



INDEX

Spring Boot Core -----

1. Spring Boot	<u>04</u>
a. Spring Boot primary goals	<u>10</u>
b. Pros/ Cons of Spring Boot	<u>11</u>
2. Different approaches of developing Spring Boot Projects	<u>11</u>
a. Spring Boot Components	<u>13</u>
b. How to use Spring Boot	<u>15</u>
3. Structure and Background for first Spring Boot App development	<u>17</u>
a. Thumb rule to develop Spring Boot Application	<u>18</u>
b. Procedure to develop Spring Boot First application showing Dependency Injection	<u>18</u>
c. Flow of Execution of First application	<u>23</u>
4. Difference Between Spring and Spring Boot	<u>27</u>
a. Different types of Application	<u>31</u>
b. Usage of Spring Boot Modules	<u>33</u>
5. Basic Annotations in Spring Boot Programming	<u>35</u>
6. Developing Spring Boot application using Java Config Annotations	<u>44</u>
7. Spring Boot Layered Application & AutoConfiguration	<u>46</u>
a. JDBC Connection Pool	<u>48</u>
b. Java Bean	<u>51</u>
c. Storyboard for Spring Boot application	<u>54</u>
d. Procedure to develop Mini Project/ Layered application	<u>57</u>
8. Spring Boot Starter Parent	<u>76</u>
a. @Value Annotation	<u>78</u>
b. Injecting values to different types of Spring bean properties from properties file	<u>88</u>
9. YML/ YAML	<u>92</u>
10. Profiles in Spring Boot	<u>96</u>
a. How to activate specific Profile	<u>99</u>
b. Child Profiles or Profiles Include	<u>107</u>
11. Runners in Spring Boot	<u>110</u>

Spring Boot Core

Spring Boot

- ✚ It is not another module in Spring.
- ✚ It is an approach of developing Spring applications, (Internally it is spring framework only).

Spring Boot = Spring - XML configuration + Auto configuration.

- ✚ This Auto configuration feature generates the common thing required in Spring Boot project dynamically and those are
 - Classes as Spring beans
 - Jars
 - DB connectivity
 - Servers
 - InMemory DB softwares
 - Plugins
 - Configurations and etc.
- JEE technologies-based project development: 1000 lines of code (Everything should be taken care programmer manually)
- Spring Framework based project development: 200 lines code (Because of Abstraction it provides on JEE technologies)
- Spring Boot based project development: 80 lines code (Because Auto configuration)

Plain JDBC Application (Technology based application):

- | | |
|---|-----------------------------|
| ▪ Load JDBC driver Class (optional) | Common logics |
| ▪ Establish the Connection with DB software | |
| ▪ Creates JDBC Statement object | |
| ▪ Send and execute SQL Query in DB software | Application specific logics |
| ▪ Gather Results and process results | |
| ▪ Exception handling | Common logics |
| ▪ Close JDBC objects | |
- While working with JAVA/ JEE technologies the programmers should take care of both common logics and application specific logics.
 - Writing same common logics in multiple apps is called as Boilerplate code.
 - The code that repeats in multiple parts of same project either with no

changes or with minor changes is called boilerplate code problem.

Spring App (using Spring Core):

AppConfig.java

```
@Configuration
@ComponentScan(basePackages="com.nt.dao")
public class AppConfig {
    @Bean(name="ds")
    public DriverManagerDataSource createDataSource () {
        DriverManagerDataSource ds=new
                                   DriverManagerDataSource();

        ds.setDriverClassName("....."),
        ds.setUrl(".....");
        ds.setUserName("....."),
        ds.setPassword ("....."),
        return ds;
    }
}
```

StudentDAOImpl.java

```
@Repository
public class StudentDAOImpl implements StudentDAO {
    @Autowired
    private DataSource ds;

    public long getStudentsCount () {
        .....
        ..... //write jdbc code to send and execute SQL query
        .....
        return count;
    }
}
```

Spring Boot:

1. Add one Spring Boot starter to CLASSPATH whose name is spring-boot-starter-jdbc (Ready Made code)

This starter has the following Spring beans through auto configurations

- | | |
|-------------------------------|-----------------------|
| a. DataSource | (Related Spring JDBC) |
| b. JdbcTemplate | |
| c. NamedParameterJdbcTemplate | |

d. and etc.

2. [application.properties](#) (fixed)

```
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=.....
spring.datasource.username=.....
spring.datasource.password=.....
```

(Fixed keys given by spring boot)

3. DAO class

[StudentDAOImpl.java](#)

@Repository

```
public class StudentDAOImpl implements StudentDAO {
```

```
    @Autowired
```

```
    private DataSource ds;
```

```
    public long getStudentsCount () {
```

```
        .....
```

```
        ... //write JDBC code to send and execute SQL query
```

```
        .....
```

```
        return count;
```

```
    }
```

```
}
```

With Spring Boot, we get

- Abstraction on Java/ technologies
- AutoConfiguration
- Simplification in application development

[Productivity will be improved]

✚ Spring boot starters are advanced dependencies (jar files) which provides libraries to require for the Spring Boot project and helps Spring Boot to perform AutoConfiguration activities.

✚ These starters should be added CLASSPATH either through pom.xml (maven) or through build.gradle (gradle).

Starter pattern in Spring Boot:

spring-boot-starter- *

E.g.,

- spring-boot-starter-jdbc

- spring-boot-starter-web
 - spring-boot-starter-devtools
 - spring-boot-starter-data-jpa
 - spring-boot-starter-data-test
 - and etc. (1000 +) starters are there.
- In Java we get APIs as libraries in form of jar files. These jar files will be having dependent jar files.
 - If classes of one jar file is using the classes of another jar file, then another file is called dependent jar file to first/ one jar file.
 - spring-context-support-<ver>.jar is having multiple dependent jar files like spring-core-<ver>.jar, commons-logging.jar, file.
 - If Java App uses any API/ library then that API related main and dependent jar files must be added to CLASSPATH.

first.jar	second.jar
(A)	(B)

App1.java

```
class App1 {
    public static void main (String args []) {
        A a = new A ();
        a.m1();
    }
}
```

- m1() definition of class A is calling m2() of class B (So we can say first.jar file is main jar file and second.jar file is dependent jar file).
- We should add both first.jar and second.jar to the CLASSPATH.

- ✚ Identifying dependent jar files for any main jar file is very complex in manual process.
- ✚ To simplify this job we got maven, gradle and etc. build tools. While developing java projects using these tools if we add main jar file dependency, then the tool download both main and dependent jar files to CLASSPATH.
- ✚ Spring boot starters are not just main and dependent jar file, they also include relevant jar files related to certain task completion through autoconfiguration. These starters can also be added as dependencies to Spring projects using maven/ gradle.

For example, spring-boot-starter-jdbc gives

- Spring JDBC jars (Main jar file)
- spring-tx-<ver>.jar (Dependent jar file)
- HikariCP (Default DataSource in Spring Boot App
HikariCP DataSource)
- Relevant logging jars
and etc.

Two types of JDBC connection objects:

1. Direct JDBC connection

created by the programmer manually

```
Class.forName(".....");
```

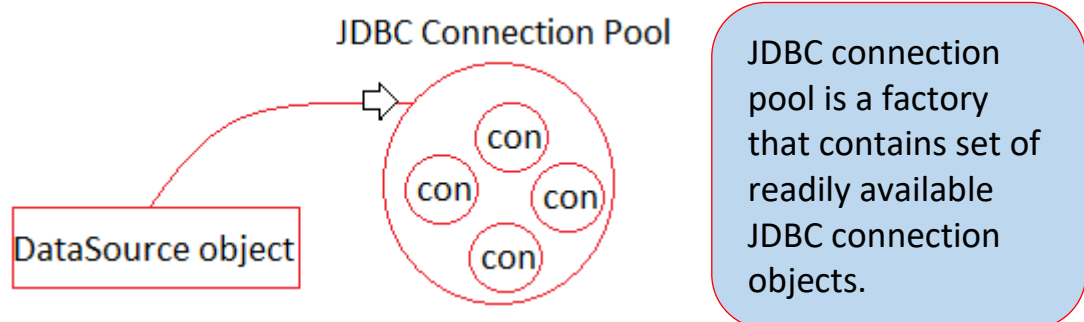
```
Connection con=DriverManager.getConnection(-,-,-);
```

Direct JDBC connection

2. Pooled JDBC Connection

Collected from JDBC con pool through DataSource object

```
Connection con=ds.getConnection();
```



- DataSource object is entry point to access each JDBC connection object from JDBC connection pool.

Two types of JDBC connection pools:

a. Standalone JDBC connection pools

Useful in Standalone Java Apps

E.g.,

- Apache DBCP
- C3pO
- Proxool
- HikariCP (best)
and etc.

- b. Server Managed JDBC connection pool
Useful in applications that are deployable in Web Server/ Application Server. Apps are web applications/ RESTful webservice Apps (Distributed apps).

E.g.,

- WebLogic managed JDBC connection pool
 - Tomcat managed JDBC connection pool
 - Wildfly managed JDBC connection pool
- and etc.

Developing Java web application using Servlet, JSP technologies (0%, 100%):

- Should add jars manually to the classpath (servlet-api.jar, JDBC driver jars and etc.).
 - Should understand and implement MVC2 architecture manually. [MVC2 (Model-View-Controller) is an architecture to develop Java web application as layered application].
 - No Ready Made FrontController Servlet we should develop manually.
 - Should arrange Web server/ Application server manually.
 - No productivity (because all logics all layers should be developed by programmers)
- and etc.

Developing Java web application using Spring MVC framework (50%, 50%):

- Should jar manually to the classpath.
 - Gives Built-in servlet component (DispatcherServlet) as FrontController servlet.
 - Gives MVC2 architecture Flow as built-in flow based on DispatcherServlet.
 - Should arrange webserver/ App server manually.
 - Gives bit of good productivity compare to Servlet, JSP technologies.
- and etc.

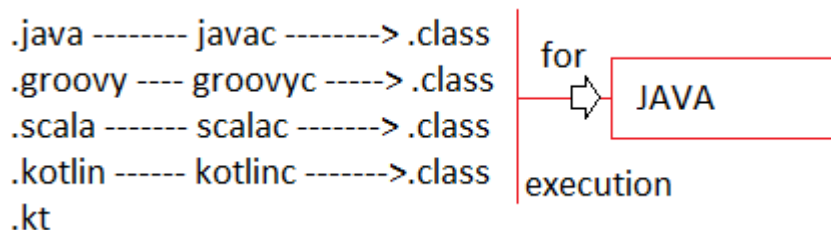
Developing Java web application using Spring Boot MVC framework (80%, 20%):

- By adding spring-boot-starter-web starter we can get both main, dependent and relevant jar files to CLASSPATH.
- Gives built-in Front Controller Servlet (DispatcherServlet).
- Gives MVC2 architecture Flow as built-in flow based on DispatcherServlet.

- Automatic Configuration of DispatcherServlet and components (pre-defined Servlet).
- Embedded servers (Tomcat/ Jetty) (need not arrange servers manually).
- Gives Great productivity.

Note:

- ✓ Java, Groovy, Scala, Kotlin and etc. are called JVM based languages i.e., they have own syntax of code development but all of them give .class files after compilation, and these .class files can be executed using same JVM.



- ✓ We can develop Spring Boot apps in the following JVM based languages
 - Java (best)
 - Groovy
 - Kotlin

Spring Boot primary goals

- Provide a radically faster and widely accessible getting started experience for all Spring development.
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (e.g., embedded servers, security, metrics, health checks, externalized configuration).
- Absolutely no code generation and no requirement for XML configuration, to avoid XML Configuration completely.
- To avoid defining more Annotation Configuration (It combined some existing Spring Framework Annotations to a simple and single Annotation).
- To avoid writing lots of import statements
- To provide some defaults to quick start new projects within no time.

