-) on that object having RS, record index as the argument values -> mapRow(-,-) copy RS object record to StudentBO class object and returns that object to queryForObject (-, -, -) method and this method returns to its caller (generally the DAO class method).

Directory Structure of DAOAnnoProj02-JdbcTemplate-CallbackInterface-RowMapper:





- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file.
- Change to Java 1.8 or its higher version.
- Rest of code copy from DAOAnnoProj01-JdbcTemplate-DirectMethods

DBScript.txt

```
CREATE TABLE "SYSTEM"."STUDENT"  
  ("SNO" NUMBER(*,0) NOT NULL ENABLE,  
   "SNAME" VARCHAR2(20 BYTE),  
   "ADDRESS" VARCHAR2(20 BYTE),  
   "AVG" FLOAT(126),  
   CONSTRAINT "STUDENT_PK" PRIMARY KEY ("SNO"));
```

IStudentDAO.java

```
package com.nt.dao;  
  
import com.nt.bo.StudentBO;  
  
public interface IStudentDAO {  
    public StudentBO getStudentByNo(int sno);  
}
```

StudentDAOImpl.java

```
package com.nt.dao;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;
```

```

import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import com.nt.bo.StudentBO;

@Repository("studDAO")
public class StudentDAOImpl implements IStudentDAO {

    private static final String GET_STUDENT_BY_ID = "SELECT SNO,
SNAME, ADDRESS, AVG FROM STUDENT WHERE SNO=?";

    @Autowired
    private JdbcTemplate jt;

    @Override
    public StudentBO getStudentByNo(int sno) {
        return jt.queryForObject(GET_STUDENT_BY_ID, new
StudentMapper(), sno);
    }
    //Nested inner class
    private static class StudentMapper implements
RowMapper<StudentBO> {
        @Override
        public StudentBO mapRow(ResultSet rs, int rowNum) throws
SQLException {
            StudentBO bo = null;
            //convert RS record into student BO class object
            bo = new StudentBO();
            bo.setSno(rs.getInt(1));
            bo.setName(rs.getString(2));
            bo.setAddress(rs.getString(3));
            bo.setAvg(rs.getFloat(4));
            return bo;
        }
    }
}

```

IStudentMgmtService.java

```

package com.nt.service;
import com.nt.dto.StudentDTO;
public interface IStudentMgmtService {
    public StudentDTO fetchStudentById(int sno);
}

```


StudentMgmtServiceImpl.java

```
package com.nt.service;

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

import com.nt.bo.StudentBO;
import com.nt.dao.IStudentDAO;
import com.nt.dto.StudentDTO;

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

    @Autowired
    private IStudentDAO dao;

    @Override
    public StudentDTO fetchStudentById(int sno) {
        StudentBO bo = null;
        StudentDTO dto = null;
        //use DAO
        bo = dao.getStudentByNo(sno);
        //copy BO to DTO
        dto = new StudentDTO();
        BeanUtils.copyProperties(bo, dto);
        return dto;
    }
}
```

StudentBO.java

```
package com.nt.bo;

import lombok.Data;

@Data
public class StudentBO {
    private int sno;
    private String sname;
    private String address;
    private float avg;
}
```

StudentDTO.java

```
package com.nt.dto;

import java.io.Serializable;

import lombok.Data;

@Data
public class StudentDTO implements Serializable {
    private int sno;
    private String sname;
    private String address;
    private float avg;
}
```

RowMapperTest.java

```
package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;

import com.nt.dto.StudentDTO;
import com.nt.service.IStudentMgmtService;

public class RowMapperTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        IStudentMgmtService service = null;
        StudentDTO dto = null;
        // Create Container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        //get Service class object
        service = ctx.getBean("studService",
IStudentMgmtService.class);
        //invoke method
        try {
```

```

        dto = service.fetchStudentById(101);
        System.out.println(dto);
    } catch (DataAccessException dae) {
        if (dae instanceof EmptyResultDataAccessException)
            System.err.println("Record not found");
        else if (dae instanceof BadSqlGrammarException)
            System.err.println("SQL syntax problem");
        else
            System.err.println("other internal probelm");
        dae.printStackTrace();
    }
    ((AbstractApplicationContext) ctx).close();
}
}

```

Note: instanceof is java operator to check whether given reference variable/object is pointing to certain type class object or not, it returns boolean value (true/false).

4 types inner classes:

- Normal inner class [To use its logics in multiple non static methods outer class]
- Nested inner class/static inner class [To use its logics in multiple static, non-static methods outer class]
- Local inner class [To use its logics in a method definition in multiple method calls]
- Anonymous inner class [To use its logics only in one method call]

Note: The methods of JdbcTemplate class will throw DataAccessException and its sub classes related Exceptions based on the problem that is raised.

Anonymous inner class-based logic while working with queryForObject (-, -, -) having RowMapper:

```

@Override
public StudentBO getStudentByNo(int sno) {
    return jt.queryForObject(GET_STUDENT_BY_ID, new
    RowMapper<StudentBO>() {
        @Override
        public StudentBO mapRow(ResultSet rs, int rowNum)

```

```

throws SQLException {
    StudentBO bo = null;
    //convert RS record into student BO class object
    bo = new StudentBO();
    bo.setSno(rs.getInt(1));
    bo.setSname(rs.getString(2));
    bo.setAddress(rs.getString(3));
    bo.setAvg(rs.getFloat(4));
    return bo;
}
}, sno);
}

```

Note: In second argument total 3 things are happening:

- One anonymous (name less) inner class created implementing RowMapper (I).
- mapRow (-, -) is implemented inside that Anonymous inner class.
- Object is created for anonymous inner class and passed it as second argument to queryForObject (-, -, -) method.

Lambda Expression based Anonymous inner class logic while working with queryForObject (-, -, -) having RowMapper:

```

@Override
public StudentBO getStudentByNo(int sno) {
    return jt.queryForObject(GET_STUDENT_BY_ID, (rs, rowNum) -
> {
        StudentBO bo = null;
        //convert RS record into student BO class object
        bo = new StudentBO();
        bo.setSno(rs.getInt(1));
        bo.setSname(rs.getString(2));
        bo.setAddress(rs.getString(3));
        bo.setAvg(rs.getFloat(4));
        return bo;
    }, sno);
}

```

Note: BeanPropertyRowMapper<T> is pre-defined implementation class of RowMapper<T> (I) having logic to copy RS object record given Java Bean class

object properties but RS object record DB table column names and Java Bean class property names must match.

```
@Override
public StudentBO getStudentByNo(int sno) {
    return jt.queryForObject(GET_STUDENT_BY_ID, new
    BeanPropertyRowMapper<>(StudentBO.class), sno);
}
```

ResultSetExtractor<T>

- ✚ If SELECT SQL Query execution gives multiple records to process then go for ResultSetExtractor<T> or RowCallbackHandler<T>.
- ✚ The Best use case is getting List of BO class objects from RS after executing Select SQL query that gives multiple records.

<T> T	<u>query</u> (String sql, <u>ResultSetExtractor</u> <T> rse) Execute a query given static SQL, reading the ResultSet with a ResultSetExtractor.
--------------------	--

<T> T	<u>query</u> (String sql, <u>ResultSetExtractor</u> <T> rse, <u>Object</u> ... args) Query given SQL to create a prepared statement from SQL and a list of arguments to bind to the query, reading the ResultSet with a ResultSetExtractor.
--------------------	---

<T> T	<u>query</u> (String sql, <u>Object</u> [] args, <u>ResultSetExtractor</u> <T> rse) Query given SQL to create a prepared statement from SQL and a list of arguments to bind to the query, reading the ResultSet with a ResultSetExtractor.
--------------------	--

Requirement: DAO class should give bunch of records as List<StudentBO> objects from Student DB table based on the given address (sadd) city1, city2, city3 values.

Note: RowMapper<T>, ResultSetExtractor<T>, RowCallbackHandler<T> are functional interfaces because they are having only one method declaration directly or indirectly.

ResultSetExtractor<T> (I) [Callback interface]

|-> public <T> extractData(ResultSet rs) throws SQLException [callback method], According to the above require method we should take <T> as List<StudentBO>.

IStudentDAO.java

```
public List<StudentBO> getStudentsByCities(String city1, String city2,
String city3);
```

StudentDAOImpl.java

```
private static final String GET_STUDENTS_BY_CITIES = "SELECT SNO,
SNAME, ADDRESS, AVG FROM STUDENT WHERE ADDRESS IN(?, ?, ?)";
```

```
@Override
public List<StudentBO> getStudentsByCities(String city1, String city2,
String city3) {
    return jt.query(GET_STUDENTS_BY_CITIES, new
StudentExtractor(), city1, city2, city3);
}
```

```
private static class StudentExtractor implements
ResultSetExtractor<List<StudentBO>>{
```

```
@Override
public List<StudentBO> extractData(ResultSet rs) throws
SQLException, DataAccessException {
    List<StudentBO> listBO = new ArrayList<>();
    //copy Rs object records to listBO
    while (rs.next()) {
        StudentBO bo = new StudentBO();
        bo.setSno(rs.getInt(1));
        bo.setSname(rs.getString(2));
        bo.setAddress(rs.getString(3));
        bo.setAvg(rs.getFloat(4));
        listBO.add(bo);
    }
    return listBO;
}
}
```

IStudentMgmtService.java

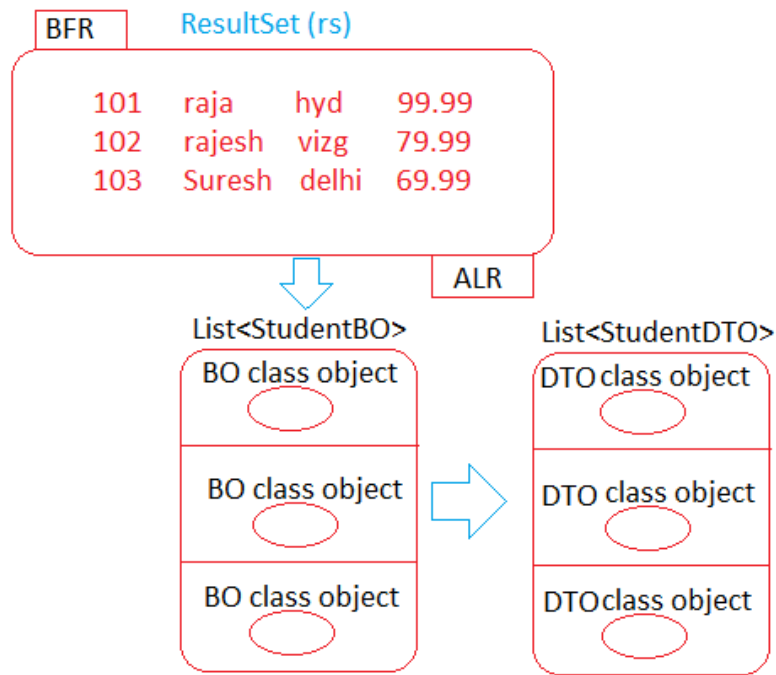
```
public List<StudentDTO> fetchStudentsByCities(String city1,String city2, String city3);
```

StudentMgmtServiceImpl.java

```
@Override
public List<StudentDTO> fetchStudentsByCities(String city1, String city2, String city3) {
    List<StudentBO> listBO = null;
    //use DAO
    listBO = dao.getStudentsByCities(city1, city2, city3);
    //copy listBO to list DTO
    List<StudentDTO> listDTO = new ArrayList<>();
    listBO.forEach(bo->{
        StudentDTO dto = new StudentDTO();
        BeanUtils.copyProperties(bo, dto);
        listDTO.add(dto);
    });
    return listDTO;
}
```

RowMapperTest.java

```
try {
    List<StudentDTO> listDTO =
service.fetchStudentsByCities("hyd", "vizg", "delhi");
    listDTO.forEach(System.out::println);
} catch (DataAccessException dae) {
    if (dae instanceof EmptyResultDataAccessException)
        System.err.println("Record not found");
    else if (dae instanceof BadSqlGrammarException)
        System.err.println("SQL syntax problem");
    else
        System.err.println("other internal probelm");
    dae.printStackTrace();
}
```



Working with anonymous inner class:

StudentDAOImpl.java

```
@Override
public List<StudentBO> getStudentsByCities(String city1, String city2,
String city3) {
    return jt.query(GET_STUDENTS_BY_CITIES, new
ResultSetExtractor<List<StudentBO>>() {
        @Override
        public List<StudentBO> extractData(ResultSet rs) throws
SQLException, DataAccessException {
            List<StudentBO> listBO = new ArrayList<>();
            //copy Rs object records to listBO
            while (rs.next()) {
                StudentBO bo = new StudentBO();
                bo.setSno(rs.getInt(1));
                bo.setSname(rs.getString(2));
                bo.setAddress(rs.getString(3));
                bo.setAvg(rs.getFloat(4));
                listBO.add(bo);
            }
            return listBO;
        }
    }, city1, city2, city3);
}
```


Working with Lambda Expression based anonymous inner class:

StudentDAOImpl.java

```
@Override
public List<StudentBO> getStudentsByCities(String city1, String city2,
String city3) {
    return jt.query(GET_STUDENTS_BY_CITIES, rs->{
        List<StudentBO> listBO = new ArrayList<>();
        //copy Rs object records to listBO
        while (rs.next()) {
            StudentBO bo = new StudentBO();
            bo.setSno(rs.getInt(1));
            bo.setName(rs.getString(2));
            bo.setAddress(rs.getString(3));
            bo.setAvg(rs.getFloat(4));
            listBO.add(bo);
        }
        return listBO;
    }, city1, city2, city3);
}
```

Working with the predefined RowMapperResultSetExtractor (C):

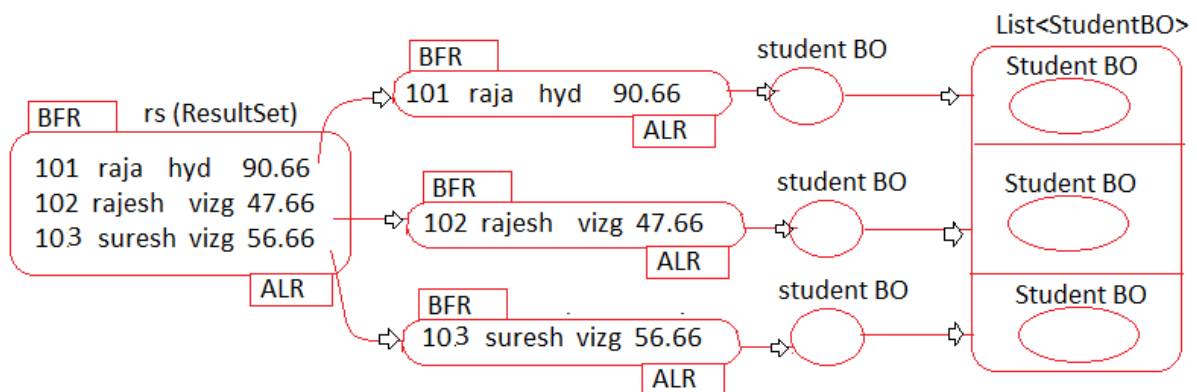
StudentDAOImpl.java

```
@Override
public List<StudentBO> getStudentsByCities(String city1, String city2,
String city3) {
    BeanPropertyRowMapper<StudentBO> bprm = null;
    //create BeanPropertyRowMapper class object that helps to
    copy each record into one BO class object
    bprm = new BeanPropertyRowMapper<>(StudentBO.class);
    return jt.query(GET_STUDENTS_BY_CITIES, new
    RowMapperResultSetExtractor<StudentBO>(bprm), city1, city2, city3);
}
```

RowCallbackHandler

- ✚ Method is: public void processRow (ResultSet rs) throws SQLException
- ✚ It is stateful be implementation class obj remembers: the state across the multiple executions of processRow (-) method.

- ✚ In contrast to a ResultSetExtractor, a RowCallbackHandler object is typically stateful: It keeps the result state within the object, to be available for later inspection.



RowCallbackHandler implementation using Nested inner class:

[StudentDAOImpl.java](#)

```

@Override
public List<StudentBO> getStudentsByCities1(String city1, String city2,
String city3) {
    List<StudentBO> listBO = new ArrayList();
    jt.query(GET_STUDENTS_BY_CITIES, new
StudentCallBackHandler(listBO), city1, city2, city3);
    return listBO;
}

private static class StudentCallBackHandler implements
RowCallbackHandler {
    private List<StudentBO> listBO;
    public StudentCallBackHandler(List<StudentBO> listBO) {
        this.listBO = listBO;
    }
    @Override
    public void processRow(ResultSet rs) throws SQLException {
        //convert RS record into BO
        StudentBO bo = new StudentBO();
        bo.setSno(rs.getInt(1));
        bo.setName(rs.getString(2));
        bo.setAddress(rs.getString(3));
        bo.setAvg(rs.getFloat(4));
        listBO.add(bo);
    }
}
  
```

Flow: jt.query(-,-,-) gets injected DS --> gets connection object from DS --> creates PS having given Query as pre-compiled query --> set city1/2/3 as the query par values --> executes Query and gets RS(main In a loop gets each record from main RS and creates separate RS and calls processRow(RS) method for multiple times. In the Process ListBO is filled up BO objects given by processRow(-,-) method.

RowCallbackHandler implementation using Anonymous inner class:
[StudentDAOImpl.java](#)

```
@Override
public List<StudentBO> getStudentsByCities1(String city1, String city2,
String city3) {
    List<StudentBO> listBO = new ArrayList();
    jt.query(GET_STUDENTS_BY_CITIES, new
RowCallbackHandler() {

        @Override
        public void processRow(ResultSet rs) throws
SQLException {

            //convert RS record into BO
            StudentBO bo = new StudentBO();
            bo.setSno(rs.getInt(1));
            bo.setSname(rs.getString(2));
            bo.setAddress(rs.getString(3));
            bo.setAvg(rs.getFloat(4));
            listBO.add(bo);

        }
    }, city1, city2, city3);
    return listBO;
}
```

RowCallbackHandler implementation using Lambda expression based
Anonymous inner class:
[StudentDAOImpl.java](#)

```
@Override
public List<StudentBO> getStudentsByCities1(String city1, String city2,
String city3) {
    List<StudentBO> listBO = new ArrayList();
    jt.query(GET_STUDENTS_BY_CITIES, rs -> {
```

```

        //convert RS record into BO
        StudentBO bo = new StudentBO();
        bo.setSno(rs.getInt(1));
        bo.setName(rs.getString(2));
        bo.setAddress(rs.getString(3));
        bo.setAvg(rs.getFloat(4));
        listBO.add(bo);
    }, city1, city2, city3);
    return listBO;
}

```

Q. What is difference b/w ResultSetExtractor and RowCallbackHandler callback Interfaces?

Ans.

ResultSetExtractor (I)	RowCallbackHandler (I)
(a) it stateless in nature because there is no need of remembering state across the multiple executions of implementation class object.	(a) It is stateful in nature because it remembers the given state like ListBO across the multiple executions of the implementation class object.
(b) extractData (-) is the callback method and it executes only for 1 time.	(b) processRow (-) is callback method and it executes for multiple times.
(c) Involves only one ResultSet object in the entire process.	(c) Involves multiple RS Objects (n+1) in the entire process, n --> records count given by "SELECT SQL Query".
(d) Good in performance	(d) Bad in performance.
(e) Support for Generics.	(e) No support for Generics.
(f) We have multiple useful readymade implementation classes.	(f) We do not have here.

NamedParameterJdbcTemplate

- ✚ It is given to support named parameters in the SQL query.
- ✚ The Limitation with positional params (?) is providing index and setting values to those parameters according to the index is bit complex especially if the query having multiple positional parameters.
- ✚ To overcome the above problem use named parameters (:<name>) which gives name to each parameter and we can set values to parameters by specifying their name.

Query with positional params:

```
SELECT EMPNO, ENAME, JOB, SAL FROM EMP WHERE EMPNO>=? AND  
EMPNO<=? 2 1
```

Query with named params:

```
SELECT EMPNO, ENAME, JOB, SAL FROM EMP WHERE EMPNO>=: min  
AND EMPNO<=: max (named parameter)
```

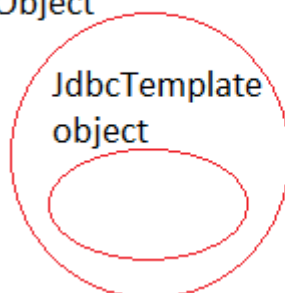
Note:

- ✓ JdbcTemplate does not support Named Parameters. It supports only Positional parameters.
- ✓ NamedParameterJdbcTemplate supports named Parameters but does not support positional parameters.

✚ This class delegates to a wrapped JdbcTemplate once the substitution from named parameters to JDBC style '?' placeholders is done at execution time. It also allows for expanding a List of values to the appropriate number of placeholders.

NamedParameterJdbcTemplate

Object



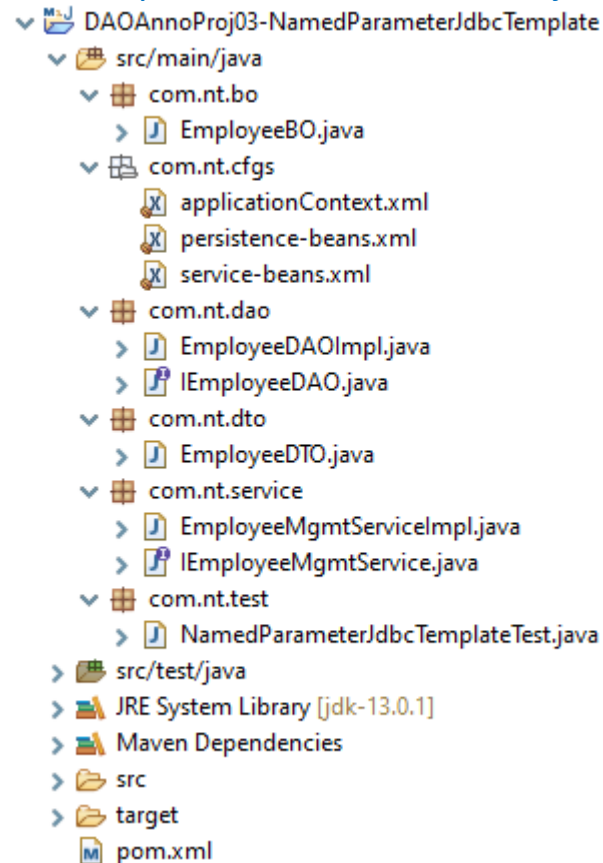
NamedParameterJdbcTemplate object has JdbcTemplate object i.e. composition (Has-A Relation)

We can set value Named Parameters in 2 ways while working with NamedParameterJdbcTemplate:

- Using Map<String, Object> object
 - here the named parameter names are keys and param values are values.
- Using SqlParameterSource (I) Implementations
 - MapSqlParameterSource (c)
 - Use its addValue (-, -) method having param name, param value as the arguments.
 - BeanPropertySqlParameterSource (c)
 - Allows to set Java Bean object values as the named parameter values but the names of named parameters and the names java bean class properties must match.

- ✚ To create NamedParameterJdbcTemplate we need DS object as the dependent object.
- ✚ NamedParameterJdbcTemplate also gives support to work with callback interfaces.
- ✚ Named Parameters are case-sensitive.

Directory Structure of DAOAnnoProj03-NamedParameterJdbcTemplate:



- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file.
- Change to Java 1.8 or its higher version.
- Rest of code copy from DAOAnnoProj01-JdbcTemplate-DirectMethods.

persistence-beans.xml

```

<!-- JdbcTemplate configuration -->
<bean id="template"
class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
    <constructor-arg ref="hkDs"/>
</bean>
  
```

EmployeeBO.java

```
package com.nt.bo;

import lombok.Data;

@Data
public class EmployeeBO {
    private Integer empno;
    private String ename;
    private String job;
    private Float sal;
}
```

EmployeeDTO.java

```
package com.nt.dto;

import lombok.Data;

@Data
public class EmployeeDTO {
    private Integer empno;
    private String ename;
    private String job;
    private Float sal;
}
```

IEmployeeDAO.java

```
package com.nt.dao;

import java.util.List;

import com.nt.bo.EmployeeBO;

public interface IEmployeeDAO {
    public String getEmployeeNameByNo(int no);
    public List<EmployeeBO> getEmployeeDetailsByDesgs(String desg1,
String desg2, String desg3);
    public int insertEmployee(EmployeeBO bo);
}
```

EmployeeDAOImpl.java

```
package com.nt.dao;

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import com.nt.bo.EmployeeBO;

@Repository("empDAO")
public class EmployeeDAOImpl implements IEmployeeDAO {

    private static final String GET_EMPLOYEE_NAME_BY_ID = "SELECT
ENAME FROM EMP WHERE EMPNO=:no";
    private static final String GET_EMPLOYEE_NAME_BY_JOBS = "SELECT
EMPNO, ENAME, JOB, SAL FROM EMP WHERE JOB IN (:desg1, :desg2,
:desg3)";
    private static final String INSERT_EMPLOYEE = "INSERT INTO EMP
(EMPNO, ENAME, JOB, SAL) VALUES (:empno, :ename, :job, :sal)";

    @Autowired
    private NamedParameterJdbcTemplate npjt;

    @Override
    public String getEmployeeNameByNo(int no) {
        Map<String, Object> paramMap = Map.of("no", no);
        String name =
npjt.queryForObject(GET_EMPLOYEE_NAME_BY_ID, paramMap,
String.class);
        return name;
    }

    @Override
```



```

    public List<EmployeeBO> getEmployeeDetailsByDesgs(String desg1,
String desg2, String desg3) {
        //Prepare MapSqlParameterSource Object having names,
values of the named parameters
        MapSqlParameterSource msps = new
MapSqlParameterSource();
        msps.addValue("desg1", desg1);
        msps.addValue("desg2", desg2);
        msps.addValue("desg3", desg3);
        List<EmployeeBO> listBO =
npjt.query(GET_EMPLOYEE_NAME_BY_JOBS, msps, rs->{
            List<EmployeeBO> listBO1 = new ArrayList<>();
            while(rs.next()) {
                EmployeeBO bo = new EmployeeBO();
                bo.setEmpno(rs.getInt(1));
                bo.setEname(rs.getString(2));
                bo.setJob(rs.getString(3));
                bo.setSal(rs.getFloat(4));
                listBO1.add(bo);
            }
            return listBO1;
        });
        return listBO;
    }

    @Override
    public int insertEmployee(EmployeeBO bo) {
        // create BeanPropertySqlParameterSource object
        BeanPropertySqlParameterSource bpsps = new
BeanPropertySqlParameterSource(bo);
        //execute query
        return npjt.update(INSERT_EMPLOYEE, bpsps);
    }
}

```

!EmployeeMgmtService.java

```

package com.nt.service;

import java.util.List;

import com.nt.dto.EmployeeDTO;

```

```

public interface IEmployeeMgmtService {
    public String fetchEmployeeNameByNo(int no);
    public List<EmployeeDTO> fetchEmployeeDetailsByDesgs(String
desg1, String desg2, String desg3);
    public String registerEmployee(EmployeeDTO dto);
}

```

EmployeeMgmtServiceImpl.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.stereotype.Service;

import com.nt.bo.EmployeeBO;
import com.nt.dao.IEmployeeDAO;
import com.nt.dto.EmployeeDTO;

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

    @Autowired
    private IEmployeeDAO dao;

    @Override
    public String fetchEmployeeNameByNo(int no) {
        return dao.getEmployeeNameByNo(no);
    }

    @Override
    public List<EmployeeDTO> fetchEmployeeDetailsByDesgs(String
desg1, String desg2, String desg3) {
        //use DAO
        List<EmployeeBO> listBO =
dao.getEmployeeDetailsByDesgs(desg1, desg2, desg3);
        //copy listBO to listDTO
        List<EmployeeDTO> listDTO = new ArrayList<>();
        listBO.forEach(bo->{
            EmployeeDTO dto = new EmployeeDTO();

```

```

        BeanUtils.copyProperties(bo, dto);
        listDTO.add(dto);
    });
    return listDTO;
}

@Override
public String registerEmployee(EmployeeDTO dto) {
    //Convert DTO to BO
    EmployeeBO bo = new EmployeeBO();
    BeanUtils.copyProperties(dto, bo);
    //use DAO
    int count = dao.insertEmployee(bo);
    return count==1?"Employee Registered":"Employee not
Registered";
}
}

```

NamedParameterJdbcTemplateTest.java

```

package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;

import com.nt.dto.EmployeeDTO;
import com.nt.service.IEmployeeMgmtService;

public class NamedParameterJdbcTemplateTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        IEmployeeMgmtService service = null;
        // Create IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        // get Service class object
        service = ctx.getBean("empService",

```