Pros/ Cons of Spring Boot

Pros of Spring Boot:

- It is very easy to develop Spring Based applications with Java or Groovy.
- It reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
- It follows "Opinionated Defaults Configuration" Approach to reduce Developer effort.
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
- It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
- It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle.
- It provides lots of plugins to work with embedded and in-memory Databases very easily.

Cons of Spring Boot:

- The biggest limitation of Spring boot is it is very tough and time-consuming process to migrate Spring framework projects into spring boot projects i.e., is not for brown field projects (project enhancement, migration) and it is purely for green field projects (new projects).

Different approaches of developing Spring Boot Projects

- ✚ Using Spring Boot CLI Tool
 - ✚ Using Spring STS IDE
 - ✚ Using Spring_INITIALIZER [\[Website\]](#)
 - ✚ By configuring STS plugin in Eclipse IDE (Best)
- ✓ Internally maven/ gradle will be used. So, internet connection is required while developing projects.

Q. Why Spring Boot?

Ans. Even though Spring is a great framework it has few pitfalls. Spring Boot was built not just to address them. It also provides direction to the future of

software development. Spring's XML based configuration is a nightmare in the world of annotation. There was no clear leader in the java framework world to support Microservices. You really don't want different teams building Microservices to adapt different set of Java Libraries and look very different to each other.

Q. What Spring Boot brings to the table?

Ans.

Convention over configuration:

- Spring Boot has taken away all the XML based configurations and provided Annotations for using the Spring Framework. You can start your application with a very minimum annotation in no time.
- This would be very helpful to the developers, the team productivity would greatly be impacted positively.

Standardization for Microservices:

- One of the main objectives of Spring Boot is to provide a unified ecosystem of libraries & standards to all the developers (teams) utilizing Microservices methodologies.
- Any project adapting Microservices would have multiple teams and we certainly don't want each of team to build the soft-wares in very different way. The teams will be also benefit from the all the annotation and tooling Spring Boot brings, however it also provides-
 1. A common platform and library support.
 2. Reduced setup, configuration time in development environment.
 3. Cloud Support

Integrated Server for Development:

- Spring Boot attaches a Tomcat/ Jetty server with the compiled Jar using Maven/ Gradle. This helps the developers to expedite the development process by using the integrated server. In 2-3 mins you could build & test a RESTful web service from scratch.

Cloud Support:

- Spring Boot provides cloud support for configuration. tools and clients. It's also compatible with Cloud Native and works seamlessly with Cloud Foundry, Pivotal etc.

Adapt & Support for 3rd Party Library:

- Spring Boot has taken a significant step and widen support for 3rd Party

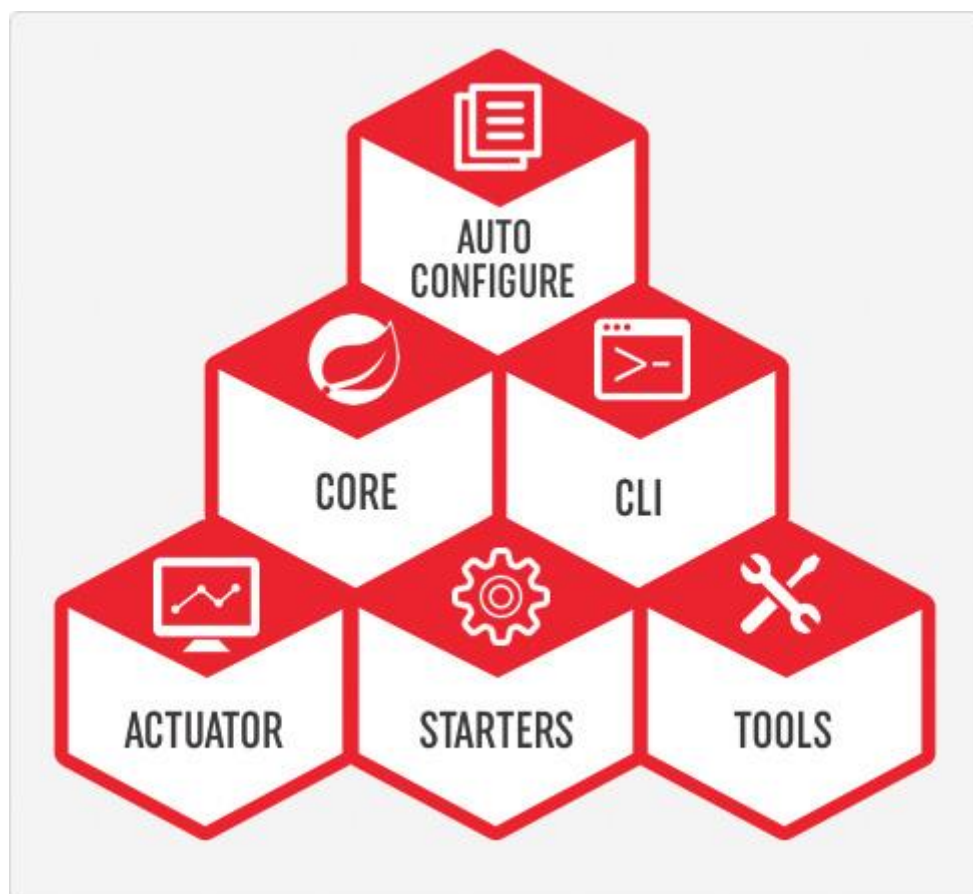
Open-Source Library like Netflix OSS, No-SQL DB, Distributed Cache etc. You will find a full list in the Spring Boot home page. However, the seamless integration using Annotation is very powerful.

Q. What is Spring Boot?

Ans. In one sentence, Spring Boot is equal to (Spring Framework – XML Configuration) + Integrated Server/ Auto configuration.



Spring Boot Components



Spring Boot Auto Configure:

- Module to auto configure a wide range of Spring projects. It will detect availability of certain frameworks (Spring Batch, Spring Data JPA, Hibernate, JDBC). When detected it will try to auto configure that framework with some sensible defaults, which in general can be overridden by configuration in an application.properties/ yml file.

Spring Boot Core:

- The base for other modules, but it also provides some functionality that can be used on its own, e.g., using command line arguments and YAML files as Spring Environment property sources and automatically binding environment properties to Spring bean properties (with validation).

Spring Boot CLI:

- Command line interface, based on ruby, to start/stop spring boot created applications.

Spring Boot Actuator:

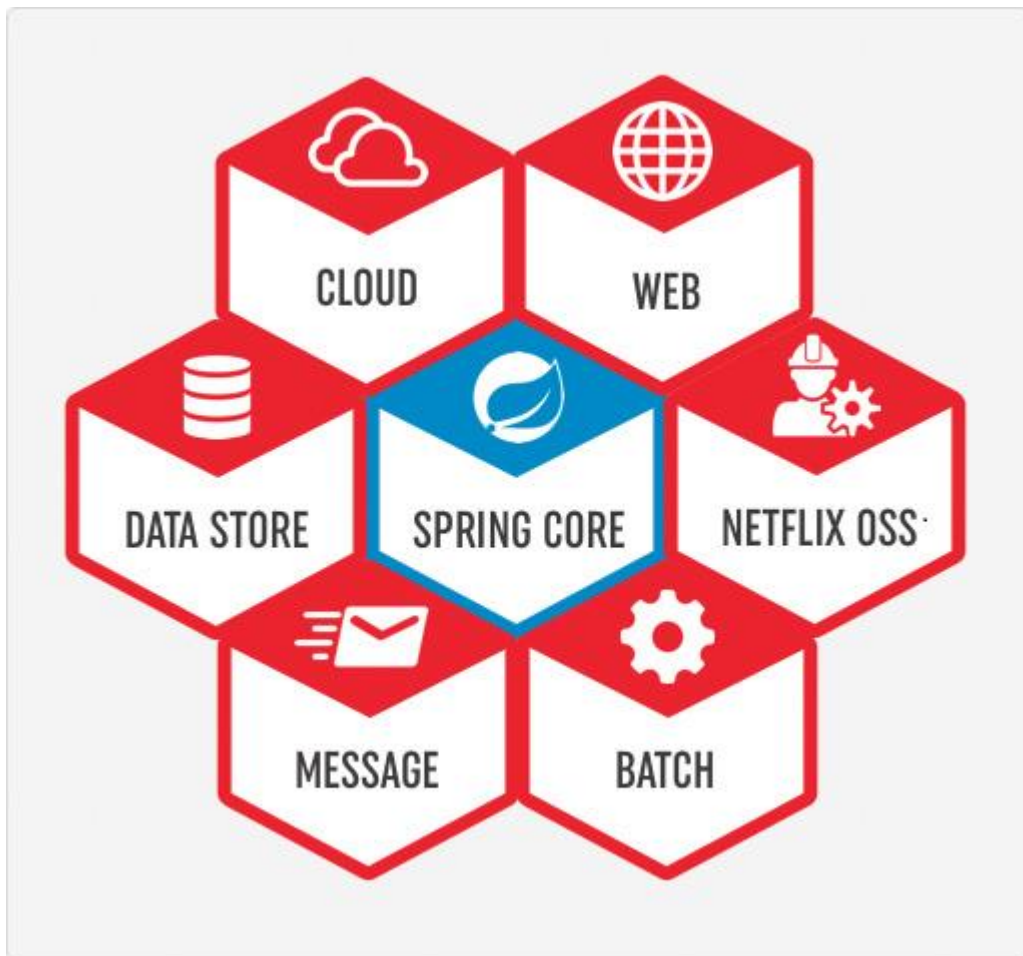
- This project, when added, will enable certain enterprise features (Security, Metrics, Default Error pages) to your application. As the auto configure module it uses auto detection to detect certain frameworks/features of your application.
- For an example, you can see all the REST Services defined in a web application using Actuator.

Spring Boot Tools:

- The Maven and Gradle build tool as well as the custom Spring Boot Loader (used in the single executable jar/ war) is included in this project.