```

IEmployeeMgmtService.class);
    // invoke method
    try {
        System.out.println("Employee Name :
"+service.fetchEmployeeNameByNo(7900));
        System.out.println("-----");
        service.fetchEmployeeDetailsByDesgs("MANAGER",
"CLERK", "SALESMAN").forEach(System.out::println);
        System.out.println("-----");
        EmployeeDTO dto = new EmployeeDTO();
        dto.setEmpno(3534);
        dto.setEname("RAJESH");
        dto.setJob("SALESMAN");
        dto.setSal(45456.6f);
        System.out.println(service.registerEmployee(dto));
    } catch (DataAccessException dae) {
        if (dae instanceof EmptyResultDataAccessException)
            System.err.println("Record not found");
        else if (dae instanceof BadSqlGrammarException)
            System.err.println("SQL syntax problem");
        else
            System.err.println("other internal probelm");
        dae.printStackTrace();
    }
    ((AbstractApplicationContext) ctx).close();
}
}

```

SimpleJdbcTemplate:

- ✚ Introduced in spring 2.x as alternate to JdbcTemplate supporting new features of that time like generics, var args and etc.
- ✚ Continued and deprecated in spring 3.x because they upgraded JdbcTemplate itself supporting features like generics, var args and etc.
- ✚ In Spring 4.1, the SimpleJdbcTemplate is removed.

SimpleJdbcInsert or Call

SimpleJdbcInsert

- ✚ A SimpleJdbcInsert is a multi-threaded reusable object providing easy insert capabilities for a table. It provides meta-data processing to simplify the code needed to construct a basic insert query. All you need to provide is the name of the table and a Map containing the column names and the column values.
- ✚ JdbcTemplate, NamedParameterJdbcTemplate, SimpleJdbcTemplate are thread safe i.e. they are single threaded objects. So, they allow only one thread at time to perform persistence operation i.e. they are not suitable multi-threaded time critical web application environment like online auction/ bidding and online counselling, online shopping and etc.
- ✚ In the above situations, we can use "SimpleJdbcInsert" for insert persistence operations because it is multithreaded. i.e. multiple threads can be performed insert operation simultaneously.
- ✚ While working with "SimpleJdbcInsert" we do not write "INSERT SQL Query" separately we just provide DS, DB table name, Map of column names, values then insert SQL Query will be generated dynamically

SimpleJdbcInsert

```
|--> DS (as dependent obj)
|--> setTable (-)
|--> int execute (Map<String, Object> map)
        takes column names and column values.
        (or)
|--> int execute (SqlParameterSource source)
        |--> MapSqlParameterSource (c)
            using addValue (-, -) we need to pass column
            names and column values
        |--> BeanPropertySqlParameterSource(c)
            Here we can pass JavaBean object as input for
            column names and column values but DB table
            column names and Java Bean property names
            must match.
```

Note: The actual insert is being handled using Spring's JdbcTemplate.

Q. SimpleJdbcInsert Internally uses JdbcTemplate for completing generated insert SQL query execution then how can say it is multi-threaded as we know JdbcTemplate is single Threaded?

Ans. If we call execute (-) method on SimpleJdbcInsert for multiple times then multiple JdbcTemplate class objects will be used internally to execute the generated Insert SQL query for multiple times, so the SimpleJdbcInsert becomes multi-threaded.

Q. Why Spring JDBC is not providing "SimpleJdbcUpdate", "SimpleJdbcDelete" and "SimpleJdbcSelect" classes?

Ans. Update, delete and select SQL queries execution takes place along with conditions. Based on given table name, column names and column value these conditions cannot be generated dynamically. So, there are no "SimpleJdbcUpdate", "SimpleJdbcDelete", "SimpleJdbcSelect" classes.

Note: Insert SQL query executes without any condition i.e. it can be generated dynamically based on the given DB table name, column names, column values. So, "SimpleJdbcInsert" is given.

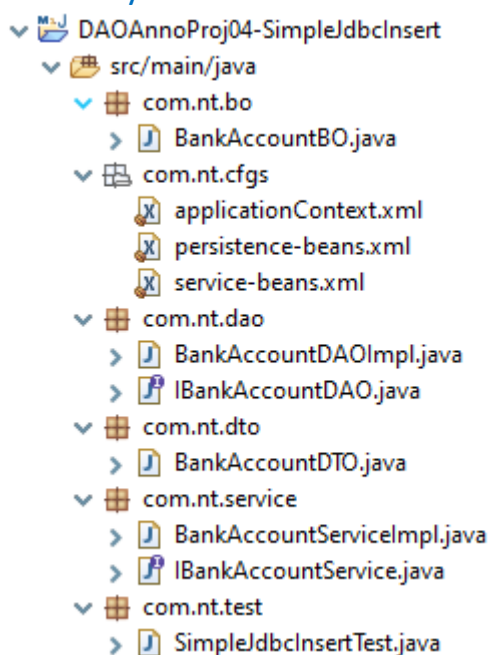
Q. How to execute update, delete, select SQL queries in multi-threaded environment?

Ans. SimpleJdbcCall multi-threaded object having ability to call PL/SQL procedures or functions so keep your update, delete, select SQL queries inside PL/SQL procedure or functions and call them by using SimpleJdbcCall object.

Q. Will SimpleJdbcInsert support positional/ named parameters?

Ans. Since programmer is not preparing query and query is generated dynamically so, there is no possibility placing of any kind of parameters.

Directory Structure of DAOAnnoProj04-SimpleJdbcInsert:



- > src/test/java
- > JRE System Library [jdk-13.0.1]
- > Maven Dependencies
- > src
- > target
- pom.xml

- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file and Change to Java 1.8 or its higher version.
- Copy the xml files and pom.xml from previous project.

persistence-beans.xml

```

    <!-- SimpleJdbcInsert configuration -->
    <bean id="sji"
class="org.springframework.jdbc.core.simple.SimpleJdbcInsert">
        <constructor-arg ref="hkDs"/>
    </bean>

```

BankAccountBO.java

```

package com.nt.bo;
import lombok.Data;
@Data
public class BankAccountBO {
    private Long accno;
    private String holderName;
    private Float balance;
    private String status;
}

```

BankAccountDTO.java

```

package com.nt.dto;
import lombok.Data;
@Data
public class BankAccountDTO {
    private Long accno;
    private String holderName;
    private Float balance;
    private String status;
}

```

IBankAccountDAO.java

```
package com.nt.dao;

import com.nt.bo.BankAccountBO;

public interface IBankAccountDAO {
    public int register(BankAccountBO bo);
}
```

BankAccountDAOImpl.java

```
package com.nt.dao;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
import org.springframework.stereotype.Repository;

import com.nt.bo.BankAccountBO;

@Repository("accDAO")
public class BankAccountDAOImpl implements IBankAccountDAO {

    @Autowired
    private SimpleJdbcInsert sji;

    @Override
    public int register(BankAccountBO bo) {
        //prepare Map object having columns and values
        Map<String, Object> map = Map.of("accno", bo.getAccno(),
"holderName", bo.getHolderName(), "balance", bo.getBalance(), "status",
bo.getStatus());
        //set db table name
        sji.setTableName("BANK_ACCOUNT");
        //execute query by generating the query dynamically
        int count = sji.execute(map);
        return count;
    }
}
```


IBankAccountMgmtService.java

```
package com.nt.service;

import com.nt.dto.BankAccountDTO;

public interface IBankAccountService {
    public String createBankAccount(BankAccountDTO dto);
}
```

BankAccountMgmtServiceImpl.java

```
package com.nt.service;

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

import com.nt.bo.BankAccountBO;
import com.nt.dao.IBankAccountDAO;
import com.nt.dto.BankAccountDTO;

@Service("accService")
public class BankAccountServiceImpl implements IBankAccountService {

    @Autowired
    private IBankAccountDAO dao;

    @Override
    public String createBankAccount(BankAccountDTO dto) {
        BankAccountBO bo = new BankAccountBO();
        //convert DTO to BO
        BeanUtils.copyProperties(dto, bo);
        //use dao
        int count = dao.register(bo);
        return count==1?"dto.getHolderName()+" your account has
created":dto.getHolderName()+" your account has not created yet";
    }
}
```

SimpleJdbcInsertTest.java

```
package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
```

```

import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;

import com.nt.dto.BankAccountDTO;
import com.nt.service.IBankAccountService;

public class SimpleJdbcInsertTest {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        IBankAccountService service = null;
        // Create IoC container
        ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        // get Service class object
        service = ctx.getBean("accService", IBankAccountService.class);
        // invoke method
        try {
            BankAccountDTO dto = new BankAccountDTO();
            dto.setAccno(64897241); dto.setHolderName("Ramesh");
            dto.setBalance(347374f); dto.setStatus("Active");
            System.out.println(service.createBankAccount(dto));
        } catch (DataAccessException dae) {
            if (dae instanceof EmptyResultDataAccessException)
                System.err.println("Record not found");
            else if (dae instanceof BadSqlGrammarException)
                System.err.println("SQL syntax problem");
            else
                System.err.println("other internal probelm");
            dae.printStackTrace();
        }
        ((AbstractApplicationContext) ctx).close();
    }
}

```

Note: If we add "spring-boot-starter-jdbc" to the spring boot project then SimpleJdbcInsert, SimpleJdbcCall will not come through AutoConfiguration we need to create them explicitly by using @Bean methods.

- If DB table column names are matching BO class object property names then we can call execute (-) of SimpleJdbcInsert having BeanPropertySqlParameterSource object as shown below.

BankAccountDAOImpl.java

```
@Override
public int register(BankAccountBO bo) {
    //prepare BeanPropertySqlParameterSource object having BO
object
    //Here column names must match to BO class property name.
    BeanPropertySqlParameterSource bpsps = new
BeanPropertySqlParameterSource(bo);
    //set DB table name
    sj.setTableName("BANK_ACCOUNT");
    //execute query by generating the query dynamically
    int count = sj.execute(bpsps);
    return count;
}
```

SimpleJdbcCall

- ✚ A SimpleJdbcCall is a multi-threaded, reusable object representing a call to a stored procedure or a stored function. It provides meta-data processing to simplify the code needed to access basic stored procedures/ functions. All you need to provide is the name of the procedure/ function and a Map containing the parameters when you execute the call. The names of the supplied parameters will be matched up with in and out parameters declared when the stored procedure was created.

- ✚ Instead of writing same persistence logic/ business logic in multiple modules as SQL queries/ Java code, it is recommended to write only for 1 time in DB s/w as stored procedure / function and use it multiple modules.

e.g.1: Authentication logic as PL/SQL procedure/ function

e.g.2: Attendance calculation logic as PL/SQL procedure function

- ✚ PL/SQL procedure does not return a value but to get multiple results from PL/SQL procedure we need to use multiple OUT params.
- ✚ PL/SQL function returns a value. So, to get multiple results from PL/SQL

function we need to get 1 result as return value and remaining results as OUT params.

- In PL/SQL procedure or function the params will have type (data type), mode.

The modes are:

- IN (default)
- OUT
- INOUT

e.g.: PL/SQL Logic in oracle

y: = x*x; x as in mode param, y as out mode param.

e.g.: PL/SQL logic in oracle

x: =x*x; x as INOUT param

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

PL/SQL procedure for Authentication:

Step 1: Make sure that one DB table is taken having usernames and passwords.

USERINFO (DB table)

	COLUMN_NAME	DATA_TYPE
1	USERNAME	VARCHAR2 (20 BYTE)
2	PASSWORD	VARCHAR2 (20 BYTE)

Step 2: create PL/SQL procedure having authentication logic

```
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;
```

SimpleJdbcCall

|--> DS (required as dependent object)
|--> setProcedureName (-) [To specify PL/SQL procedure name]
|--> setFunction(true/false) [Set to true, if the above name is PL/SQL
function name otherwise set to false (default)]

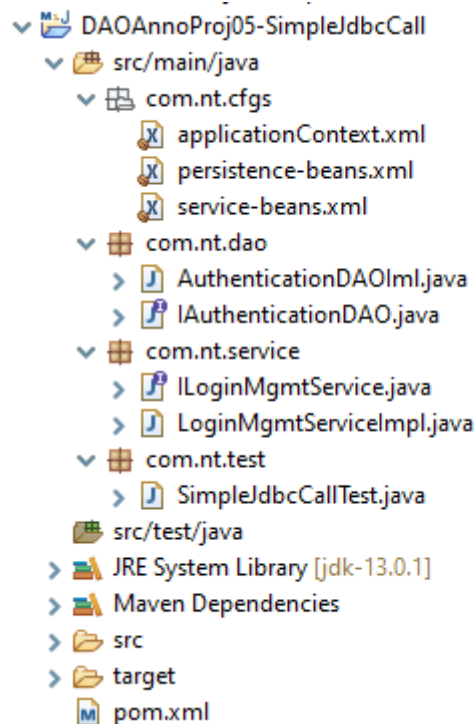
To call PL/SQL Procedure:

|--> Map<String, Object> execute (Map<String, Object>)
1st Map, gives OUT Param names and values as Map object.
2nd Map, to supply IN param names and values as Map object.
|--> Map<String, Object> execute (SqlParameterSource source)

To call PL/SQL Function:

|--> T<T> executeFunction (Class <T> returnType, Map<String,?>
inParams)
|--> T<T> executeFunction (Class <T> returnType, SqlParameterSource
inParams)

Directory Structure of DAOAnnoProj05-SimpleJdbcCall:



- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file and Change to Java 1.9 or its higher version.
- Copy the xml files and pom.xml from previous project.

persistence-beans.xml

```
<!-- SimpleJdbcCall configuration -->
<bean id="sjc"
class="org.springframework.jdbc.core.simple.SimpleJdbcCall">
    <constructor-arg ref="hkDs"/>
</bean>
```

IAuthenticationDAO.java

```
package com.nt.dao;

public interface IAuthenticationDAO {
    public String authentication(String user, String pwd);
}
```

AuthenticationDAOImpl.java

```
package com.nt.dao;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.simple.SimpleJdbcCall;
import org.springframework.stereotype.Repository;

@Repository("authDAO")
public class AuthenticationDAOImpl implements IAuthenticationDAO {

    @Autowired
    private SimpleJdbcCall sjc;

    @Override
    public String authentication(String user, String pwd) {
        //Set procedure name
        sjc.setProcedureName("P_AUTHENTICATION");
        //prepare Map object of IN params
        Map<String, ?> inParams = Map.of("UNAME", user, "PASS",
pwd);

        //call PL/SQL procedure
        Map<String, ?> outParams = sjc.execute(inParams);
        return (String) outParams.get("RESULT");
    }

}
```

ILoginMgmtService.java

```
package com.nt.service;

public interface ILoginMgmtService {
    public String login(String user, String pwd);
}
```

LoginMgmtServiceImpl.java

```
package com.nt.service;

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

import com.nt.dao.IAuthenticationDAO;

@Service("loginService")
public class LoginMgmtServiceImpl implements ILoginMgmtService {

    @Autowired
    private IAuthenticationDAO dao;

    @Override
    public String login(String user, String pwd) {
        //use DAO
        String result=dao.authentication(user, pwd);
        return result;
    }
}
```

SimpleJdbcCallTest.java