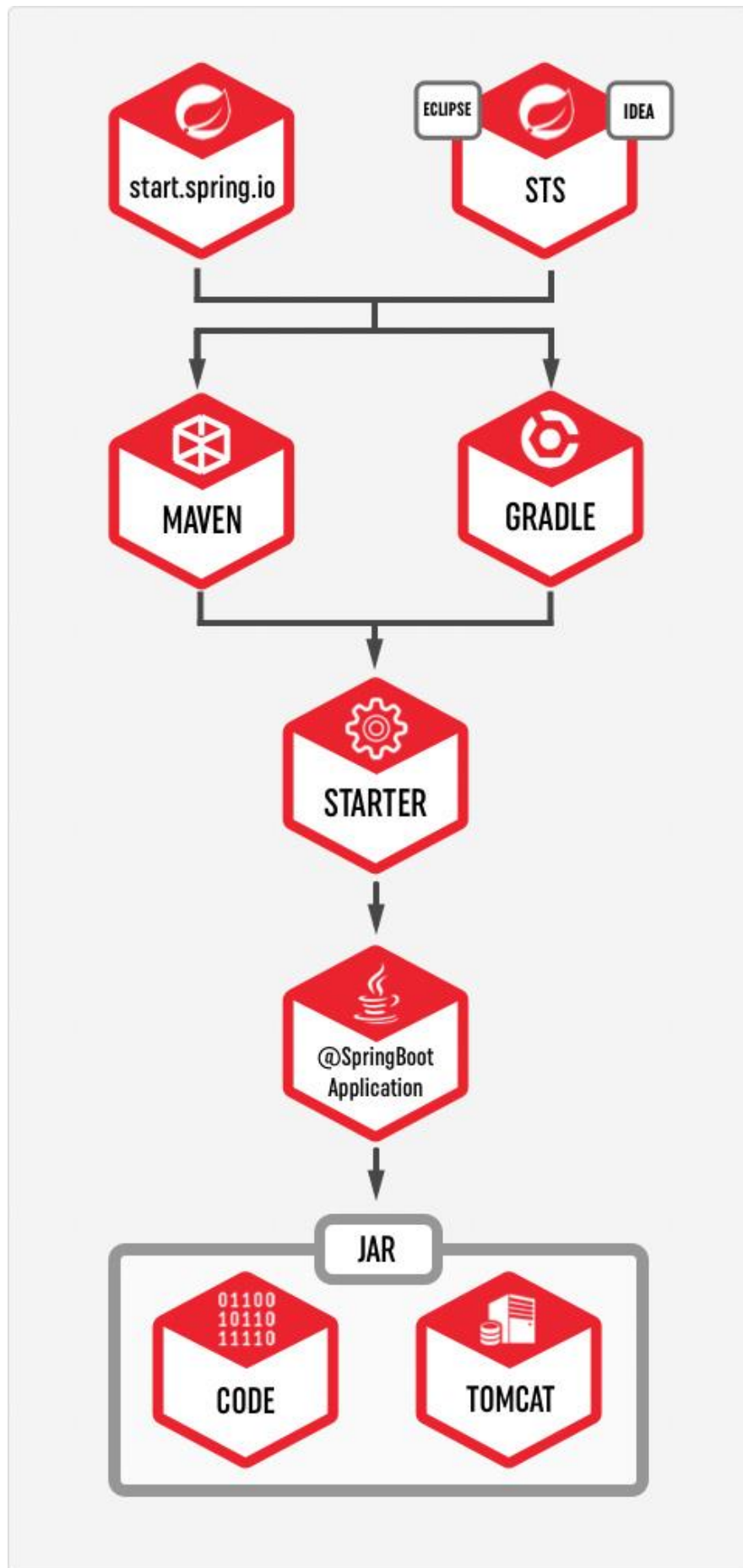
Spring Boot Starters:

- Different quick start projects to include as a dependency in your maven or gradle build file. It will have the needed dependencies for that type of application.
- Currently there are many starter projects (We will learn about few of them in the next section) and many more are expected to be added.
- Spring Boot incorporates many starters' packages (in Maven & Gradle) which you can include in order to add appropriate support in your project.
- At a high-level there are so far 6 types of starters packages available. You can find all of them in Spring Boot official Website.



How to use Spring Boot

- ✚ In another post we will see how to use and run Spring Boot, however here are the steps you need to follow.
- ✚ You can use spring initialize to create the initial setup. You can visit either start.spring.io or use STS (Spring Tool Suite) support available in IDEA or Eclipse to choose all the Spring Boot Starters.
- ✚ You need to also choose whether to use Maven or Gradle as the build tool.
- ✚ If you are using start.spring.io, you need to then download the zip and configure your workspace. Otherwise using your preferred IDE will automatically create the required file in the workspace.
- ✚ Add your code as required
- ✚ You can either use clean package or use IDEA or Eclipse to build and create the jar file.
- ✚ By default, the JAR would include integrated Tomcat server, so just by executing the JAR you should be able to use your program.



Structure and Background for first Spring Boot application development

```
(Target class)                                (Dependent class)
WishMessageGenerator ----- LocalDateTime date;
    |--> LocalDateTime
        (HAS-A property injection)
    |--> public String generateWishMessage (String user) {
        ..... //uses the Injected LocalDateTime object to know the
        ..... current hour of the day and to generate wish message
    }
```

Stereotype annotations:

@Component, @Service, @Controller, @Repository and etc.
[To configure Java class as Spring bean i.e., internally creates java class object and makes it as Spring bean]

To Perform Field level injections:

```
use @Autowired at field level
@Component("wmg")
public class WishMessageGenerator {
    @Autowired
    private LocalDateTime date;
    public String generateWishMessage(String user) {
        ..... //uses the Injected LocalDateTime object
        ..... to know the current hour of the day
        ..... and to generate wish message
    }
}
```

To configure pre-defined Java class as Spring Bean:

Use @Bean method definition in @Configuration class or in Main class.
(Alternate to Spring bean configuration file (XML))

@SpringBootApplication

```
--> @Configuration (To make java class as configuration class)
--> @ComponentScan (To recognize current package and its sub
                    packages maintained stereo type annotations-
                    based java classes as spring beans).
--> @EnableAutoConfiguration (To enable AutoConfiguration)
```



Spring Boot gives single useful annotations by combining related multiple annotations

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiguration

Note: In Spring Boot application, we develop Configuration class using @SpringBootApplication Annotation which internally uses @Configuration and other annotations.

```
@SpringBootApplication
public class AppConfig {
```

```
    @Bean(name="ldt")
    public LocalDateTime createLDT () {
        return LocalDateTime.now ();
    }
    .....
}
```

This method returned LocalDateTime object will become Spring bean having bean id "ldt"

Thumb rule to develop Spring Boot Application

- Configure user-defined classes as Spring beans using stereo type annotations.
- Configure pre-defined classes as Spring beans using @Bean methods of @SpringBootApplication/ @Configuration class only when they are not coming through autoconfiguration.
- Provide inputs to AutoConfiguration process using application.properties/ yml file.
- Get IoC container from SpringApplication.run(-) and complete your client application jobs.

Spring Boot = Spring f/w - XML files + AutoConfiguration + Embedded Servers + Embedded DB +

Procedure to develop Spring Boot First application showing Dependency Injection

Step 1: Keep the following software setup ready

Eclipse (2021-12) with STS Plugin (4.14.0), JDK 1.8+

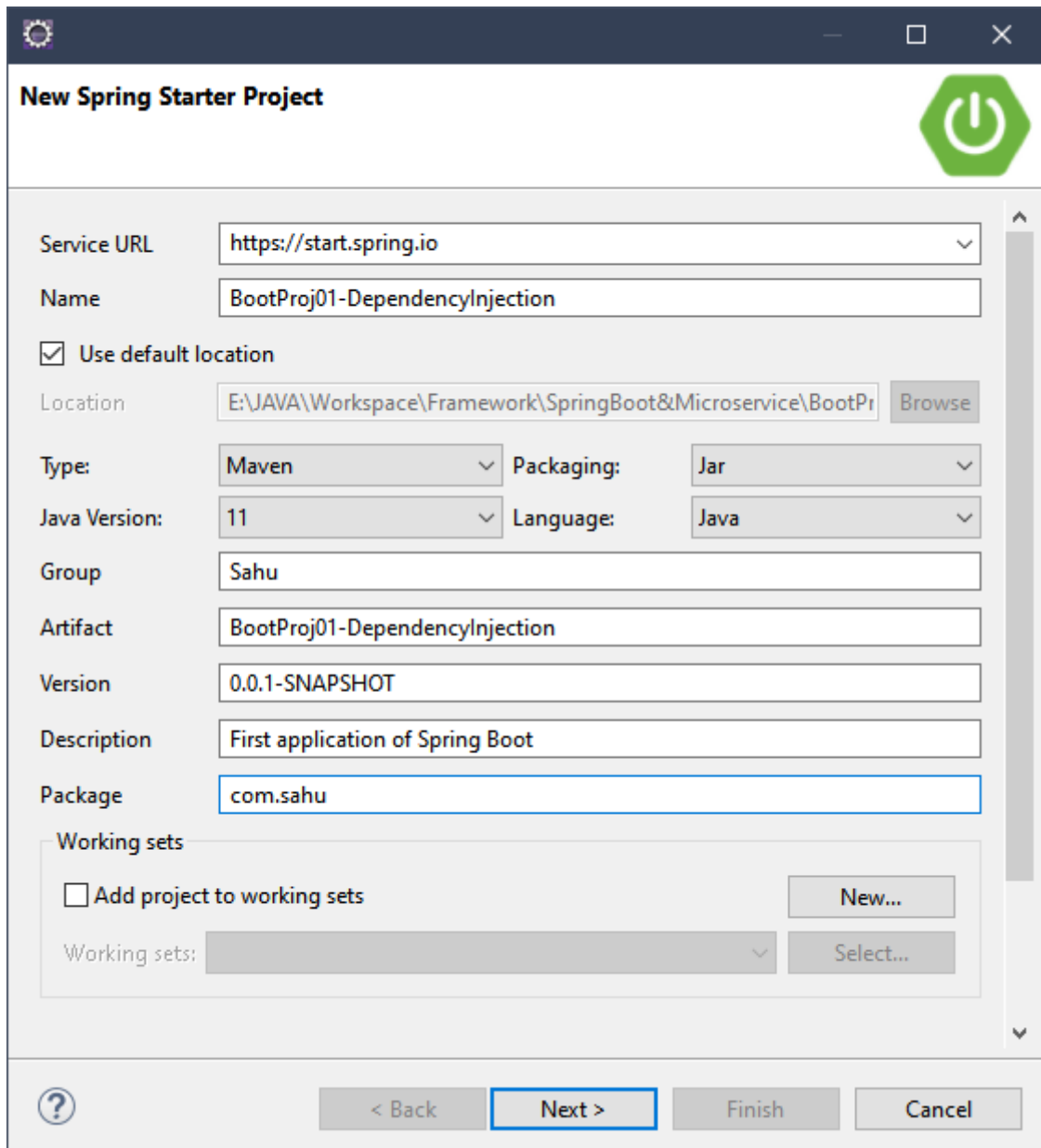
Note: If you are using Eclipse 2021-09 or higher don't use Spring Tool 3.9.14 use Spring Tool 4 (4.14.0) because Spring Tool 3.9.14 support they removed.

- To install STS plugin: Help -> Eclipse market place -> search for STS -> select Spring Tools 4 (aka Spring Tool Suit 4) 4.1.4.0 RELEASE -> install -> select all -> next -> accept terms and conditions -> restart IDE.
- Plugin is a patch software that provides additional functionalities to existing s/w. STS plugin makes Eclipse to develop Spring, Spring Boot apps very easily, more over bring STS IDE features to Eclipse IDE.

Step 2: Launch Eclipse IDE by choosing Workspace folder (The folder where projects will be saved)

Step 3: Create spring starter Project in eclipse IDE.