```
package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;

import com.nt.service.ILoginMgmtService;

public class SimpleJdbcCallTest {
```

```

    public static void main(String[] args) {
        ApplicationContext ctx = null;
        ILoginMgmtService service = null;
        // Create IoC container
        ctx = new
        ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        // get Service class object
        service = ctx.getBean("loginService", ILoginMgmtService.class);
        // invoke method
        try {
            System.out.println(service.login("nimu", "nimu@123"));
        } catch (DataAccessException dae) {
            if (dae instanceof EmptyResultDataAccessException)
                System.err.println("Record not found");
            else if (dae instanceof BadSqlGrammarException)
                System.err.println("SQL syntax problem");
            else
                System.err.println("other internal probelm");
            dae.printStackTrace();
        }
        ((AbstractApplicationContext) ctx).close();
    }
}

```

Mapping SQL Operations as Sub classes

Another limitation with JdbcTemplate/ NamedParameterJdbcTemplate/
SimpleJdbcTemplate:

in DAO class

```

public String getEmpNameByNo (int no) {
    string name = jt.queryForObject("SELECT ENAME FROM EMP
                                   WHERE EMPNO=?", String.class, no);
    return name;
}

```

If the above DAO method/ jt.queryForObject(-,-,-) is called for multiple times then

- a. Gathers the injected DS obj from JdbcTemplate object for multiple times. (ok)

- b. Gathers JDBC connection object from JDBC connection pool for multiple times. (ok)
- c. makes the given SQL query as pre-compiled Query for multiple times by creating PreparedStatement object for multiple times. (no ok)
 - making the same SQL query as pre-compiled SQL query for multiple times is unnecessary and also degrades the performance.
- d. Sets the values to query param for multiple times and executes query for multiple times. (ok)
- e. Process/ convert the results for multiple times. (ok)

✚ To overcome the above problem use "Mapping SQL Operations as Cub classes" approach, which says for every SQL query develop one sub class extending `SqlQuery<T>` (AC) (for select query) or from `SqlUpdate<T>` for non-select query. We generally these sub classes as inner classes in the DAO class.

✚ In these sub classes we give DS, SQL query to their super classes class (`SqlQuery<T>/ SqlUpdate` classes) only 1 for time, so that collecting connection object from JDBC connection pool, creating PreparedStatement object having given query as pre-compiled SQL query happens only for 1 time and sub classes objects start representing pre-compiled queries, so that DAO class methods can use the objects of sub classes for multiple times to execute the pre-compiled SQL queries for multiple times.

✚ `SqlQuery<T>` (AC) is having more abstract methods to implement. So, prefer using which having a smaller number of abstract methods to implement.

```
java.lang.Object
  org.springframework.jdbc.object.RdbmsOperation
    org.springframework.jdbc.object.SqlOperation
      org.springframework.jdbc.object.SqlQuery<T>
        org.springframework.jdbc.object.MappingSqlQueryWithParameters<T>
          org.springframework.jdbc.object.MappingSqlQuery<T>
```

✚ On the each select SQL query related sub class obj of `SqlQuery<T>/ MappingSqlQuery<T>` we can call

- a. `List<T> execute(...)`: If the Select Query gives bunch of records.
- b. `<T> findObject(...)`: If the Select Query gives single record.

DAO class

```
@Repository("studDAO")
public class StudentDAOImpl implements StudentDAO {
    private static final String GET_STUDENT_BY_ADDRS = "SELECT SNO,
        SNAME, SADD, AVG FROM STUDENT WHERE SADD=?";

    private StudentSelector1 selector1;

    @Autowired
    public StudentDAOImpl(DataSource ds){
        selector1 = new StudentSelector1(ds,
            GET_STUDENT_BY_ADDRS);
    } //Constructor

    public List<StudentBO> getStudentsByAddrs(String addrs) {
        List<StudentBO> listBO=selector1. execute(addrs);
        return listBO;
    } //method

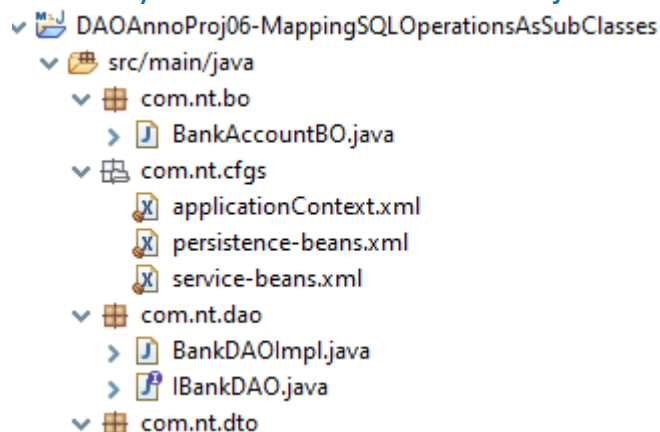
    //sub class as inner class in DAO
    private class StudentSelector1 extends MappingSqlQuery<StudentBO>
    {
        //constructor
        public StudentSelector1(DataSource ds, String query) {
            super(ds,query);
            super.declareParameter(new
                SqlParameter(Types.VARCHAR));
            //registrering param(?) with JDBC data type
            super.compile();
        }
        public StudentBO mapRow(ResultSet rs, int rowNum) throws
            SQL Exception {
            //convert RS record to BO class obj
            StudentBO bo=new StudentBO();
            bo.setSno(rs.getInt(1));
            bo.setSname(rs.getString(2));
            bo.setSadd(rs.getString(3));
            bo.setAvg(rs.getFloat(4));
            return bo;
        } //mapRow(-,-)
    }
}
```

```
    } //inner class  
} //DAO class
```

Flow of execution:

- IoC container creation --> pre-instantiation of singleton scope beans so DS, DAO classes pre-instantiated and DS is injected to DAO --> In that process DAO constructor executes and calls sub class cum inner class (StudentSelector1) constructor due to this sub class cum inner class (StudentSelector1) gives DS, query to its super class (MappingSqlQuery) only for 1 time and creates PreparedStatement object by making given SQL query as pre-compiled Query because of super.compile() only for 1 time [At the end the sub class cum inner class (StudentSelector1) represents pre-compiled SQL query].
- Service class method calls DAO method (getStudentsByAdrs(-)) for multiple times, so selector1.execute(-) also called for multiple time --> In this process values to query params will be set for multiple times query execution takes place for multiple times gathering RS object processing that object to ListBO by calling mapRow (-, -) takes place for multiple times --> returns ListBO back to DAO class method for multiple times.
 - a. Gathering and using DS happens for 1 time. (OK)
 - b. Gathering jdbc con object from JDBC connection pool happens for 1 time. (OK)
 - c. Creating PreparedStatement object by making the SQL query as pre-compiled SQL query happens for 1 time. (OK)
 - d. Setting values query params and executing query happens for multiple times. (OK)
 - e. Gathering results and processing results happens for multiple times. (OK)

Directory Structure of DAOAnnoProj06-MappingSQLOperationsAsSubClasses:



```

> BankAccountDTO.java
▼ com.nt.service
  > BankMgmtServiceImpl.java
  > IBankMgmtService.java
▼ com.nt.test
  > MappingSQLOperationTest.java
src/test/java
> JRE System Library [jdk-13.0.1]
> Maven Dependencies
> src
> target
pom.xml

```

- Develop the above directory structure using maven setup and package, class, XML file and add the jar dependencies to pom.xml, then use the following code with in their respective file and Change to Java 1.9 or its higher version.
- Copy the xml files and pom.xml from previous project.
- Remove the any JDBC template configuration because her we need only DataSource.

BankAccountBO.java

```

package com.nt.bo;
import lombok.Data;
@Data
public class BankAccountBO {
    private Long accNo;
    private String holderName;
    private Float balance;
    private String status;
}

```

BankAccountDTO.java

```

package com.nt.dto;
import lombok.Data;
@Data
public class BankAccountDTO {
    private Long accNo;
    private String holderName;
    private Float balance;
    private String status;
}

```

IBankDAO.java

```
package com.nt.dao;

import java.util.List;

import com.nt.bo.BankAccountBO;

public interface IBankDAO {
    public List<BankAccountBO> getBankAccountsByBalanceRange(float
start, float end);
    public BankAccountBO getBankAccountByAccNo(long accno);
    public int updateBankAccountsWithBonusByBalance(float bonus,
float maxBalance);
}
```

BankDAOImpl.java

```
package com.nt.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.asm.Type;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;
import org.springframework.jdbc.object.SqlUpdate;
import org.springframework.stereotype.Repository;

import com.nt.bo.BankAccountBO;

@Repository("bankDAO")
public class BankDAOImpl implements IBankDAO {

    private static final String GET_ACCOUNT_BY_BAL_RANGE="SELECT
ACCNO, HOLDERNAME, BALANCE, STATUS FROM BANK_ACCOUNT WHERE
BALANCE>=? AND BALANCE<=?";

    private static final String GET_ACCOUNT_BY_ACCNO="SELECT
ACCNO, HOLDERNAME, BALANCE, STATUS FROM BANK_ACCOUNT WHERE
ACCNO=?";

    private static final String UPDATE_BONUS_BY_BALANCE="UPDATE
BANK_ACCOUNT SET BALANCE=BALANCE+? WHERE BALANCE<?";
```

```

BankAccountSelectorByBalRange selector1;
BankAccountSelectorByAccNo selector2;
BankAccountUpdaterByBalance updator1;

@Autowired
public BankDAOImpl(DataSource ds) {
    selector1 = new BankAccountSelectorByBalRange(ds,
GET_ACCOUNT_BY_BAL_RANGE);
    selector2 = new BankAccountSelectorByAccNo(ds,
GET_ACCOUNT_BY_ACCNO);
    updator1 = new BankAccountUpdaterByBalance(ds,
UPDATE_BONUS_BY_BALANCE);
}

@Override
public List<BankAccountBO> getBankAccountsByBalanceRange(float
start, float end) {
    List<BankAccountBO> listBO = selector1.execute(start, end);
    return listBO;
}

private static class BankAccountSelectorByBalRange extends
MappingSqlQuery<BankAccountBO> {

    public BankAccountSelectorByBalRange(DataSource ds, String
query) {

        super(ds, query); // gives DS, query to super class
        //register query param (?) with JDBC type
        super.declareParameter(new
SqlParameter(Type.FLOAT));
        super.declareParameter(new
SqlParameter(Type.FLOAT));
        //make super class compiler query
        super.compile();
    }

    @Override
    public BankAccountBO mapRow(ResultSet rs, int rowNum)
throws SQLException {
        // convert Rs records into BO class object
        BankAccountBO bo = new BankAccountBO();
        bo.setAccNo(rs.getLong(1));
    }
}

```

```

        bo.setHolderName(rs.getString(2));
        bo.setBalance(rs.getFloat(3));
        bo.setStatus(rs.getString(4));
        return bo;
    } //mapRow(-,-)
} //inner class

/*-----*/

@Override
public BankAccountBO getBankAccountByAccNo(long accno) {
    BankAccountBO bo = selector2.findObject(accno);
    return bo;
}

private static class BankAccountSelectorByAccNo extends
MappingSqlQuery<BankAccountBO> {

    public BankAccountSelectorByAccNo(DataSource ds, String
query) {

        super(ds, query); // gives DS, query to super class
        //register query param (?) with JDBC type
        super.declareParameter(new
SqlParameter(Type.LONG));
        //make super class compiler query
        super.compile();
    }

    @Override
    public BankAccountBO mapRow(ResultSet rs, int rowNum)
throws SQLException {
        // convert Rs records into BO class object
        BankAccountBO bo = new BankAccountBO();
        bo.setAccNo(rs.getLong(1));
        bo.setHolderName(rs.getString(2));
        bo.setBalance(rs.getFloat(3));
        bo.setStatus(rs.getString(4));
        return bo;
    } //mapRow(-,-)
} //inner class

//-----

```

```

@Override
public int updateBankAccountsWithBonusByBalance(float bonus,
float maxBalance) {
    int count = updator1.update(bonus, maxBalance);
    return count;
}

private static class BankAccountUpdaterByBalance extends
SqlUpdate {

    public BankAccountUpdaterByBalance(DataSource ds, String
query) {

        super(ds, query); // gives DS, query to super class
        //register query param (?) with JDBC type
        super.declareParameter(new
SqlParameter(Type.FLOAT));
        super.declareParameter(new
SqlParameter(Type.FLOAT));
        //make super class compiler query
        super.compile();
    }

} //inner class

} //outer class

```

IBankMgmtService.java

```

package com.nt.service;

import java.util.List;

import com.nt.dto.BankAccountDTO;

public interface IBankMgmtService {
    public List<BankAccountDTO>
fetchBankAccountsByBalanceRange(float start, float end);
    public BankAccountDTO fetchBankAccountsByAccNo(long accno);
    public String addBonusToBankAccountsByBalance(float bonus, float
maxBalance);
}

```


BankMgmtServiceImpl.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.stereotype.Service;

import com.nt.bo.BankAccountBO;
import com.nt.dao.IBankDAO;
import com.nt.dto.BankAccountDTO;

@Service("bankService")
public class BankMgmtServiceImpl implements IBankMgmtService {

    @Autowired
    private IBankDAO dao;

    @Override
    public List<BankAccountDTO>
fetchBankAccountsByBalanceRange(float start, float end) {
    //use DAO
    List<BankAccountBO> listBO =
dao.getBankAccountsByBalanceRange(start, end);
    //convert listBO to listDTO
    List<BankAccountDTO> listDTO = new ArrayList<>();
    listBO.forEach(bo->{
        BankAccountDTO dto = new BankAccountDTO();
        BeanUtils.copyProperties(bo, dto);
        listDTO.add(dto);
    });
    return listDTO;
}

    @Override
    public BankAccountDTO fetchBankAccountsByAccNo(long accno) {
        // use dao
        BankAccountBO bo = dao.getBankAccountByAccNo(accno);
        //convert DTO to bo
        BankAccountDTO dto = new BankAccountDTO();
        BeanUtils.copyProperties(bo, dto);
    }
}
```

```

        return dto;
    }

    @Override
    public String addBonusToBankAccountsByBalance(float bonus, float
maxBalance) {
        //use dao
        int count =
dao.updateBankAccountsWithBonusByBalance(bonus, maxBalance);
        return count==0?"No accounts found for add bonus":count+"
No. of records are added with bonus:"+bonus;
    }
}

```

IBankMgmtService.java

```

package com.nt.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.BadSqlGrammarException;

import com.nt.service.IBankMgmtService;

public class MappingSQLOperationTest {

    public static void main(String[] args) {
        // Create IoC container
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("com/nt/cfgs/applicationContext.xml");
        // get Service class object
        IBankMgmtService service = ctx.getBean("bankService",
IBankMgmtService.class);
        // invoke method
        try {
            System.out.println("Bank ACcounts having balance
Range 10000 to 20000");
            service.fetchBankAccountsByBalanceRange(10000,
20000).forEach(System.out::println);

```

```

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

        System.out.println(service.fetchBankAccountsByAccNo(33232));
        System.out.println("-----");

        System.out.println(service.addBonusToBankAccountsByBalance(500,
10000));
        } catch (DataAccessException dae) {
            if (dae instanceof EmptyResultDataAccessException)
                System.err.println("Record not found");
            else if (dae instanceof BadSqlGrammarException)
                System.err.println("SQL syntax problem");
            else
                System.err.println("other internal probelm");
            dae.printStackTrace();
        }
        ((AbstractApplicationContext) ctx).close();
    }
}

```

Working with properties file and yml/ yaml files in Spring/ Spring Boot

- ✚ We can read inputs from properties file/ yml file to Spring bean properties in two ways
 - a. using @Value (given spring framework 3.0) (Does not support bulk reading)
 - We should add on the top of each property.
 - Reading values into array/list/set/map and HAS-A Object is complex (not recommended to do).
 - Property name in bean class and key in properties/ yml file need to not match.

yml -> Yiant markup language/ yamaling markup language.
 yaml -> yet another markup language.
 A different approach of maintaining key-values pairs.

Note:

- ✓ In spring boot application application.properties/ yml file will be detected and loaded automatically as part application flow from

src/main/resources folder.

- ✓ The properties/ yml files having other name or location must be configured explicitly using @PropertySource annotation.

application.properties

```
per.info.id=101  
per.info.name=raja
```

@Component

@Data

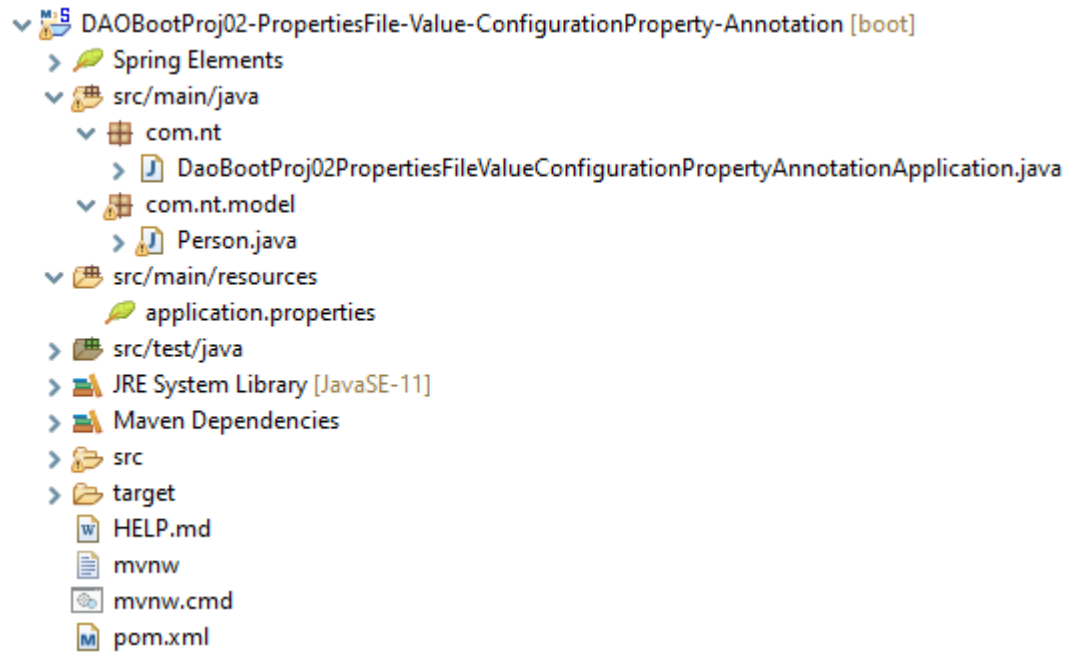
```
public class Person {  
    @Value("${per.info.id}")  
    private int pid;  
    @Value("${per.info.name}")  
    private String pname;  
}
```

b. @ConfigurationProperties (supports Bulk reading)

- Given by spring boot 1.0.
- Allows to read values into simple, array/ list/ set/ map, HAS-A object properties.
- We need to apply only the top of Spring bean class by specifying prefix. So, values will be bound to Spring bean class properties at once.
- Here keys in properties/ yml file must match with Spring bean class property names.
- All keys in properties file must have common prefix and that common prefix must be specified in @ConfigurationProperties (prefix="....").
- In properties file the allowed special characters in keys are ".", "-", "_".

Directory Structure of DAOBootProj02-PropertiesFile-Value-ConfigurationProperty-Annotation:

- Develop the below directory structure using Spring Starter Project option and package and classes also.
- Many jars' dependencies will come automatically in pom.xml because while developing Spring Starter Project we choose some jars. They are
 - Lombok API
- Then use the following code with in their respective file.



Person.java

```
package com.nt.model;

import java.util.List;
import java.util.Map;
import java.util.Set;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import lombok.Data;

@Data
@Component("per")
@ConfigurationProperties(prefix = "per.info")
public class Person {
    private int id;
    private String name;
    private int[] marks1;
    private List<Integer> marks2;
    private Set<Integer> marks3;
    private Map<String, Integer> phones;
}
```

application.properties

```
#Simple properties
per.info.id=101
per.info.no=102
per.info.name=rakesh

#Arrays [prefix.var[index]=value]
per.info.marks1[0]=40
per.info.marks1[1]=50
per.info.marks1[2]=60

#List Collection [prefix.var[index]=value]
per.info.marks2[0]=50
per.info.marks2[1]=60
per.info.marks2[2]=70

#Set Collection [prefix.var[index]=value]
per.info.marks3[0]=60
per.info.marks3[1]=70
per.info.marks3[2]=80

#Map Collection [prefix.var.key=value]
per.info.phones.residence=99999999
per.info.phones.office=88888888
per.info.phones.personal=77777777
```

DaoBootProj02PropertiesFileValueConfigurationPropertyAnnotationApplication.java

```
package com.nt;

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

import com.nt.model.Person;

@SpringBootApplication
public class
DaoBootProj02PropertiesFileValueConfigurationPropertyAnnotationApplica
tion {

    public static void main(String[] args) {
        ApplicationContext ctx = null;
    }
}
```

```

        //get IoC container
        ctx =
SpringApplication.run(DaoBootProj02PropertiesFileValueConfigurationProp
ertyAnnotationApplication.class, args);
        //get Spring bean object
        Person pers = ctx.getBean("per", Person.class);
        System.out.println(pers);
    }
}

```

- ✚ On certain bean property of spring bean class if we place both @Value, @ConfigurationProperties (indirectly from top of the class) effect with two different keys and value the @ConfigurationProperties value will be taken as the final value.

application.properties

```

per.info.id=101
per.info.no=102

```

Person.java

```

@Data
@Component("per")
@ConfigurationProperties(prefix = "per.info")
public class Person {
    @Value("${per.info.no}")
    private int id;
}

```

Note: While working with @ConfigurationProperties it is recommended to add the following dependency in pom.xml file to generate Metadata about entries/ keys of properties/ yml file. Due to this all warnings in properties will go off.

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-
processor</artifactId>
    <optional>true</optional>
</dependency>

```

For working with HAS-A relation:

Create a addition class in com.nt.model package Job.java and write the following code in their respective files.

application.properties

```
#Has-A Relation objecy type property [prefix.HAS-A-var.var=value]
per.info.job.company=HCL
per.info.job.desg=programmer
per.info.job.deptno=9001
per.info.job.salary=34565.6
```

Job.java

```
package com.nt.model;

import lombok.Data;

@Data
public class Job {
    private String company;
    private String desg;
    private int deptno;
    private float salary;
}
```

Person.java

```
@Data
@Component("per")
@ConfigurationProperties(prefix = "per.info")
public class Person {

    private int id;
    private String name;
    private String address;
    private int[] marks1;
    private List<Integer> marks2;
    private Set<Integer> marks3;
    private Map<String, Integer> phones;
    private Job job; //HAS-A relation property

}
```


- ✚ In most cases we use pre-defined keys and their values in application.properties to provide instructions/ inputs related to autoconfiguration.

- ✚ The Beans of AutoConfiguration internally use this @ConfigurationProperties to read values from properties.

Example

```
@ConfigurationProperties(prefix = "spring.datasource")
public class DataSourceProperties implements BeanClassLoaderAware,
InitializingBean {
    .....
}
```

- ✚ While preparing element values to array/ list/ set collection inline syntax in properties file as shown below

application.properties

```
#Array/ Set/ List Collection [prefix.var[index]=value1,value2,value3,...]
per.info.marks1=40,50,60
```

YML/ YAML

- ✚ Yaml markup language/ Yamaling markup language (yaml).
- ✚ Yet Another Markup language (YAML).
- ✚ Alternate to properties file, very useful when lengthy keys at same level because it avoids duplicates from the keys by maintaining key and values in hierarchy manner.

application.properties

```
info.per.id=101
info.per.name=raja
info.per.address=hyd
```

The word "info.per" is repeated for multiple times in the keys there is duplication in the keys.

application.yml

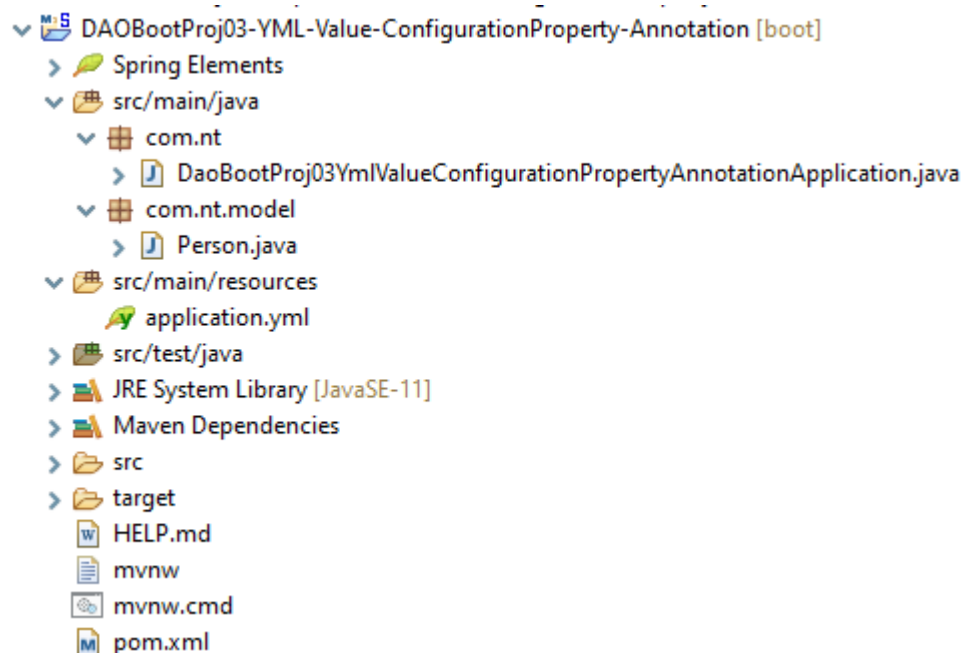
```
info:  level1 node
per:   level2 node
  id: 101
  name: raja
  address: hyd
```

Here the word "info.per" is not repeated in the keys by maintaining data as hierarchical data.

- ✚ Extension can be, .yaml or .yml.

- ✚ Spring boot internally uses "snack yaml API" to parse and convert yaml file into properties file.
- ✚ While writing "nodes" in yaml file you must give minimum one space and allowed special symbols in the keys are "_", "-", ".".
- ✚ Same level nodes must start at same place (same column number in the file) if not errors will come (this indicates we must maintain proper indentation).
- ✚ Both application.yml or application.properties will be detected and loaded by Spring boot automatically during the application startup from main/java/resources folder.
- ✚ We can bind yaml file data to Spring bean class properties/ variables either using @Value (given by spring) or using @ConfigurationProperties (given Spring Boot) annotations.
- ✚ Yml files are node based, space sensitive and indentation-based files.

Directory Structure of DAOBootProj03-YML-Value-ConfigurationProperty-Annotation:



- Develop the below directory structure using Spring Starter Project option and package and classes also.
- Many jars' dependencies will come automatically in pom.xml because while developing Spring Starter Project we choose some jars. They are
 - Lombok API
- Change the application.properties to application.yml.
- Then use the following code with in their respective file.

Person.java

```
package com.nt.model;

import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

import lombok.Data;

@Data
@Component("per")
@ConfigurationProperties(prefix = "info.per")
public class Person {
    private int id;
    private String name;
    private String address;
}
```

application.yml

```
info:
  per:
    id: 101
    name: raja
    address: hyd
```

DaoBootProj03YmlValueConfigurationPropertyAnnotationApplication.java

```
package com.nt;

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

import com.nt.model.Person;

@SpringBootApplication
public class
DaoBootProj03YmlValueConfigurationPropertyAnnotationApplication {

    public static void main(String[] args) {

        ApplicationContext ctx = null;
        //get IoC container
        ctx =
```

```

SpringApplication.run(DaoBootProj03YmlValueConfigurationProperty
AnnotationApplication.class, args);
    //get Spring bean object
    Person pers = ctx.getBean("per", Person.class);
    System.out.println(pers);
}
}

```

application.properties

```

info.per.id=101
info.per.name=raja
info.job.desg=clerk
info.job.salary=9000
company.location=hyd
company.name=HCL

```

application.yml

```

info:
  per:
    id: 101
    name: raja
  job:
    desg: clerk
    salary: 9000
company:
  location: hyd
  name: HCL

```

Note: # symbol in properties file, yml file indicates comment.

Array/ List/ Set Collection:

application.properties

```

info.per.marks[0]=60
info.per.marks[1]=70
info.per.marks[2]=80

```

These array elements:

60	70	80
0	1	2

application.yml

```

info:
  per:
    marks:
      - 60
      - 70
      - 80

```

Map/ Properties Collection:

application.properties

#prefix.var.key=value

info.per.phones.residence=999999

info.per.phones.office=88888888

info.per.phones.personal=7777777

application.yml

info:

per:

phones:

residence: 999999

office: 888888

personal: 777777

acts keys and values in map collection

phones (Map collection)

residence	999999
office	888888
personal	777777

keys

values

Person.java

```
@Data
@Component("per")
@ConfigurationProperties(prefix = "info.per")
public class Person {
    private int id;
    private String name;
    private String address;
    private int marks1[];
    private List<Integer> marks2;
    private Set<Integer> marks3;
    private Map<String, Object> phones;
}
```

application.yml

```
info:
  per:
    #Simple properties
    id: 101
    name: raja
    address: hyd
    #Array
    marks1:
      - 30
```