File -> menu -> new -> others -> search spring starter project



New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

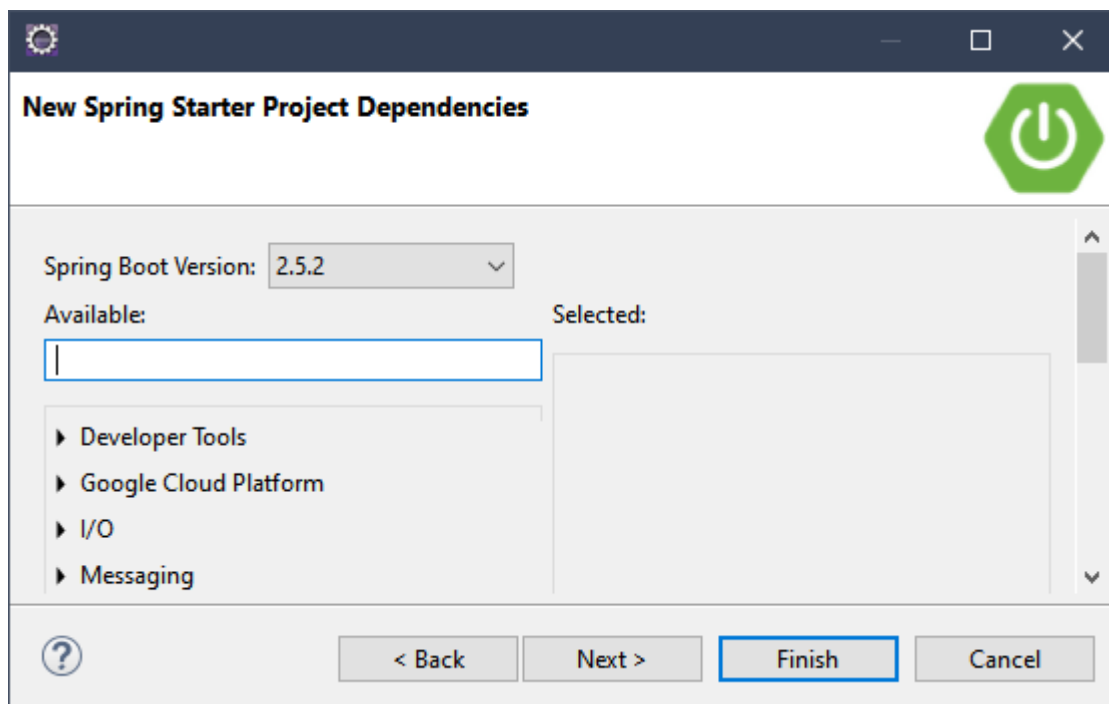
Package:

Working sets

☐ Add project to working sets

Working sets:

Service URL : Fixed URL where Spring Boot project created.
 Name : Name of the project
 Packaging : Jar (For standalone application)
 Language : Java (Language to develop Spring boot app)
 Group : Organization name
 Artifact : Project name
 Version : Project version
 Package : Root package name where client cum configuration class will be created having @SpringBootApplication annotation.



click on Next -> (No starter) -> Next/ Finish -> Finish (If click next).

Note: In Maven/ Gradle tool project/ jar file is identified with 3 details

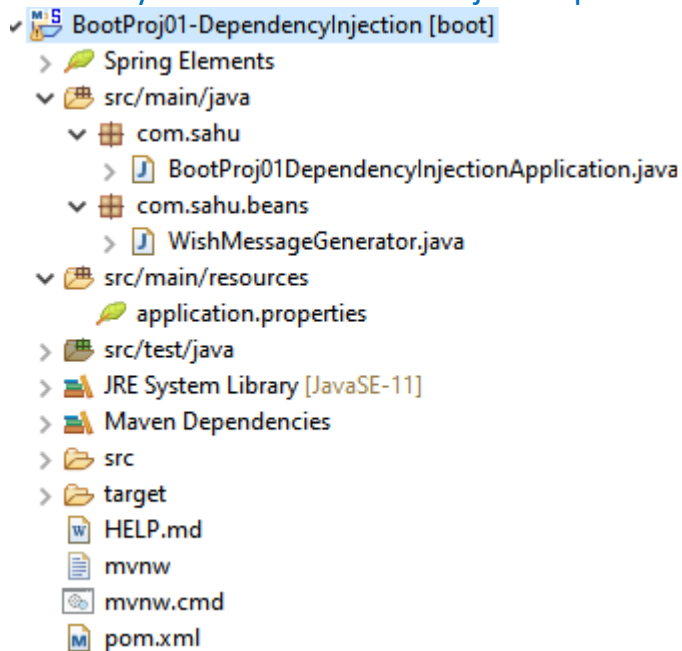
- Group Id (company name of project/ jar file)
- Artifact Id (project/ jar file name)
- Version (project/ jar file version)

Q. What is difference @Component and @Bean?

Ans.

- @Component can make only user-defined classes spring beans and it is class level annotation.
- @Bean can make any user-defined/ pre-defined class object given by underlying method as spring bean and it can be used at method level only in @Configuration class or in @SpringBootApplication class.

Directory Structure of BootProj01-DependencyInjection:



- Develop the above directory structure using Spring Starter Project option and no starter project required and create the package and classes also.
- Then use the following code with in their respective file.

WishMessageGenerator.java

```
package com.sahu.beans;

import java.time.LocalDateTime;

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

@Component("wmg")
public class WishMessageGenerator {

    @Autowired
    private LocalDateTime localDateTime;

    public WishMessageGenerator(LocalDateTime localDateTime) {

        System.out.println("WishMessageGenerator.WishMessageGenerator(
)");
    }
}
```

```

public String generateWishMessage(String user) {

    System.out.println("WishMessageGenerator.generateWishMessage()
");
        //convert in hour
        int hour = LocalDateTime.getHour();
        if(hour<12)
            return "Good Morning "+user;
        else if(hour<16)
            return "Good Afternoon "+user;
        else if(hour<20)
            return "Good Evening "+user;
        else
            return "Good Night "+user;
    }

}

```

BootProj1DependencyInjectionApplication.java

```

package com.sahu;

import java.time.LocalDateTime;

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

import com.sahu.beans.WishMessageGenerator;

@SpringBootApplication
public class BootProj01DependencyInjectionApplication {

    @Bean("ITD")
    public LocalDateTime createLocalDateTime() {

        System.out.println("BootProj01DependencyInjectionApplication.creat
eLocalDateTime()");
        return LocalDateTime.now();
    }

}

```

```

    public static void main(String[] args) {
        //get IoC container
        ApplicationContext ctx =
SpringApplication.run(BootProj01DependencyInjectionApplication.class,
args);

        //get target spring bean class from IoC container
        //WishMessageGenerator wGenerator = ctx.getBean("wmg",
WishMessageGenerator.class);
        WishMessageGenerator wGenerator =
ctx.getBean(WishMessageGenerator.class);
        //invoke method
        System.out.println(wGenerator.generateWishMessage("Sahu"));
        //close container
        ((ConfigurableApplicationContext) ctx).close();
    }
}

```

Note: SpringApplication.run (-, -) internally uses AnnotationConfigApplicationContext class to create and return IoC container by taking given java class as @Configuration class (In fact it takes current class nothing but client application cum configuration class as the configuration class).

Flow of Execution of First application

(6) Search in sub package of root package

WishMessageGenerator.java

package com.sahu.beans;

import java.time.LocalDateTime;

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

import org.springframework.stereotype.Component;

@Component("wmg") (7) Loads the class and creates the object

public class WishMessageGenerator {

@Autowired (8) Detects @Autowired on property in spring bean using

(10) **private** LocalDateTime **localDateTime**; reflection API and

BeanPostProcessor

public WishMessageGenerator(LocalDateTime localDateTime) {

```
System.out.println("WishMessageGenerator.WishMesageGenerator()");  
}
```

```
public String generateWishMessage(String user) { (17)
```

```
System.out.println("WishMessageGenerator.generateWishMessage()");  
    //convert in hour  
    int hour = LocalDateTime.getHour();  
    if(hour<12)  
        return "Good Morning "+user; (18)  
    else if(hour<16)  
        return "Good Afternoon "+user;  
    else if(hour<20)  
        return "Good Evening "+user;  
    else  
        return "Good Night "+user;  
    }  
}
```

BootProj1DependencyInjectionApplication.java

```
package com.sahu;
```

```
import java.time.LocalDateTime;
```

```
import org.springframework.boot.SpringApplication;
```

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

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

```
import org.springframework.context.annotation.Bean;
```

```
import com.sahu.beans.WishMessageGenerator;
```

```
@SpringBootApplication (4) Uses @ComponentScan annotation
```

```
public class BootProj01DependencyInjectionApplication {
```

```
    (5) Pre-instantiation of Singleton scope beans
```

```
    @Bean("ITD") (9)
```

```
    public LocalDateTime createLocalDateTime() {
```

```
        System.out.println("BootProj01DependencyInjectionApplication.createLocalDateTime()");
```

```
        return LocalDateTime.now();
```

```
    }
```



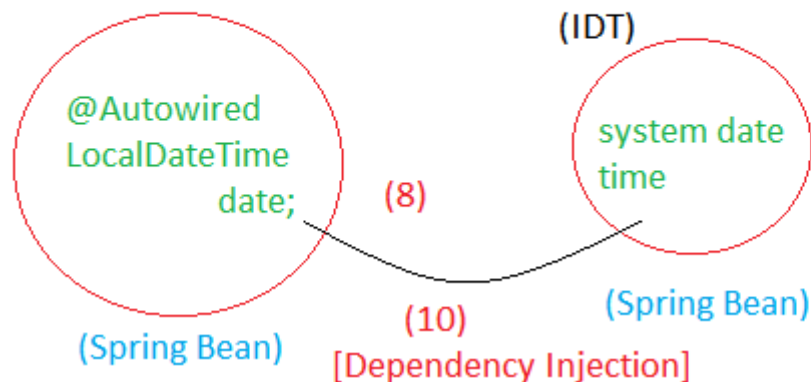
```

(1)
public static void main(String[] args) {
    //get IoC container (2) Performs many things including IoC
(12)    ApplicationContext ctx = container creation
    SpringApplication.run(BootProj01DependencyInjectionApplication.class, args);
    //get target spring bean class from IoC container
    //WishMessageGenerator wGenerator = ctx.getBean("wmg",
    WishMessageGenerator.class); (3) Loads and creates the current class object
(15)    WishMessageGenerator wGenerator = because it also
    ctx.getBean(WishMessageGenerator.class); (13) @Configuraion class
    //invoke method (16)
(19)    System.out.println(wGenerator.generateWishMessage("Sahu"));4
    //close container
    ((ConfigurableApplicationContext) ctx).close(); (20)
}
}

```

WishMessageGenerator
class object (wmg) (7)

LocalDateTime object
(IDT) (9)



(Map Collection)
Internal Cache of IoC container (11)

(14?)

IDT	LocalDateTime object reference
wmg	WishMessageGenerator object reference

keys values

X

(21) cache
will be
vanished

(23) IoC container closing

(24) All the objects that created in main (-) method will be vanished at the end of main (-) method.

Note:

- ✓ Container first creates singleton scope stereo type annotation-based user-defined spring bean class objects later it calls singleton scope @Bean methods of @Configuration class to make returned objects as Spring beans.
- ✓ If Prototype scope bean is dependent to singleton scope target bean, then IoC container creates Dependent spring bean class object along with singleton scope spring bean class object but it does not mean pre-instantiation is happened on prototype scope bean or it does not mean prototype scope is changed to singleton scope, just happened to support/ complete pre-instantiation and injections on singleton scope target spring bean.
- ✓ Every @Configuration class is one Spring bean having default scope singleton. So, its object reference will also be maintained in internal cache having default bean id.
- ✓ If we do not give bean id to stereotype annotations-based Spring beans then the class name taking first letter in lowercase becomes default bean id.

e.g.

```
@Component
public class WishMessageGenerator {
    .....
    .....
}
```

default bean is: wishMessageGenerator

- ✓ If we do not give bean id to @Bean methods-based spring beans then the method name itself becomes default spring bean id.

e.g.

```
@Bean
public LocalDateTime createLocalDateTime() {
    return LocalDateTime.now();
}
```