```

- 40
- 50
#List
marks2:
- 40
- 50
- 60
#set
marks3:
- 50
- 60
- 70
#Map
phones:
  residence: 9999999
  office: 888888
  personal: 777777

```

- ✚ Use properties file if the keys are smaller and the nodes/ prefixes are no repeating.
- ✚ Use yml file if the keys are lengthy and the nodes/ prefixes are repeating.

For HAS-A relation/ Object type property:

Person.java

```

package com.nt.model;
import lombok.Data;
@Data
public class Job {
    private String desg;
    private float salary;
    private String company;
    private String skills[];
}

```

Person.java

```

public class Person {
    private Job job;
}

```

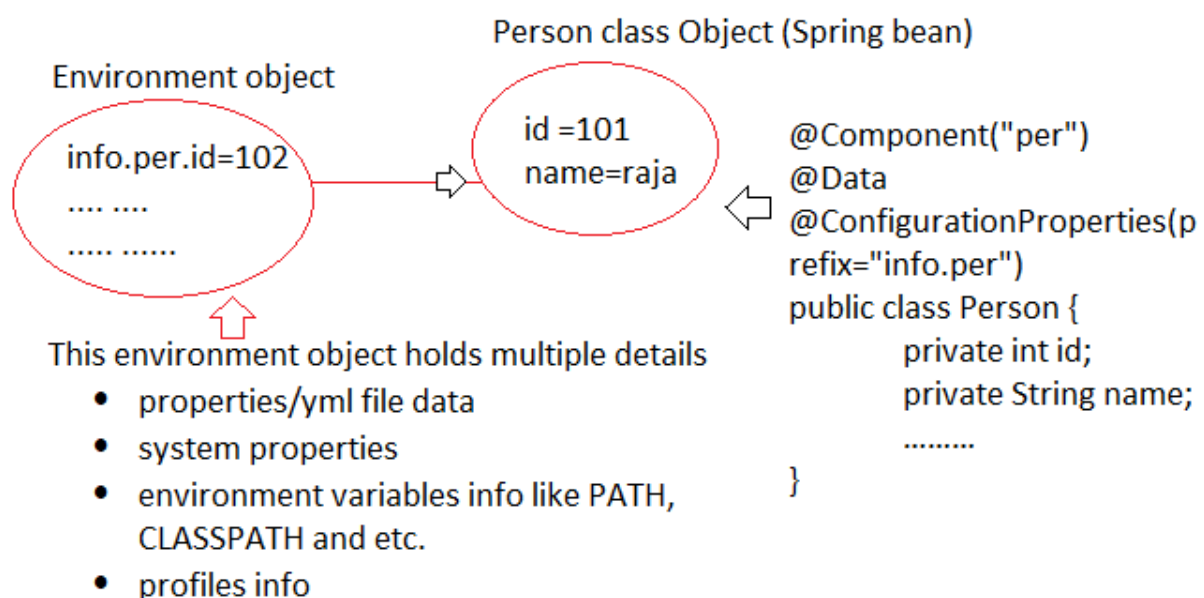
application.yml

```
info:
  per:
    #Reference Type
  job:
    desg: Manager
    salary: 783423.3
    company: HCL
    skills:
      - Java
      - Spring
      - Hibernate
      - J2EE
```

Note: Eclipse IDE there is built-in converter to convert .properties file to .yml file to right click on properties file -> convert .yml file.

Internal flow of @ConfigurationProperties and @Value:

- #1. Spring boot detects and loads application.properties/ yml file [if it yml file it will be converted into properties file internally using snakeyaml]
- #2. Reads keys and values of properties/ yml file into Environment object (InMemory object created in IOC container).
- #3. Collects the values from Environment object and binds to Spring Bean class obj properties based on @Value or @ConfigurationProperties annotation.



Q. If we place both application.properties and application.yml files in spring boot application having same keys and different values then what happens?

Ans. The values kept properties file will be taken as final values.

Note: If certain key is not available in application.properties file, it will be gathered from application.yml

Q. What is the differences and similarities between properties file and yml file.

Ans.

YAML(.YML)	.PROPERTIES
Spec can be found here.	It doesn't really actually have a spec. The closest thing it has to a spec is actually the Javadoc.
Human Readable (both do quite well in human readability).	Human Readable.
Supports key/ value, basically map, List and scalar types (int, string etc.).	Supports key/ value, but doesn't support values beyond the string.
Its usage is quite prevalent in many languages like Python, Ruby, and Java.	It is primarily used in java.
Hierarchical Structure.	Non-Hierarchical Structure.
Spring Framework doesn't support @PropertySources with .yaml files.	Supports @PropertySources with .properties file.
If you are using spring profiles, you can have multiple profiles in one single .yaml file.	Each profile need one separate .properties file.
While retrieving the values from .yaml file we get the value as whatever the respective type (int, string etc.) is in the configuration.	While in case of the .properties files we get strings regardless of what the actual value type is in the configuration.

Q. When should we use .properties or .yaml file?

Ans. If keys are lengthy having multiple common nodes then for yaml files because it avoids the repetition of common nodes otherwise go for properties file.

Q. What is the difference b/w @Value and @ConfigurationProperties?

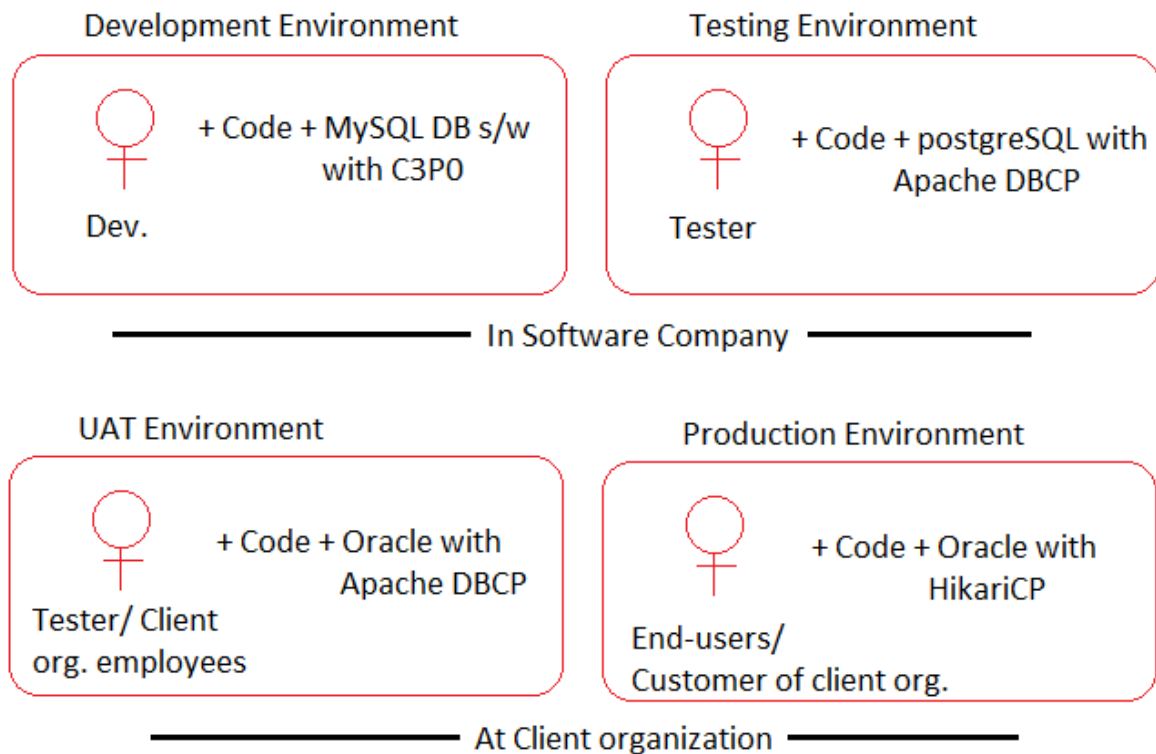
Ans.

@Value	@ConfigurationProperties
a) Given by Spring framework 2.0, So it can be used in both Spring and Spring boot programming.	a) Given by Spring Boot 1.x, So it can be used only in spring boot programming.
b) Useful for reading single value from .properties or yaml file.	b) Useful for reading bulk values by giving common prefix from properties or yaml file.
c) Can be applied at method level and field level, param level and etc. (but not at class level).	c) Can be applied only on class level and method level.
d) Common prefix is not required to read values from properties or yaml files.	d) Common prefix is required and logical operators.
e) Allows to use SPEL (Spring expression language). @Value ("#{2*10}") private int age;	e) Not possible to work with SPEL

Note: SPEL: Allows to work with arithmetic and logic operators

Profiles in Spring or Spring Boot

- ✚ Environment is the setup that required to execute/ test the application/ project
- ✚ For a s/w project we need to have different environments or profiles they are,
 - Development Environment [required in project development]
 - Testing Environment [required in project testing]
 - UAT Environment/ Pilot [required in UAT -> after releasing testing at client side]
 - Production Environment [required in Project live execution] and etc.



- ✚ So, far we writing single application.properties/ yml file in Spring Boot project having the input details. But that properties file must be changed environment to environment or profile to profile as discussed above.
- ✚ Instead of the we can develop multiple properties files for multiple environments/ profiles on 1 per environment/ profile basis and we can activate one environment/ profile based on the requirement.

Syntax:

application-<env/profile name>.properties (or) application-<env/profile>.yml
e.g.:

application.properties/ yml (base/default properties file)

application-dev.properties/yml (for dev env/profile)

application-test.properties/yml (for test env/profile)

application-uat.properties/yml (for uat env/profile)

application-prod.properties/yml (for production env/profile)

- ✚ To make Spring Beans working for certain profile we can use @Profile annotation on the top of stereotype annotation-based spring bean classes or @Bean methods of @Configuration class.

```
@Profile({"uat","prod"})
```

```
@Repository("oraCustDAO")
```

```
public class OracleCustomerDAOImpl implements CustomerDAO {.....}
```

```
@Profile({"dev","test"})
@Repository("mysqlCustDAO")
public class MySQLCustomerDAOImpl implements CustomerDAO {.....}
```

```
@Configuration
@ComponentScan(basePackages="com.nt.dao")
public class PersistenceConfig {
```

```
    @Bean
    @Profile({"uat","test"})
    public DataSource createApacheDBCPDS() {
        ....
    }
```

```
    @Bean
    @Profile("dev")
    public DataSource createC3PODS() {
        ....
    }
```

```
    @Bean
    @Profile("prod")
    public DataSource createHKCPDS() {
        .....
    }
```

```
}
```

To activate specific profile dynamically at runtime:

- a. Using base/ default profile/yml file (best)

[application.properties](#)

spring.profiles.active=dev

[application.yml](#)

spring:

profiles:

active: dev

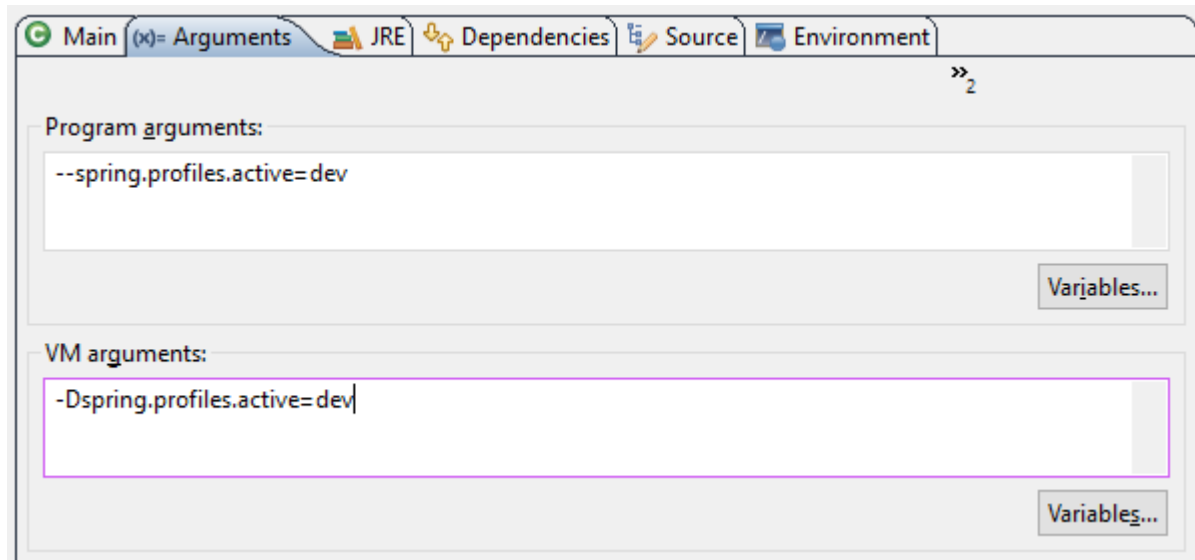
- b. Using command line args (optional args)

--spring.profiles.active=dev

c. Using System properties (VM arguments)

-Dspring.profiles.active=dev

- In eclipse IDE Run As -> Run Configurations -> arguments tab -> program arguments (command line args) -> VM arguments (system properties)



Example App on Spring Profile using Spring Boot:

- a. Keep Spring Boot mini Project ready.
- b. Add additional jars/dependencies in pom.xml
C3P0, Apachedbcp2
- c. Go to DAO classes, write code with JdbcTemplate and also specify @Profile on the top of classes.
- d. Develop multiple properties files for multiple profiles.
- e. Activate one profile from application.properties.

Directory Structure of DAOBootProj04-MiniProject-UsingProfiles:

- Develop the below directory structure using Spring Starter Project option and package and classes also.
- Many jars' dependencies will come automatically in pom.xml because while developing Spring Starter Project we choose some jars. They are
 - JDBC API
 - Oracle Driver
 - MySQL Driver
 - Lombok API
- Then use the following code with in their respective file.

- ▼ DAOBootProj04-MiniProject-UsingProfiles [boot]
 - > Spring Elements
 - ▼ src/main/java
 - ▼ com.nt
 - > DAOBootProj04MiniProjectUsingProfileApplication.java
 - ▼ com.nt.bo
 - > CustomerBO.java
 - ▼ com.nt.controller
 - > MainController.java
 - ▼ com.nt.dao
 - > CustomerDAO.java
 - > MySQLCustomerDAOImpl.java
 - > OracleCustomerDAOImpl.java
 - ▼ com.nt.dto
 - > CustomerDTO.java
 - ▼ com.nt.service
 - > CustomerMgmtService.java
 - > MySQLCustomerMgmtServiceImpl.java
 - > OracleCustomerMgmtServiceImpl.java
 - ▼ com.nt.vo
 - > CustomerVO.java
 - ▼ src/main/resources
 - application-dev.properties
 - application-prod.properties
 - application-test.properties
 - application-uat.properties
 - application.properties
 - > src/test/java
 - > JRE System Library [JavaSE-1.8]
 - > Maven Dependencies
 - > src
 - > target
 - HELP.md
 - mvnw
 - mvnw.cmd
 - pom.xml

CustomerBO.java

```

package com.nt.bo;

import lombok.Data;
@Data
public class CustomerBO {
    private String cname;
    private String cadd;
    private float pAmnt;
    private float interAmt;
}

```

CustomerVO.java

```
package com.nt.vo;

import lombok.Data;

@Data
public class CustomerVO {
    private String cname;
    private String cadd;
    private String pAmt;
    private String time;
    private String rate;
}
```

CustomerDTO.java

```
package com.nt.dto;

import java.io.Serializable;
import lombok.Data;

@Data
public class CustomerDTO implements Serializable{
    private String cname;
    private String cadd;
    private float pAmt;
    private float rate;
    private float time;
}
```

CustomerDAO.java

```
package com.nt.dao;

import com.nt.bo.CustomerBO;

public interface CustomerDAO {
    public int insert(CustomerBO bo) throws Exception;
}
```

MySQLCustomerDAOImpl.java

```
package com.nt.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.nt.bo.CustomerBO;

@Repository("mysqlCustDAO")
public class MySQLCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
CUSTOMER(CNAME, CADD, PAMT, INTERAMT) VALUES(?,?,?,?)";

    @Autowired
    private JdbcTemplate jt;

    @Override
    public int insert(CustomerBO bo) throws Exception {
        int count = jt.update(CUSTOMER_INSERT_QUERY,
bo.getCname(), bo.getCadd(), bo.getPAMnt(), bo.getInterAmt());
        return count;
    }
}
```

OracleCustomerDAOImpl.java

```
package com.nt.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.nt.bo.CustomerBO;

@Repository("oracleCustDAO")
public class OracleCustomerDAOImpl implements CustomerDAO {

    private static final String CUSTOMER_INSERT_QUERY = "INSERT INTO
CUSTOMER_DB VALUES(CNO_SEQ.NEXTVAL,?,?,?,?)";

    @Autowired
    private JdbcTemplate jt;

    @Override
```

```

        public int insert(CustomerBO bo) throws Exception {
            int count = jt.update(CUSTOMER_INSERT_QUERY,
                bo.getCname(), bo.getCadd(), bo.getPAmnt(), bo.getInterAmt());
            return count;
        }
    }
}

```

CustomerMgmtService.java

```

package com.nt.service;

import com.nt.dto.CustomerDTO;

public interface CustomerMgmtService {
    public String calculateSimpleInterestAmount(CustomerDTO dto)
    throws Exception;
}

```

MySQLCustomerMgmtServiceImpl.java

```

package com.nt.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

import com.nt.bo.CustomerBO;
import com.nt.dao.CustomerDAO;
import com.nt.dto.CustomerDTO;

@Service("custService")
@Profile({"dev", "test"})
public final class MySQLCustomerMgmtServiceImpl implements
    CustomerMgmtService {

    @Autowired
    @Qualifier("mysqlCustDAO")
    private CustomerDAO dao;

    @Override
    public String calculateSimpleInterestAmount(CustomerDTO dto)
    throws Exception {
        float interAmt = 0.f;
    }
}

```



```

        CustomerBO bo = null;
        int count = 0;
        //Calculate the simple interest amount from DTO
        interAmt = (dto.getPAMt()*dto.getTime()*dto.getRate())/100;
        //Prepare CustomerBO having persist able Data
        bo = new CustomerBO();
        bo.setCname(dto.getCname());
        bo.setCadd(dto.getCadd());
        bo.setPAMnt(dto.getPAMt());
        bo.setInterAmt(interAmt);
        //use the dao
        count = dao.insert(bo);
        if (count==0)
            return "Customer registration failed - Insert amount is :
"+interAmt;
        else
            return "Customer registration succeded - Insert amount
is : "+interAmt;
    }
}

```

OracleCustomerMgmtServiceImpl.java

```

package com.nt.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Profile;
import org.springframework.stereotype.Service;

import com.nt.bo.CustomerBO;
import com.nt.dao.CustomerDAO;
import com.nt.dto.CustomerDTO;

@Service("custService")
@Profile({"uat", "prod"})
public final class OracleCustomerMgmtServiceImpl implements
CustomerMgmtService {

    @Autowired
    @Qualifier("oracleCustDAO")
    private CustomerDAO dao;
}

```

```

@Override
public String calculateSimpleInterestAmount(CustomerDTO dto)
throws Exception {
    float interAmt = 0.f;
    CustomerBO bo = null;
    int count = 0;
    //Calculate the simple interest amount from DTO
    interAmt = (dto.getPAMt()*dto.getTime()*dto.getRate())/100;
    //Prepare CustomerBO having persist able Data
    bo = new CustomerBO();
    bo.setCName(dto.getCName());
    bo.setCadd(dto.getCadd());
    bo.setPAMnt(dto.getPAMt());
    bo.setInterAmt(interAmt);
    //use the dao
    count = dao.insert(bo);
    if (count==0)
        return "Customer registration failed - Insert amount is :
"+interAmt;
    else
        return "Customer registration succeded - Insert amount
is : "+interAmt;
    }
}

```

MainController.java

```

package com.nt.controller;

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

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

@Controller("controller")
public final class MainController {

    @Autowired
    private CustomerMgmtService service;

    public String processCustomer(CustomerVO vo) throws Exception {

```

```

        CustomerDTO dto = null;
        String result = null;
        //covert Customer VO to customer DTO
        dto = new CustomerDTO();
        dto.setName(vo.getName());
        dto.setCadd(vo.getCadd());
        dto.setPAmt(Float.parseFloat(vo.getPAmt()));
        dto.setTime(Float.parseFloat(vo.getTime()));
        dto.setRate(Float.parseFloat(vo.getRate()));
        //use Service
        result = service.calculateSimpleInterestAmount(dto);
        return result;
    }
}

```

application.properties

```

#Activate Profile
spring.profiles.active=dev

```

application-uat.properties

```

#DataSource configuration For uat. (Oracle, ApacheDBCP)
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

#To change the Defeault DataSource type of autoconfiguration
spring.datasource.type=org.apache.commons.dbcp2.BasicDataSource

```

application-test.properties

```

#DataSource configuration For Test Env. (MySQL, ApacheDBCP)
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///nssp713db
spring.datasource.username=root
spring.datasource.password=root

#To change the Defeault DataSource type of autoconfiguration
spring.datasource.type=org.apache.commons.dbcp2.BasicDataSource

```

application-prod.properties

```
#DataSource configuration For Prod Env. (Oracle, ApacheDBCP)
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
```

application-dev.properties

```
#DataSource configuration For Dev. Env. (MySQL, C3P0)
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///nssp713db
spring.datasource.username=root
spring.datasource.password=root

#To change the Defeualt DataSource type of autoconfiguration
spring.datasource.type=com.mchange.v2.c3p0.ComboPooledDataSource
```

DAOBootProj04MiniProjectUsingProfileApplication.java

```
package com.nt;

import java.util.Scanner;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

@SpringBootApplication
public class DAOBootProj04MiniProjectUsingProfileApplication {
    public static void main(String[] args) {
        ApplicationContext ctx = null;
        MainController controller = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        String result = null;
        Scanner sc = null;
        //Read inputs from end-user using scanner
```

```

        sc = new Scanner(System.in);
        System.out.println("Enter the following Details: ");
        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");
        Amount = sc.next();
        System.out.print("Enter Customer Time : ");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
        vo.setCname(name);
        vo.setCadd(address);
        vo.setPAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //get controller class object
        //get container
        ctx =
SpringApplication.run(DAOBootProj04MiniProjectUsingProfileApplication.cl
ass, args);
        //get controller class
        controller = ctx.getBean("controller", MainController.class);
        //invoke methods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("Internal probelm :
"+e.getMessage());
            e.printStackTrace();
        }
        //close container
        ((AbstractApplicationContext) ctx).close();
    }
}

```

Taking Spring Boot Profile as YML files:

Directory Structure of DAOBootProj05-MiniProject-UsingProfiles-YML:

- ✚ Copy paste the DAOBootProj04-MiniProject-UsingProfiles with name DAOBootProj05-MiniProject-UsingProfiles-YML.
- ✚ And covert .properties file to .yml file.

application.yml

```
spring:
  profiles:
    active: dev
```

application-uat.yml

```
spring:
  datasource:
    driver-class-name: oracle.jdbc.driver.OracleDriver
    password: manager
    type: org.apache.commons.dbcp2.BasicDataSource
    url: jdbc:oracle:thin:@localhost:1521:xe
    username: system
```

application-test.yml

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    type: org.apache.commons.dbcp2.BasicDataSource
    url: jdbc:mysql:///nssp713db
    username: root
```

application-prod.yml

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

application-dev.yml

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    type: com.mchange.v2.c3p0.ComboPooledDataSource
    url: jdbc:mysql:///nssp713db
    username: root
```

Directory Structure of DAOBootProj06-MiniProject-UsingProfiles-SingleYML:

- ✚ Copy paste the DAOBootProj05-MiniProject-UsingProfiles-YML with name DAOBootProj06-MiniProject-UsingProfiles-SingleYML.
- ✚ Add the following code in their respective file and except application.yml file delete all yml files

application.yml

```
#Dev
spring:
  profiles: dev
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    type: com.mchange.v2.c3p0.ComboPooledDataSource
    url: jdbc:mysql:///nssp713db
    username: root
---    #Acts as a separator
#Prod
spring:
  profiles: prod
  datasource:
    driver-class-name: oracle.jdbc.driver.OracleDriver
    password: manager
    url: jdbc:oracle:thin:@localhost:1521:xe
    username: system
---
#Test
spring:
  profiles: test
  datasource:
```

```

driver-class-name: com.mysql.cj.jdbc.Driver
password: root
type: org.apache.commons.dbcp2.BasicDataSource
url: jdbc:mysql:///nssp713db
username: root
---
#Uat
spring:
  profiles: uat
  datasource:
    driver-class-name: oracle.jdbc.driver.OracleDriver
    password: manager
    type: org.apache.commons.dbcp2.BasicDataSource
    url: jdbc:oracle:thin:@localhost:1521:xe
    username: system
---
#Base
spring:
  profiles:
    active: dev

```

Working with Profiles in 100%Code driven configurations:

Step 1: Keep 100% code driven configurations Mini Project ready

Step 2: Add Apache dbcp2, C3P0, HikariCP jars build.gradle or pom.xml file

Step 3: Make sure that DAO classes are linked to profiles properly using

Step 4: Develop PersistenceConfig.java class having @Bean methods linked with Profiles.

Note: If you don't put Spring bean in any profile then it will be used for all profiles in our mini project, we can use all Service, Controller classes without placing in profiles to make them common for all profiles.

Step 5: Activate Profile from client application.

Directory Structure of DAO100pProj02-MiniProject-Profiles:

- Develop the below directory structure using gradle setup and package, class, XML file and add the jar dependencies to build.gradle then use the following code with in their respective file.
- Rest of code copy from previous project

- ▼ DAO100pProj02-MiniProject-Profiles
 - > Spring Elements
 - ▼ src/main/java
 - ▼ com.nt.bo
 - > CustomerBO.java
 - ▼ com.nt.config
 - > AppConfig.java
 - > ControllerConfig.java
 - > PersistenceConfig.java
 - > ServiceConfig.java
 - ▼ com.nt.controller
 - > MainController.java
 - ▼ com.nt.dao
 - > CustomerDAO.java
 - > MySQLCustomerDAOImpl.java
 - > OracleCustomerDAOImpl.java
 - ▼ com.nt.dto
 - > CustomerDTO.java
 - ▼ com.nt.service
 - > CustomerMgmtService.java
 - > CustomerMgmtServiceImpl.java
 - ▼ com.nt.test
 - > RealTimeDITest.java
 - ▼ com.nt.vo
 - > CustomerVO.java
 - > src/main/resources
 - > src/test/java
 - > src/test/resources
 - > JRE System Library [JavaSE-13]
 - > Project and External Dependencies
 - > bin
 - > gradle
 - > src
 - build.gradle

AppConfig.java

```

package com.nt.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;

@Configuration
@Import({PersistenceConfig.class, ServiceConfig.class,
ControllerConfig.class})
public class AppConfig {

}

```

ControllerConfig.java

```
package com.nt.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.nt.controller")
public class ControllerConfig {

}
```

PersistenceConfig.java

```
package com.nt.config;

import javax.sql.DataSource;

import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.core.JdbcTemplate;

import com.mchange.v2.c3p0.ComboPooledDataSource;
import com.zaxxer.hikari.HikariDataSource;

@Configuration
@ComponentScan(basePackages = "com.nt.dao")
public class PersistenceConfig {

    @Bean
    @Profile("dev")
    public DataSource createC3P0DS() throws Exception {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass("com.mysql.cj.jdbc.Driver");
        ds.setJdbcUrl("jdbc:mysql:///nssp713db");
        ds.setUser("root");
        ds.setPassword("root");
        return ds;
    }

    @Bean
    @Profile("test")
```

```

public DataSource createApacheDBCPDSMySQL() throws Exception {
    BasicDataSource bds = new BasicDataSource();
    bds.setDriverClassName("com.mysql.cj.jdbc.Driver");
    bds.setUrl("jdbc:mysql:///nssp713db");
    bds.setUsername("root");
    bds.setPassword("root");
    return bds;
}

@Bean
@Profile("uat")
public DataSource createApacheDBCPDSOracle() throws Exception {
    BasicDataSource bds = new BasicDataSource();
    bds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
    bds.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
    bds.setUsername("system");
    bds.setPassword("manager");
    return bds;
}

@Bean
@Profile("prod")
public DataSource createHikariCPDS() throws Exception {
    HikariDataSource hds = new HikariDataSource();
    hds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
    hds.setJdbcUrl("jdbc:oracle:thin:@localhost:1521:xe");
    hds.setUsername("system");
    hds.setPassword("manager");
    return hds;
}

@Bean
@Profile("dev")
public JdbcTemplate createJTUsingC3PODS() throws Exception {
    return new JdbcTemplate(createC3PODS());
}

@Bean
@Profile("test")
public JdbcTemplate createJTUsingApacheDPCPDSWithMySQL()
throws Exception {
    return new JdbcTemplate(createApacheDBCPDSMySQL());
}

```

```

    }

    @Bean
    @Profile("uat")
    public JdbcTemplate createJTUsingApacheDPCPDSWithOracle()
    throws Exception {
        return new JdbcTemplate(createApacheDBCPDSOracle());
    }

    @Bean
    @Profile("prod")
    public JdbcTemplate createJTUsingHikariCPDS() throws Exception {
        return new JdbcTemplate(createHikariCPDS());
    }
}

```

ServiceConfig.java

```

package com.nt.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.nt.service")
public class ServiceConfig {
}

```

MySQLCustomerDAOImpl.java