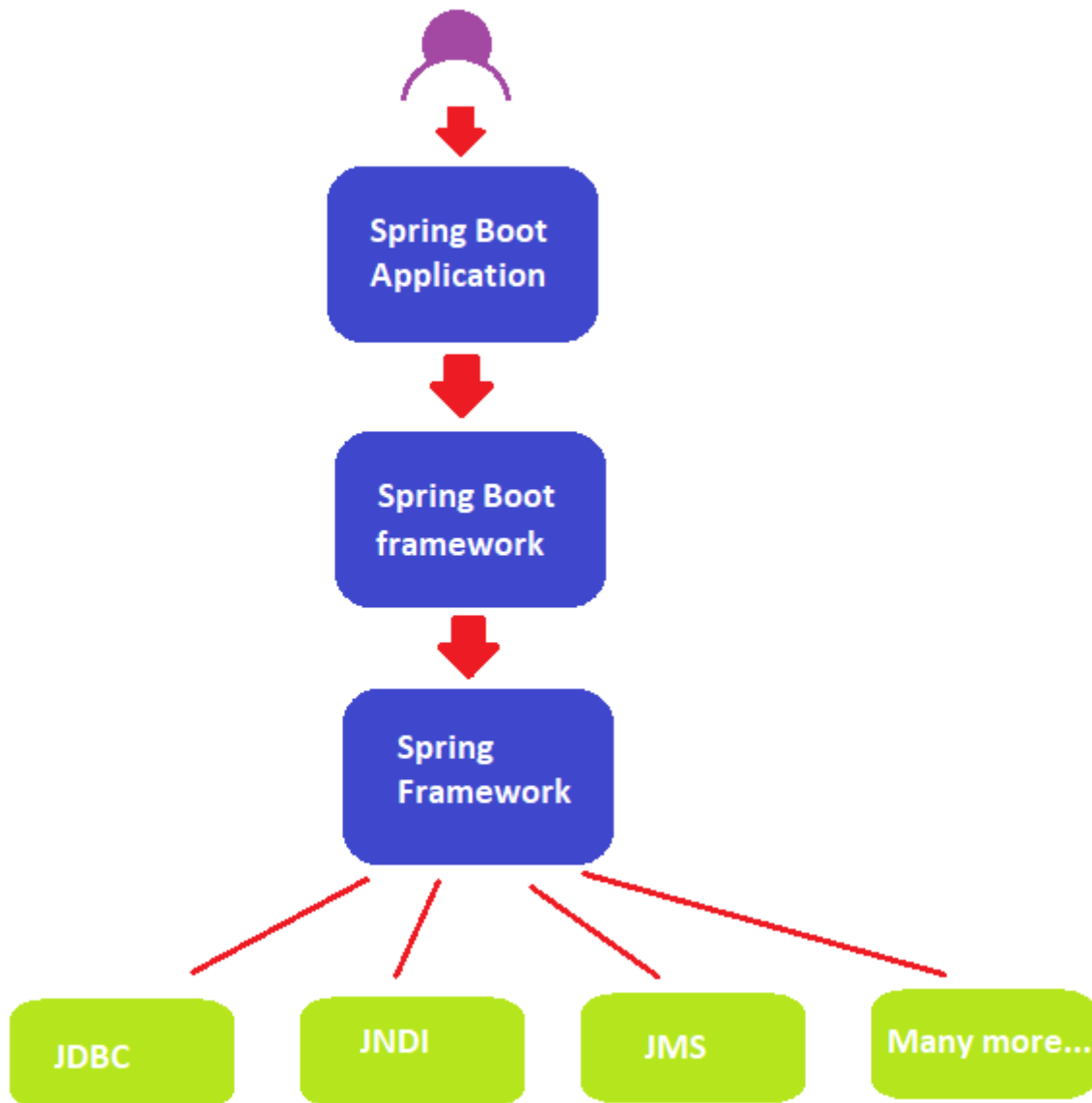
default bean id: createLocalDateTime (Method name)

Difference Between Spring and Spring Boot

Spring	Spring Boot
a. It is also called Java EE framework or Application framework.	a. It is just called Spring Boot Framework
b. Provides abstraction on Java, JEE technologies and simplifies their App development.	b. Provides abstraction on Spring framework and simplifies Spring apps development.
c. Avoids boilerplate code related to JAVA, JEE Technologies based application development (But still continues some boilerplate code).	c. Avoids Spring framework related boilerplate code directly.
d. The Main feature of Spring framework is Dependency Management (both Dependency Injection and Dependency lookup).	d. The main feature of Spring Boot framework is AutoConfiguration (giving commons things automatically).
e. Supports XML driven configuration to provide inputs/ configuration to IoC container.	e. Does not support XML driven configuration directly. Spring Boot avoids or minimize XML driven configuration. [If want to use certain features of Spring for which annotations are not there like method replacer, bean aliasing, inner beans, tiles, bean inheritance, collection merging and etc. we need to link Spring bean configuration file (XML file) to @SpringBootApplication class using @ImportResource).
f. Allows to develop Apps using 3 types of configurations, i. XML driven configuration ii. Annotation driven configuration iii. 100% Code driven/ Java Config approach configuration	f. Supports only one style of configuration that is through Annotations directly giving auto configuration inputs through application.properties/ yml. Note: Can add special configuration using XML file through @ImportResource.
g. Programmer creates IoC container explicitly (except in Spring MVC applications).	g. Programmer does not create IoC container rather he gets it by calling SpringApplication.run(-,-) according to application type.

h. Does not give Embedder Server, so run spring-based web applications we need to arrange server explicitly.	h. Gives Tomcat, Jetty and etc. server as Embedded servers in web application (only for testing).
i. Does not give any InMemory Databases.	i. Gives embedded InMemory Database like H2 (only for testing).
j. Supports good amount of Loose coupling using Spring bean configuration file (XML file) support (Changing one dependent with another dependent for target class without touching the java source code).	j. Loose coupling bit less because all configuration takes place through annotations which is java code.
k. We need to added dependencies (jar files) manually and directly (using maven/ gradle we can get dependent jar files when we add main jar files i.e., still we need relevant jar files explicitly).	k. Spring Boot gives starters (kind of dependencies) which provides main jar files, dependent jar files and relevant jar files. (Here also we use maven/ gradle to add starters).
l. Suitable for new projects development, to enhance existing projects and to migrate project from one version of Spring to another version.	l. Not suitable for Migration projects and also not suitable to convert Spring projects to Spring Boot projects. But very much suitable new projects development from scratch level.
m. Bit light weight compare to Spring Boot because no AutoConfiguration support.	m. Bit Heavy weight compare to Spring because of AutoConfiguration many unnecessary objects will be created. Like spring-boot-starter-jdbc gives HikariDataSource, JdbcTemplate, NamedParameterJdbcTemplate, DataSourceTransactionManager comes through autoconfiguration.
n. Spring framework is suitable for developing standalone apps and small scale and medium scale web applications.	n. Spring Boot is good to develop large scale web applications, Distributed apps and Microservice Architecture based application [Microservice architecture says develop different modules as different projects and integrate them

	using different third-party libraries].
o. No support for Microservices architecture-based application development.	o. Supports more exclusively.



- ✚ Webservices is given to develop Distributed Application. Webservices can implemented in two approaches
 - a. SOAP based webservices (Legacy - Almost outdated)
 - Jax-rpc, jax-ws are Technologies to develop SOAP based webservices.
 - Apache CFX, AXIS are frameworks to develop SOAP based webservices.

b. RESTful webservice (Popular - Hot Cake)

- Jax-RS (metro) is technology to develop Restful webservice.
- Jersey, Spring Rest are frameworks to develop Restful webservice.

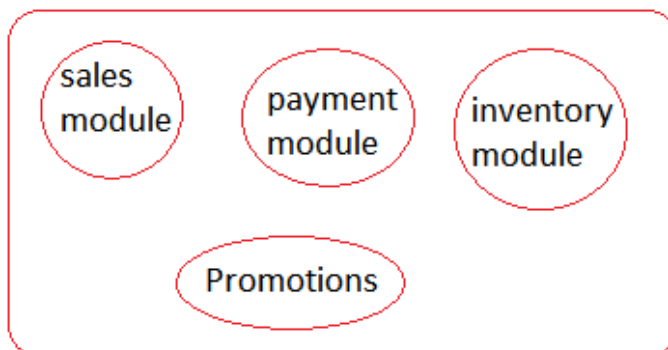
✚ Microservices are built on the top of Restful webservice to develop different services/ modules as different projects (each project is called one microservice) later these projects (microservices) can be integrated for single project or multiple projects.

✚ Spring, Spring Boot we can develop standalone apps, web applications, distributed apps and etc.

Enterprise apps = web applications + distributed applications.

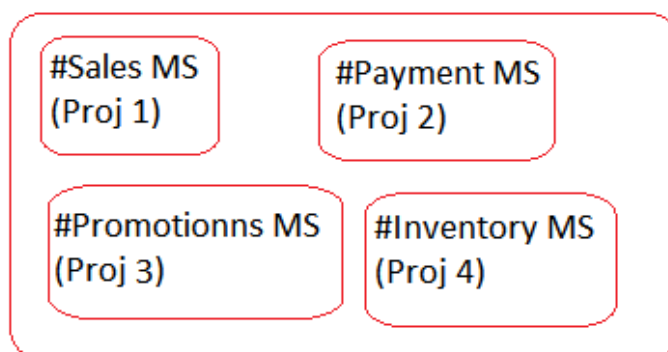
Note: We can develop these enterprise applications either having regular architecture nothing but developing different services as different modules of single project (Monolithic architecture), or we can develop using Microservice architecture nothing but developing different services as different projects and integrated them later.

E-Commerce Application (Monolithic architecture)



[These modules are different packages E-commerce App So these tightly coupled with one E-commerce App].

E-Commerce Application (Microservice Architecture)



[These Micro services are independent projects, so they can be used either on one E-commerce App or in multiple E-commerce App]

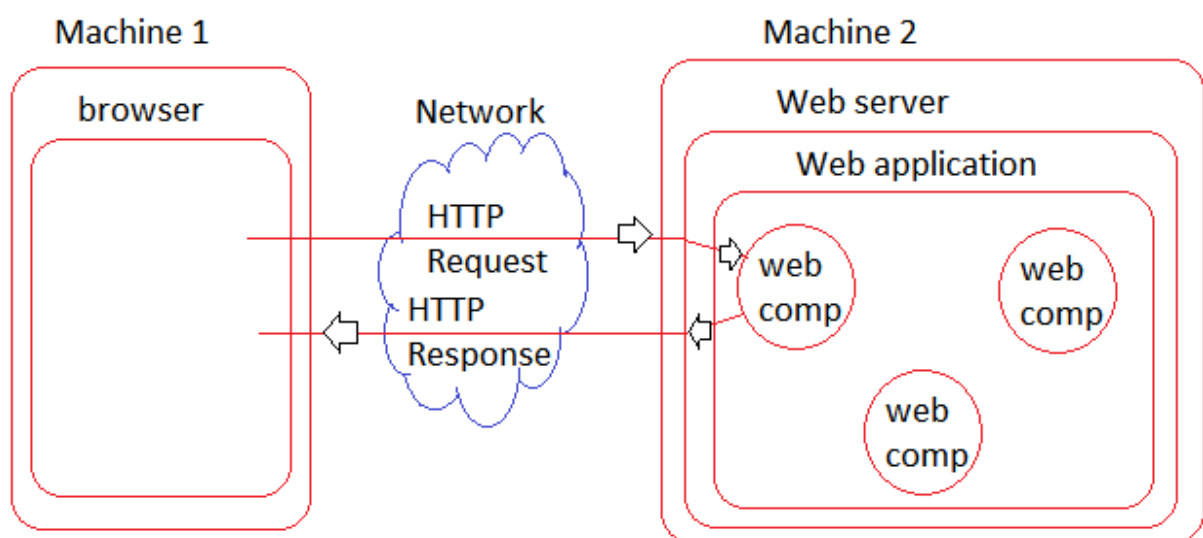
Different types of Application

Standalone Application:

- The application that is specific to one computer and allows only one user at a time to operate is called standalone application.
E.g.,
 - class with main (-) method.
 - calculator app, anti-virus s/w, desktop game and etc.
- We generally use Spring boot core module to develop these standalone applications.

Web Application:

- It is client-server application, where Server application is software application is handling HTTP requests, generating HTTP response. Where client application is browser s/w generating http requests, getting HTTP response.
E.g., nareshit.com, gmail.com, yahoo.com, flipkart.com, etc.



- Standalone applications are specific to one computer and will be operated by one user at time. So, manually execution takes place.
- Once the web application is kept on to the internet, it can request by lacs of users from different locations using browsers 24/7. So manual execution of web application and its web components is not going to work out.
- We need a special s/w to automate the execution of web application and its web components that is nothing but Web server s/w.
- Web server s/w is a piece of s/w that automates web application and its web components execution i.e., it takes the requests from clients

(browsers) by listening to request continuously, passes them to appropriate web components of web application, executes the web component dynamically to process the requests, and gathers the results to passes them back to clients (browsers) without human intervention.

- Web server: E.g., Tomcat (java), Jetty (java), Apache webserver (php), IIS (.net), NodeJS (JS), etc.
- Application server = Web server ++
E.g., WebLogic, Glassfish, Wildfly, jBoss, jRun, WebSphere and etc.

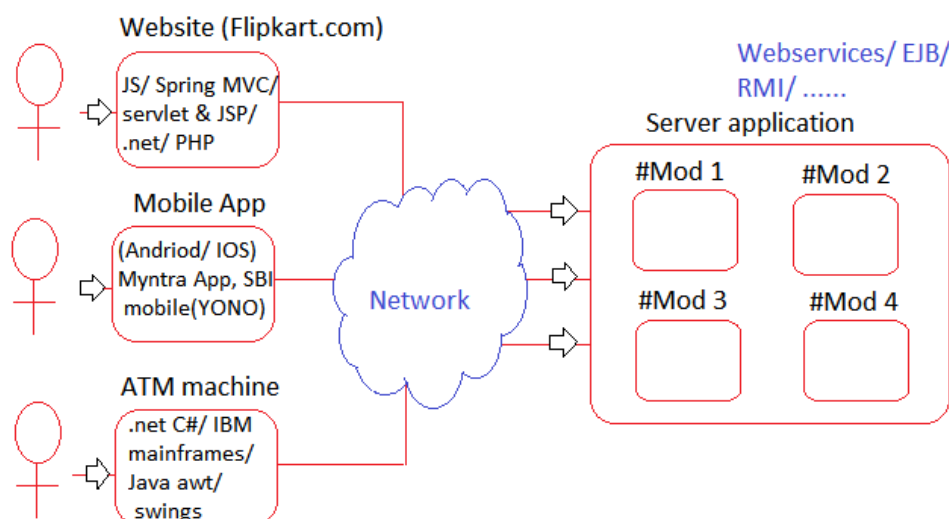
Note: Tomcat 7.x+ onwards can be called as Application server.

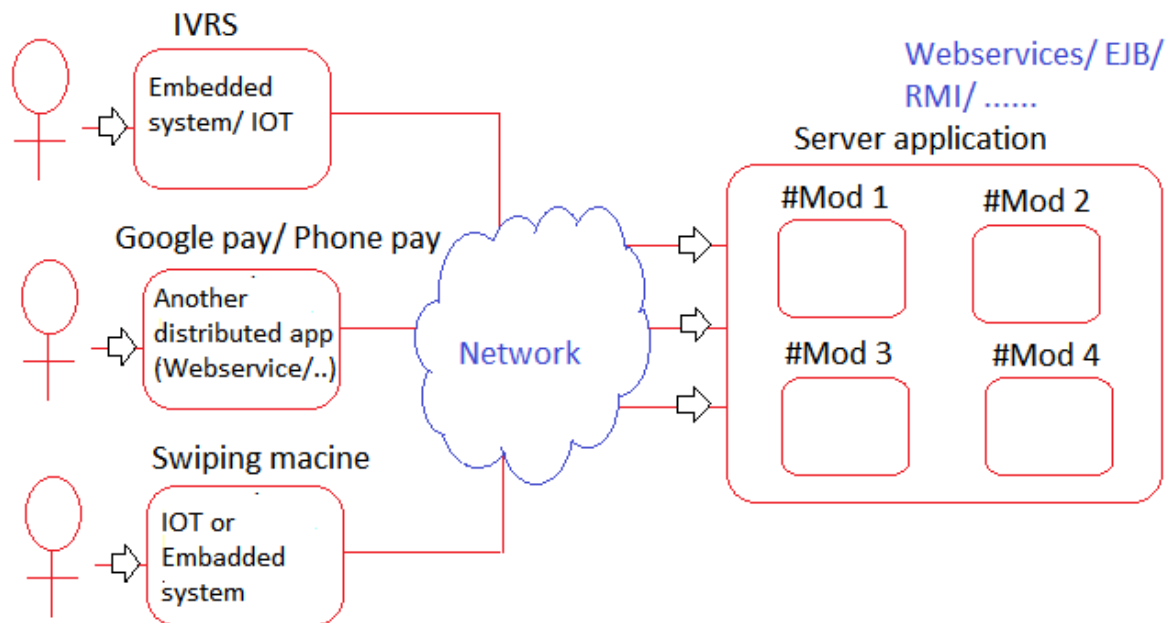
- ✚ Spring Boot MVC module is given to develop web application based MVC architecture (Model-View-Controller Architecture which is layered architecture).

Distributed Application/ Remoting Application

- It is Client-server application where both client and server applications are s/w applications interacting with other through method calls (No request and response here).
- Here the Server application of Distributed application allows different types of Local and remote client applications to access various services of server application.
- The different types of client's applications are Desktop client apps, web applications, mobile apps, IVRS Apps, other Distributed app, IoT devices.
- Generally, web applications are C to B applications (Customer to Business applications) whereas Distributed applications are B to B/ B2B applications.

SBI Banking App (Server app of Distributed App)





Note: IoT: Internet of Things (Where non-human beings uses the internet directly). E.g., Smart glasses, Smart watches, Google assistant, Smart ACs, Smart TV and etc.

E-Commerce application while using Card payment:

[Payment broker]
[Payment gateway]

Browser ---> Flipkart.com app ---> PayPal app ---> VISA/ MASTER/ Metro/

Web application
Distributed
Distributed

Rupay app ---> Banking app (SBI/ ICICI/ ...) ---> DB s/w

Distributed

E-Commerce application while using UPI payment:

Browser ---> Flipkart.com app ---> Google pay / Phone pay app ---> Banking app
(SBI/ ICICI/ ...) ---> DB s/w

Usage of Spring Boot Modules

- ✚ Spring Boot Rest (RESTful webservises environment) can be used to develop Distributed applications (both client and server applications).
- ✚ Spring Boot JDBC/ Spring Boot ORM, Spring Boot Data JPA modules are given to make Spring Boot standalone/ web application/ distributed applications interacting with SQL DB s/w and to perform CURD operations.
- ✚ Spring Boot Data NoSQL module is given to make Spring Boot standalone/ web application/ distributed applications interacting with NoSQL DB s/w and to perform CURD operations.

- Spring Boot AOP, Spring Boot Tx Mgmt are given to make Spring Boot standalone, web applications, distributed applications to added additional and optional and configurable (ability to enable or disable) secondary services/ logics like logging, auditing, Tx Mgmt, performance monitoring and etc.
- Spring Boot Security, OAuth modules are given to add security to web applications (Spring Boot MVC)/ Distributed applications (Spring Rest).
- Spring Boot Mail module is given to add Mailing facility to the application.
- Spring Boot Batch, Spring Boot Scheduling module are given to add to perform batch processing of huge amount of Data periodically and etc.
- Spring Boot Actuators module is given to add non-functional features to Spring Boot applications like Health metrics, memory management and etc.

Note: All the modules of Spring Boot, can also use as Spring modules.

Q. What is the different b/w Web application Distributed application?

Ans.

Web application	Distributed application
a. Client-server app where client is browser and server are s/w application.	a. Client-Server app where both client and server apps are s/w application.
b. It is C2B (Customer to Business application).	b. It is B2B (Business to Business application).
c. Allows only one type of Client (browser).	c. Allows different types of Client Apps (Desktop Apps/ mobile Apps/ IVRS Apps/ web applications/ other distributed apps/ IoT or Embedded System Apps/).
d. Communication takes place through request-response model.	d. Communication takes place through method calls.
e. It is Thin Client-Fat Server application.	e. It is Fat Server-Fat Client application.
f. We can use Servlet, JSP technologies of JAVA to develop web applications.	f. We can use RMI, EJB, JAX-RPC (Webservices), JAX-WS (Webservice), JAX-RS (Webservice) and etc. are JAVA technologies to develop Java based Distributed application.

g. Spring MVC, Spring Boot MVC, JSF, Struts, Webwork and etc. Java frameworks to develop Java web applications.	g. Apache AXIS, Apache CXF, Jersey, Spring Rest/ Spring Boot Rest and etc. are Java frameworks to develop java based Distributed applications.
h. E.g., Flipkart.com, amazon.in, nareshit.com and etc.	h. Google Pay/ Phone Pay/..., VISA App/ Master App/ ..., PayPal/ PayUMoney/ Rozar Pay/ ..., SBI App/ ICICI App, NSE app/ BSE App, ICC Score component, Weather Report component, COVID Report app, IRCTC App, OYO Server App and etc.

Basic Annotations in Spring Boot Programming

- ✚ The Spring annotations can also use in Spring Boot programming because Spring Boot internally uses Spring.
- ✚ Basic Annotations are following types

Annotations for Configurations:

These are useful to configure Java classes as Spring beans nothing but makes underlying IoC container to recognize Java classes as Spring bean by creating object for java class having bean id as object name (reference variable). All these are Stereo type annotations.

1. **@Component**: Makes Java class as Spring bean.
2. **@Service**: Makes Java class as Spring bean cum Service class (business logic -> calculations, sorting, indexing, filtering + Transaction Mgmt).
3. **@Repository**: Makes Java class as Spring bean cum DAO class (persistence logic - CURD operations) + JDBC Exception translation to Spring Exceptions.
4. **@Controller**: Makes Java class as Spring bean cum Controller class + HTTP Request handling.
5. **@RestController**: Makes Java class as Spring bean cum Rest Controller class + HTTP Request handling through method call in Restful webservice environment.
6. **@Configuration**: Makes Java class as Spring bean cum Configuration class (alternate Spring bean configuration file (XML file)).
7. **@ControllerAdvice**: To perform Exception handling related configuration in @Controller, @RestController classes (Here also Java class will be taken as Spring bean).
8. **@Bean**: Allows to makes the @Configuration class methods returned

Java class objects as Spring beans (We use this to configure for pre-defined Java classes as Spring beans).

Data Annotations:

These annotations are given to assign/ inject/ lookup data required for Spring bean class properties.

Given by Spring

1. **@Autowired**: To inject other Spring bean object to current Spring bean property.
2. **@Value**: To inject simple values, collections and etc.
3. **@ConfiguraitonProperties**: To inject simple values, collections and etc. (Given by Spring Boot).
4. **@Qualifier**: To solve the ambiguity problems related Dependency injection.
5. **@Primary**: To solve the ambiguity problems related Dependency injection.
6. **@Lookup**: To perform Lookup method injection.
Given by JSE, JEE modules as Java config annotations (Can be used in multiple frameworks)
7. **@Named**: To configure Java class as Spring bean and to resolve ambiguity problem.
8. **@Resource**: For injecting like @Autowired.
9. **@Inject**: For Injection like @Autowored.