```

@Repository("mysqlCustDAO")
@Profile({"dev", "test"})
public class MySQLCustomerDAOImpl implements CustomerDAO {
}

```

OracleCustomerDAOImpl.java

```

@Repository("mysqlCustDAO")
@Profile({"uat", "prod"})
public class OracleCustomerDAOImpl implements CustomerDAO {
}

```

CustomerMgmtServiceImpl.java

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

    @Autowired
    private CustomerDAO dao;
```

RealTimeDITest.java

```
package com.nt.test;

import java.util.Scanner;

import
org.springframework.context.annotation.AnnotationConfigApplicationCont
ext;
import org.springframework.core.env.ConfigurableEnvironment;

import com.nt.config.AppConfig;
import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter the following Details for registration :
");
        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");
        Amount = sc.next();
```

```

        System.out.print("Enter Customer Time : ");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
        vo.setCname(name);
        vo.setCadd(address);
        vo.setPAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //Create BeanFactory [IoC] container
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext();
        //get Environment object from IoC container
        ConfigurableEnvironment env = ctx.getEnvironment();
        env.setActiveProfiles("dev");
        ctx.register(AppConfig.class);
        ctx.refresh();
        //get controller class object
        controller = ctx.getBean("controller", MainController.class);
        //invoke methods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        } catch (Exception e) {
            System.out.println("Internal problem :
"+e.getMessage());
            e.printStackTrace();
        }
    } //main
} //class

```

build.gradle

```

plugins {
    // Apply the java-library plugin to add support for Java Library
    id 'application'
}
mainClassName = 'com.nt.test.RealTimeDITest'

```

```

run {
    standardInput = System.in
}
repositories {
    mavenCentral()
}
dependencies {
    // https://mvnrepository.com/artifact/org.springframework/spring-
context-support
    implementation group: 'org.springframework', name: 'spring-context-
support', version: '5.2.8.RELEASE'
    // https://mvnrepository.com/artifact/org.springframework/spring-
jdbc
    implementation group: 'org.springframework', name: 'spring-jdbc',
version: '5.2.8.RELEASE'
    //
https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc6
    implementation group: 'com.oracle.database.jdbc', name: 'ojdbc6',
version: '11.2.0.4'
    // https://mvnrepository.com/artifact/mysql/mysql-connector-java
    implementation group: 'mysql', name: 'mysql-connector-java',
version: '8.0.21'
    // https://mvnrepository.com/artifact/com.zaxxer/HikariCP
    implementation group: 'com.zaxxer', name: 'HikariCP', version: '3.4.5'
    // https://mvnrepository.com/artifact/org.projectlombok/lombok
    implementation group: 'org.projectlombok', name: 'lombok', version:
'1.18.16'
    // https://mvnrepository.com/artifact/com.mchange/c3p0
    implementation group: 'com.mchange', name: 'c3p0', version:
'0.9.5.5'
    //
https://mvnrepository.com/artifact/org.apache.commons/commons-dbcp2
    implementation group: 'org.apache.commons', name: 'commons-
dbcp2', version: '2.8.0'
}

```

```

org.sf.core.env.Enviroment(I)
|      extends
org.sf.core.env.ConfigurableEnviroment (I)

```

Note: Environment object is IoC Container maintained internal object having profiles info, properties file info, system properties info and environment variable info.

Directory Structure of DAO100pProj03-MiniProject-Profiles-PropertiesFile:

- ✚ Copy paste the DAO100pProj02-MiniProject-Profiles application and change rootProject.name to DAO100pProj03-MiniProject-Profiles-PropertiesFile in settings.gradle file.
- ✚ Add com.nt.commons package with jdbc.properties file
- ✚ Add the following code in applicationContext.xml as part of the replace for the particular section.

jdbc.properties

```
#For MySQL
jdbc.mysql.driverclass=com.mysql.cj.jdbc.Driver
jdbc.mysql.url=jdbc:mysql:///nssp713db
jdbc.mysql.username=root
jdbc.mysql.password=root

#For Oracle
jdbc.oracle.driverclass=oracle.jdbc.driver.OracleDriver
jdbc.oracle.url=jdbc:oracle:thin:@localhost:1521:xe
jdbc.oracle.username=system
jdbc.oracle.password=manager
```

PersisteneConfig.java

```
package com.nt.config;

import javax.sql.DataSource;

import org.apache.commons.dbcp2.BasicDataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.context.annotation.PropertySource;
import org.springframework.core.env.Environment;
import org.springframework.jdbc.core.JdbcTemplate;
```



```

import com.mchange.v2.c3p0.ComboPooledDataSource;
import com.zaxxer.hikari.HikariDataSource;

@Configuration
@ComponentScan(basePackages = "com.nt.dao")
@PropertySource("com/nt/commons/jdbc.properties")
public class PersistenceConfig {

    @Autowired
    private Environment env;

    @Bean
    @Profile("dev")
    public DataSource createC3P0DS() throws Exception {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setDriverClass(env.getProperty("jdbc.mysql.driverclass"));
        ds.setJdbcUrl(env.getProperty("jdbc.mysql.url"));
        ds.setUser(env.getProperty("jdbc.mysql.username"));
        ds.setPassword(env.getProperty("jdbc.mysql.password"));
        return ds;
    }

    @Bean
    @Profile("test")
    public DataSource createApacheDBCPDSMySQL() throws Exception {
        BasicDataSource bds = new BasicDataSource();

        bds.setDriverClassName(env.getProperty("jdbc.mysql.driverclass"));
        bds.setUrl(env.getProperty("jdbc.mysql.url"));
        bds.setUsername(env.getProperty("jdbc.mysql.username"));
        bds.setPassword(env.getProperty("jdbc.mysql.password"));
        return bds;
    }

    @Bean
    @Profile("uat")
    public DataSource createApacheDBCPDSOracle() throws Exception {
        BasicDataSource bds = new BasicDataSource();

        bds.setDriverClassName(env.getProperty("jdbc.oracle.driverclass"));
        bds.setUrl(env.getProperty("jdbc.oracle.url"));
        bds.setUsername(env.getProperty("jdbc.oracle.username"));
    }
}

```

```

        bds.setPassword(env.getProperty("jdbc.oracle.password"));
        return bds;
    }

    @Bean
    @Profile("prod")
    public DataSource createHikariCPDS() throws Exception {
        HikariDataSource hds = new HikariDataSource();

        hds.setDriverClassName(env.getProperty("jdbc.oracle.driverclass"));
        hds.setJdbcUrl(env.getProperty("jdbc.oracle.url"));
        hds.setUsername(env.getProperty("jdbc.oracle.username"));
        hds.setPassword(env.getProperty("jdbc.oracle.password"));
        return hds;
    }

    @Bean(autowire = Autowire.BY_TYPE)
    public JdbcTemplate createJT() throws Exception {
        return new JdbcTemplate();
    }
}

```

Spring profiles in XML + Annotation or XML configurations-based Spring App development:

We need to use "profile" attribute of <beans> tag.

Step 1: Keep Mini Project ready (XML + Annotations based).

Step 2: Add Apache dbcp2, C3P0 and HikariCP jar files.

Step 3: Takes multiple persistence-beans.xml files from for multiple profiles and import them in applicationContext.xml.

Step 4: Develop DAO class ready to work for profile having @Profile

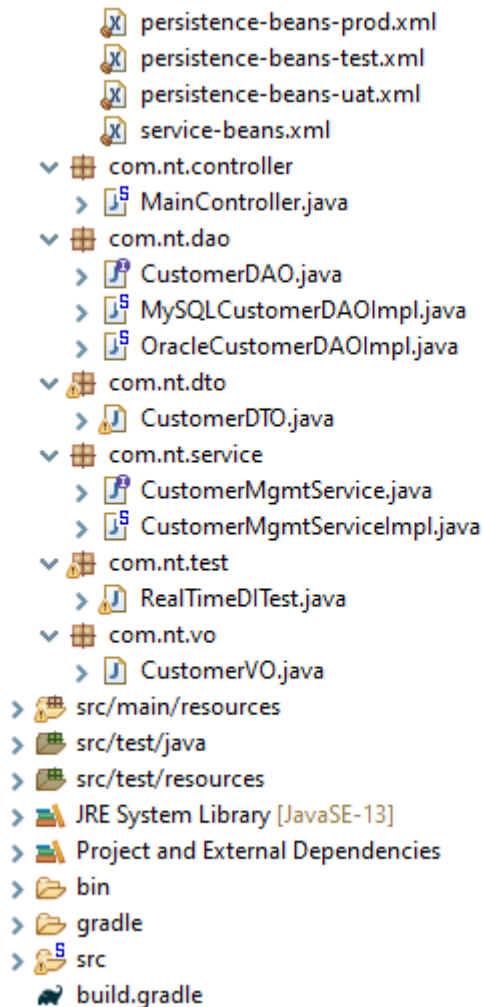
Step 5: Activate Profile from Client application.

Directory Structure of DAOAnnoProj07-MiniProject-Profiles:

```

v DAOAnnoProj07-MiniProject-XML-Annotation
  > Spring Elements
  v src/main/java
    v com.nt.bo
      > CustomerBO.java
    v com.nt.cfigs
      applicationContext.xml
      controller-beans.xml
      persistence-beans-dev.xml

```



- Develop the above directory structure using gradle setup and package, class, XML file and add the jar dependencies to build.gradle.
- Then add the following code with in their respective file.
- Rest of code copy from previous project.

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">

    <import resource="persistence-beans-dev.xml"/>
    <import resource="persistence-beans-test.xml"/>
```

```
<import resource="persistence-beans-uat.xml"/>
<import resource="persistence-beans-prod.xml"/>
<import resource="service-beans.xml"/>
<import resource="controller-beans.xml"/>
```

```
</beans>
```

controller-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
  <context:component-scan base-package="com.nt.controller"/>
</beans>
```

service-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-4.3.xsd">
  <context:component-scan base-package="com.nt.service"/>
</beans>
```

persistence-beans-dev.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans profile="dev"
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
    http://www.springframework.org/schema/context
```

```

http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <bean id="c3p0DS"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///nssp713db"/>
        <property name="user" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="c3p0DS"/>
    </bean>

    <context:component-scan base-package="com.nt.dao"/>

</beans>

```

persistence-beans-prod.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans profile="prod"
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <bean id="hikariCPDS" class="com.zaxxer.hikari.HikariDataSource">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver" />
        <property name="jdbcUrl"
value="jdbc:oracle:thin:@localhost:1521:xe" />
        <property name="username" value="system"/>
        <property name="password" value="manager"/>
    </bean>
    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="hikariCPDS"/>
    </bean>
    <context:component-scan base-package="com.nt.dao"/>

</beans>

```

persistence-beans-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans profile="test"
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <bean id="dpcpDS"
class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName"
value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///nssp713db"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>
    <bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dpcpDS"/>
    </bean>
    <context:component-scan base-package="com.nt.dao"/>
</beans>
```

persistence-beans-uat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans profile="uat"
xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <bean id="dpcpDS"
class="org.apache.commons.dbcp2.BasicDataSource">
        <property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver" />
        <property name="url"
```

```

value="jdbc:oracle:thin:@localhost:1521:xe" />
    <property name="username" value="system"/>
    <property name="password" value="manager"/>
</bean>

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">
    <constructor-arg ref="dpcpDS"/>
</bean>

<context:component-scan base-package="com.nt.dao" />

</beans>

```

persistence-beans-uat.xml

```

package com.nt.test;

import java.util.Scanner;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.env.ConfigurableEnvironment;

import com.nt.controller.MainController;
import com.nt.vo.CustomerVO;

public class RealTimeDITest {

    public static void main(String[] args) {
        Scanner sc = null;
        String name=null, address=null, Amount=null, time=null,
rate=null;
        CustomerVO vo = null;
        MainController controller = null;
        String result = null;
        //Read inputs from end-user using scanner
        sc = new Scanner(System.in);
        System.out.println("Enter the following Details for registration :
");

        System.out.print("Enter Customer Name : ");
        name = sc.next();
        System.out.print("Enter Customer Address : ");
        address = sc.next();
        System.out.print("Enter Customer Principle Amount : ");

```



```

        Amount = sc.next();
        System.out.print("Enter Customer Time : ");
        time = sc.next();
        System.out.print("Enter Customer Rate of Interest: ");
        rate = sc.next();
        //Store into VO class object
        vo = new CustomerVO();
        vo.setCname(name);
        vo.setCadd(address);
        vo.setPAmt(Amount);
        vo.setTime(time);
        vo.setRate(rate);
        //Create BeanFactory [IoC] container
        ClassPathXmlApplicationContext ctx = new
ClassPathXmlApplicationContext();
        //get Environment object
        ConfigurableEnvironment env = (ConfigurableEnvironment)
ctx.getEnvironment();
        //set Active profile
        env.setActiveProfiles("dev");
        //set sprig bean configuration file
        ctx.setConfigLocation("com/nt/cfgs/applicationContext.xml");
        ctx.refresh();
        //get controller class object
        controller = ctx.getBean("controller", MainController.class);
        //invoke methods
        try {
            result = controller.processCustomer(vo);
            System.out.println(result);
        }
        catch (Exception e) {
            System.out.println("Internal problem :
"+e.getMessage());
            e.printStackTrace();
        }

    } //main

} //class

```


Spring Boot Flow for Standalone application

- Flow starts from main () method of the SpringBootApplication. From that main method run () method of Spring Application is called.
- Locates and loads application.properties/ yml file.
- Their application first check for active Profile.
- So here based on the active profile set in application.properties/ yml the Profile related object is created.
- Activate profile related application-<xxx>.properties/.yml file is located and loaded.
- Then based on the application type ApplicationContext object is created. If it is standalone → AnnotationConfigApplicationContext (yes)
If it is web app → AnnotationConfigServletWebServerApplicationContext
- Autoconfiguration based spring bean objects will be created based on the jar files that are added by collecting inputs from the active profile specific properties or yml file.
- After AutoConfiguration is done, it creates spring bean object based on configuration class @Bean methods, stereotype annotations having scope singleton by scanning Configuration classes with the support of @Import, @ComponentScan annotations. In this Process the dependency Injections on beans will also be completed.
- All the above beans will be placed Internal Cache/HashMap of IOC container.
- All the created objects are registered with JMX as MBeans. Java Management Extensions (JMX) is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (such as printers) and service-oriented networks. Those resources are represented by objects called MBeans (for Managed Bean).
- Then the remaining logics of main method is getting executed like ctx.getBean(-) and calling business methods.
- Closing ApplicationContext container by calling ctx.close();.
- All singleton scope bean objects will be destroyed.
- All beans/ objects/ devices and etc. (MBeans) that registered with JMX will be unregistered.

----- The END -----