Spring Bean Life cycle annotations:

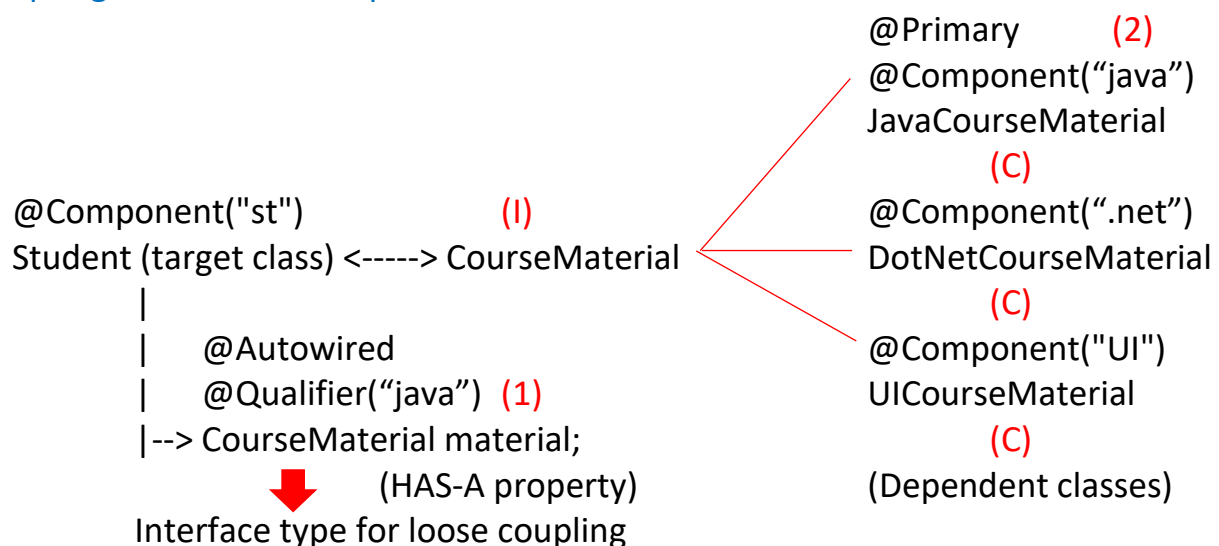
1. **@PostConstruct**: To make Java method of Spring bean class as init life cycle method.
2. **@PreDestroy**: To make Java method of Spring bean class as destroy life cycle method.

Misc. Annotations:

1. **@Import**: To Link one @Configuration class with another @Configuration class.
2. **@ImportResource**: To Link one Spring bean configuration file (XML file) with @Configuration class.
3. **@ComponentScan**: To make IoC container to look for stereo type annotations-based Java classes in the given Java packages and to make them as Spring beans.

4. **@EnableAutoConfiguration**: To enable auto configuration in Spring Boot application.
5. **@Lazy**: To enable Lazy instantiation on singleton scope Spring beans.
6. **@Scope**: To Specify Spring bean scope.
7. **@PropertySource, @PropertySources**: To configure Properties file(s) with Spring bean or @Configuration class.
8. **@SpringBootApplication**: Combination of 3 annotations to added Spring Boot behaviour to application
 - a. @Configuration/SpringBootConfiguration
 - b. @ComponentScan
 - c. @EnableAutoConfiguration
9. **@Required**: Deprecated from Spring 5.1 onwards.
10. **@DependsOn**: Forces the IoC container to initialize one or more beans. This annotation is directly used on any class or indirectly annotated with @Component or on methods annotated with @Bean.
11. **@Profile**: It is a logical grouping that can be activated programmatically. It can be used on type-level annotation on any class or it can be used as a meta-annotation for composing custom stereo type annotations or as a method-level annotation on any @Bean method. and etc.

Spring Boot 2nd Development:

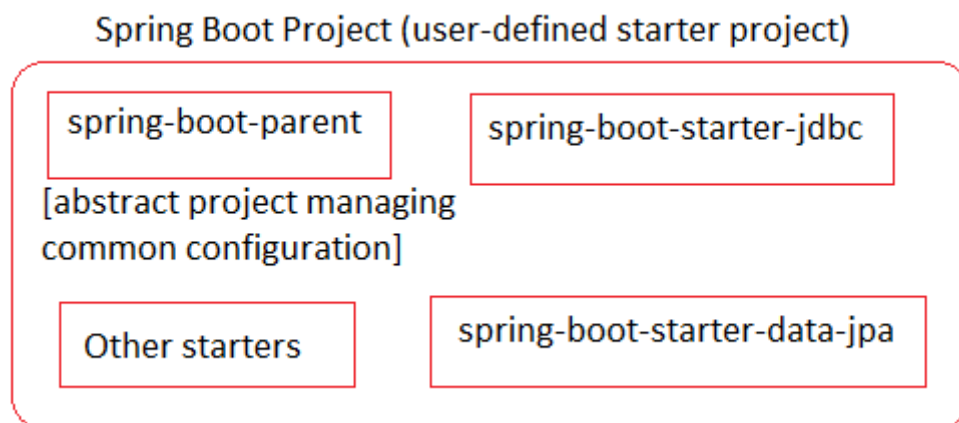


We can use either (1) or (2) to solve the ambiguity problem.

Note: Since multiple dependent classes are there. To make them as same type of classes, it is recommended to make those classes implementing common interface or extending from common super class.

Strategy Design Pattern:

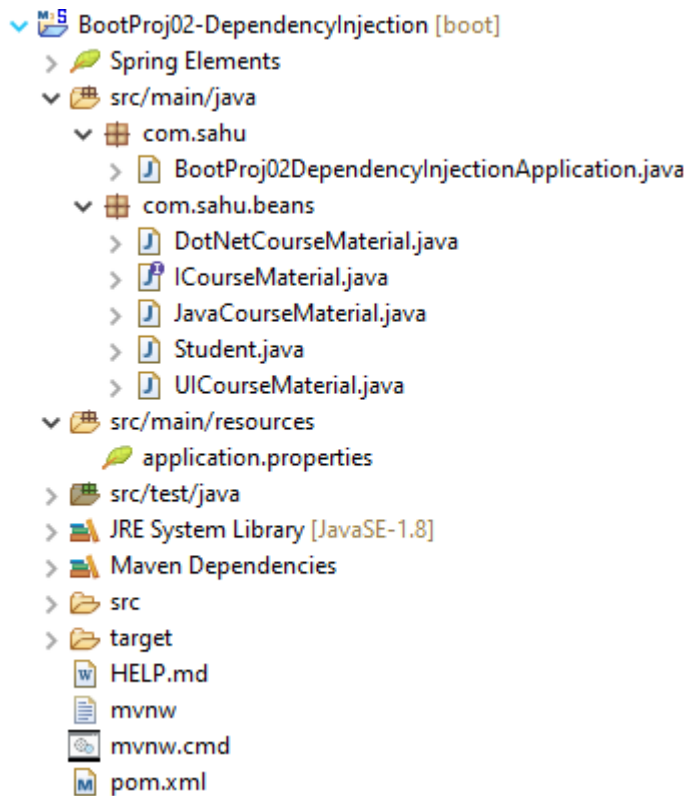
- This DP gives set of principles/ rules to develop target and dependent classes of Dependency Management as loosely coupled interchangeable parts.
- There are 3 principles/ rules
 - a. Prefer/ Favor composition over inheritance
 - b. Code to Interfaces/ abstract classes i.e., never code to concrete classes/ implementation classes (To achieve loose coupling).
 - c. Our code must be open for extension and must be closed for modification.
- Extension: Interfaces/ abstract classes-based programming helps us to keep code open for extension because it allows to add more dependent classes for the common interface.
- Modification: By taking the methods of both target and dependent classes as the final methods or by taking classes itself as final classes we can stop method overriding in sub classes (This is nothing but keeping our code closed for modification).



Note: While developing other Spring Boot applications/ projects we can also add previous projects as starters if needed.

Directory Structure of BootProj02-DependencyInjection:

- Develop the below directory structure using Spring Starter Project option and no starter project required and create the package and classes also.
- Then use the following code with in their respective file.



ICourseMaterial.java

```
package com.sahu.beans;

public interface ICourseMaterial {
    public String courseContent();
    public double price();
}
```

JavaCourseMaterial.java

```
package com.sahu.beans;

import org.springframework.stereotype.Component;

@Component("java")
public final class JavaCourseMaterial implements ICourseMaterial {

    public JavaCourseMaterial() {

        System.out.println("JavaCourseMaterial.JavaCourseMaterial()");
    }

    @Override
    public String courseContent() {
```

```

        return "1.OOPS\n2.Exception Handling\n3.Collection";
    }

    @Override
    public double price() {
        return 400;
    }
}

```

DotNetCourseMaterial.java

```

package com.sahu.beans;

import org.springframework.stereotype.Component;

@Component("dotNet")
public final class DotNetCourseMaterial implements ICourseMaterial {

    public DotNetCourseMaterial() {

        System.out.println("DotNetCourseMaterial.DotNetCourseMaterial()")
;
    }

    @Override
    public String courseContent() {
        return "1.C# OOPS\n2.C# Exception Handling\n3.C#
Collection";
    }

    @Override
    public double price() {
        return 300;
    }
}

```

UICourseMaterial.java

```

package com.sahu.beans;

import org.springframework.stereotype.Component;

```



```

@Component("ui")
public final class UICourseMaterial implements ICourseMaterial {

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

    @Override
    public String courseContent() {
        return "1.HTML\n2.CSS\n3.JavaScript";
    }

    @Override
    public double price() {
        return 200;
    }

}

```

Student.java

```

package com.sahu.beans;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Component;

@Component("stu")
public final class Student {
    @Autowired
    @Qualifier("java")
    private ICourseMaterial material;

    public void preparation(String examName) {
        System.out.println("Preparation started for "+examName);
        String courseContent = material.courseContent();
        double price = material.price();
        System.out.println("Preparation is going on
using\n"+courseContent+"\nwith price "+price);
        System.out.println("Preparation is completed for
"+examName);
    }

}

```

BootProj02DependencyInjectionApplication.java

```
package com.sahu;

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

import com.sahu.beans.Student;

@SpringBootApplication
public class BootProj02DependencyInjectionApplication {

    public static void main(String[] args) {
        //Get IoC container
        ApplicationContext context =
SpringApplication.run(BootProj02DependencyInjectionApplication.class,
args);

        //Get target Spring bean
        Student student = context.getBean("stu", Student.class);
        //invoke business method
        student.preparation("CTS-interview");
        //close container
        ((ConfigurableApplicationContext) context).close();
    }
}
```

Q. If Qualifier (-) and @Primary annotations are pointing two different Dependent Spring beans while injecting dependent to target bean class then what happens?

Ans. Qualifier (-) specified Dependent bean gets priority and that dependent class object will be injected to target spring bean class object.

Note:

- ✓ We can pass place holders \${<key>} only in @Value annotation to get values from properties files, i.e., we cannot pass them in other annotations like @Qualifier and etc. We can also pass place holders in various tags (<property>, <constructor-arg>, <alias> and etc.) of Spring bean configuration file.

- ✓ application.properties/ application.yml file of src/main/resources will be detected and loaded automatically as part of bootstrap activities of Spring Boot app that takes place in SpringApplication.run (-) method.
- ✓ Spring Beans and their bean ids management is always server side or backend programming activity especially changing from one dependent to another dependent is completely programmer's technical activity and we cannot involve non-technical end user in this operation. So, try to collect required dependent bean ids from properties files (like application.properties) or XML files (Spring bean configuration file) which under control of programmers.

Q. How can we change one dependent to another dependent in Spring Boot application dynamically at runtime through soft coding process to continue loose coupling for programmers.

Ans. We can do that by taking support of application.properties/ yml along with <alias> tag of Spring Bean Configuration file

Step 1: Keep any Spring app ready that is having ambiguity problem.

Step 2: Add user-defined message (key=value) in application.properties file by specifying required dependent spring bean id.

application.properties

```
course.choose=java
```

Step 3: Take Spring bean configuration file and provide alias name for bean id collected from properties file using <alias> tag.

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">
    <alias name="$${course.choose}" alias="courseID"/>
</beans>
```

\$\${course.choose}: Place holder representing bean id collect from application.properties file.

courseID: Alias's name

Step 4: Specify alias name @Qualifier (-) to resolve ambiguity problem.

Student.java

```
@Component("stu")
public final class Student {

    @Autowired
    @Qualifier("courseID")
    private ICourseMaterial material;
```

Step 5: Link applicationContext.xml file with Spring Boot application.

BootProj02DependencyInjectionApplication.java

```
@SpringBootApplication
@ImportResource("com/sahu/cfg/applicationContext.xml")
public class BootProj02DependencyInjectionApplication {
```

Note:

- ✓ To solve ambiguity problem without taking Spring bean configuration file, we need to use Spring/ Spring Boot profiles.
- ✓ To link Spring bean configuration file with @Configuration class, use @ImportResource. Similarly, to link one @Configuration class with another @Configuration class use @Import annotation.

Developing Spring Boot application using Java Config Annotations

- ✚ These are supplied by JSE, JEE modules (from Java 9 they are released as independent libraries).
- ✚ For these annotations the underlying framework or container or server decides the functionality. i.e.,
 - In Spring, the Spring f/w decides the functionality.
 - In Hibernate, the Hibernate f/w decides the functionality.
 - In Servlet, the ServletContainer decides the functionality.E.g., @Named, @Resource, @Inject, @PostConstruct, @PreDestroy and etc.
- **@Named:** To configure Java class as Spring bean and resolve ambiguity problem.
- **@Inject, @Resource:** Alternate to @Autowired for Dependency

Injection.

Note:

- ✓ @Resource cannot be applied at constructor injection.
- ✓ Spring Bean classes with Spring supplied annotations like @Component, @Autowired and etc. are invasive/ not non-invasive Spring beans. To make them non-invasive take the support of Java Config annotations.
- ✓ As of now very limited Java config annotations are available. So, it is practically impossible to develop entire Spring or Spring Boot app only with Java config annotations. So, prefer the following priority order
 - a. Java Config Annotations
 - b. Spring Annotations/ Spring Boot Annotations
 - c. Third Party Annotations
 - d. Custom Annotations
- ✓ Since there are no alternate frameworks for Spring frameworks. So, programmers are not thinking non-invasive Spring beans development and they are using Spring supplied annotations everywhere.

Directory Structure of BootProj03-DependencyInjection-JavaConfigAnnotation:

- Copy and paste BootProj02-DependencyInjection and change the name to BootProj03-DependencyInjection-JavaConfigAnnotation.
- Change the following code according to their respective file.
- And add the dependency Javax Inject in pom.xml.

DotNetCourseMaterial.java

```
@Named("dotNet")
public final class DotNetCourseMaterial implements ICourseMaterial {
```

JavaCourseMaterial.java

```
@Named("java")
public final class JavaCourseMaterial implements ICourseMaterial {
```

UICourseMaterial.java

```
@Named("ui")
public final class UICourseMaterial implements ICourseMaterial {
```

Student.java

```
@Named("stu")
public final class Student {

    //@Inject
    @Resource(name = "courseID")
    //@Named("courseID")
    private ICourseMaterial material;
```

Pom.xml

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

- ✚ @Inject can be used for field level, constructor level, setter method level, arbitrary method level injections. Whereas @Resource can be used for field level, setter method level, arbitrary method level injections.
- ✚ While working with @Inject we need to use another annotation @Named to solve ambiguity problem whereas while working @Resource its "name" param itself can be used to solve the same problem.
- ✚ To use different Java config annotations, we need to different jar files for working with @Inject, @Resource, @Named annotations add

```
<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId>
    <version>1</version>
</dependency>
```

Spring Boot Layered Application & AutoConfiguration

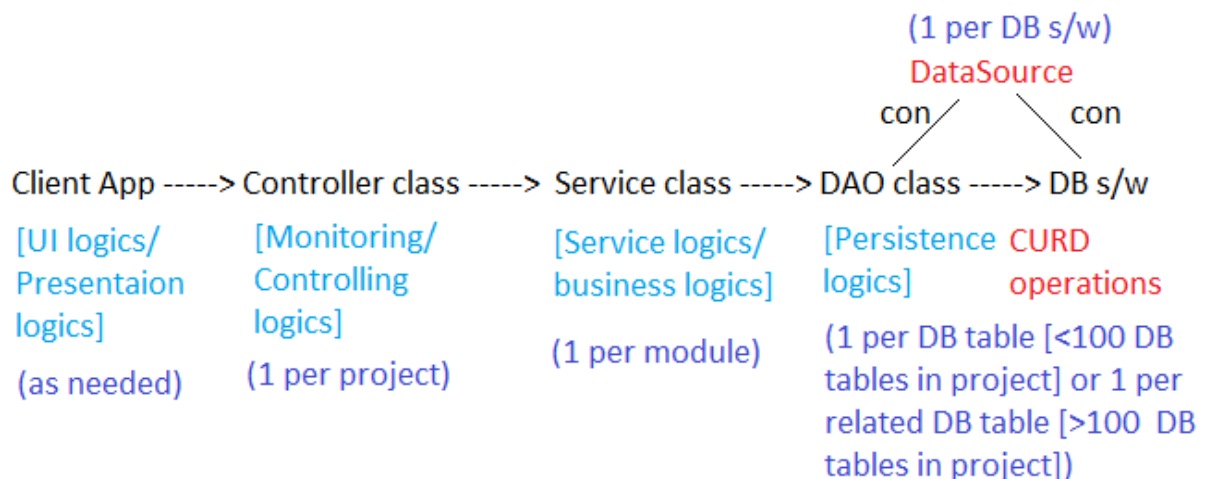
Problem: Keeping multiple logics in single class is bad practice because it makes the code as clumsy code without having clean separation between logics (It is like bachelor single room).

Solution: Develop the application as the layered app which makes the programmer to keep different logics different set of classes and making them

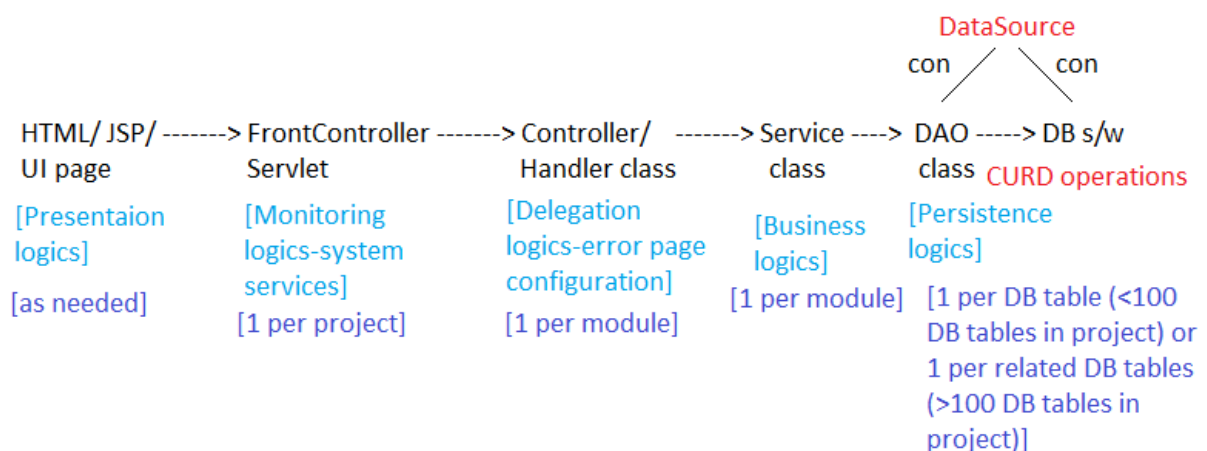
talking with each other (It is like 3BHK, 4BHK).

Note: Each layer of layered application represents specific type of logics having either one class or set of related classes.

Standalone Layered Application:



Web Based Layered Application:



- Project contains modules
- Module contains applications
- Application contains programs
- Program contains statements/ instructions

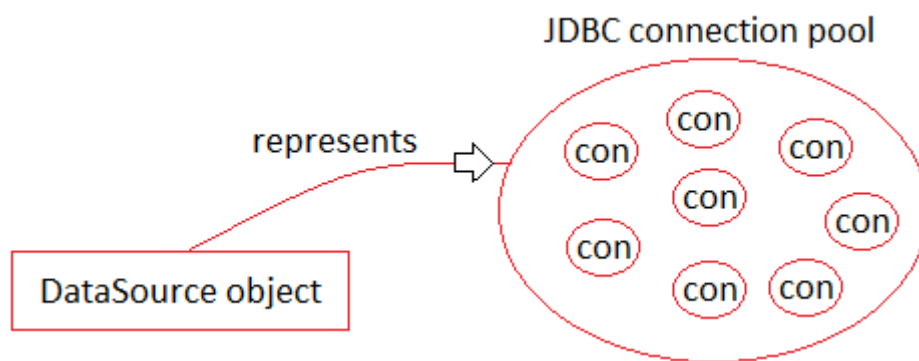
E.g., University Project

Counselling
academics
Sports and culture activities
Finance

Examinations
Placements
Payroll management
and etc.

JDBC Connection Pool

- It is a factory that contains set of readily available JDBC connection objects before actually being used.
- DataSource object represents JDBC connection pool and acts as entry point for JDBC connection pool i.e., all the JDBC connection objects from connection pool can be gathered and can be returned only through DataSource object.



To get one JDBC connection object from JDBC connection pool:

```
Connection con=ds.getConnection();
```

To release JDBC connection object back to JDBC connection pool:

```
con.close();
```

JDBC connection pool advantages:

- Reusability of JDBC connection objects
- With minimum JDBC connection objects we can make maximum applications requests talking to DB s/w.
- Managing JDBC connection objects, creating new connection objects and deleting idle JDBC connection objects will be taken care by JDBC connection pool itself.

Note: All JDBC connection objects in JDBC connection pool represents connectivity with same DB s/w.

E.g., JDBC connection pool for oracle means all the JDBC connection objects in the JDBC connection pool represents connectivity with the same Oracle DB s/w.

Different types of JDBC connection objects:

1. Direct JDBC connection
 - Created by the programmer manually.
E.g.,
`Class.forName(".....");`
`Connection con = DriverManager.getConnection (-, -, -);`
2. Pooled JDBC connection
 - Collected from JDBC connection pool through DataSource object.
E.g., `Connection con=ds.getConnection();`

Different types of JDBC connection pools:

1. Standalone JDBC connection pools
 - Suitable in standalone Apps
E.g., Apache DBCP, C3PO, HikariCP (best), Proxool (Given by third party vendors)
 - Spring supplied JDBC connection pool (DriverManagerDataSource, SingletonConnectionDataSource) only for testing.
 - Hibernate supplied built-in JDBC connection pool only for testing.
2. Server Managed JDBC connection pool
 - Suitable in the applications that are deployable in the servers (Web applications, Restful applications).
E.g., WebLogic managed JDBC connection pool, Tomcat managed JDBC connection pool, Glassfish managed JDBC connection pool, Wildfly managed JDBC connection pool (best) and etc.

Note: If one application wants to talk with more than one DB s/w then we need to take more than 1 DataSource objects representing more than 1 JDBC connection pool.

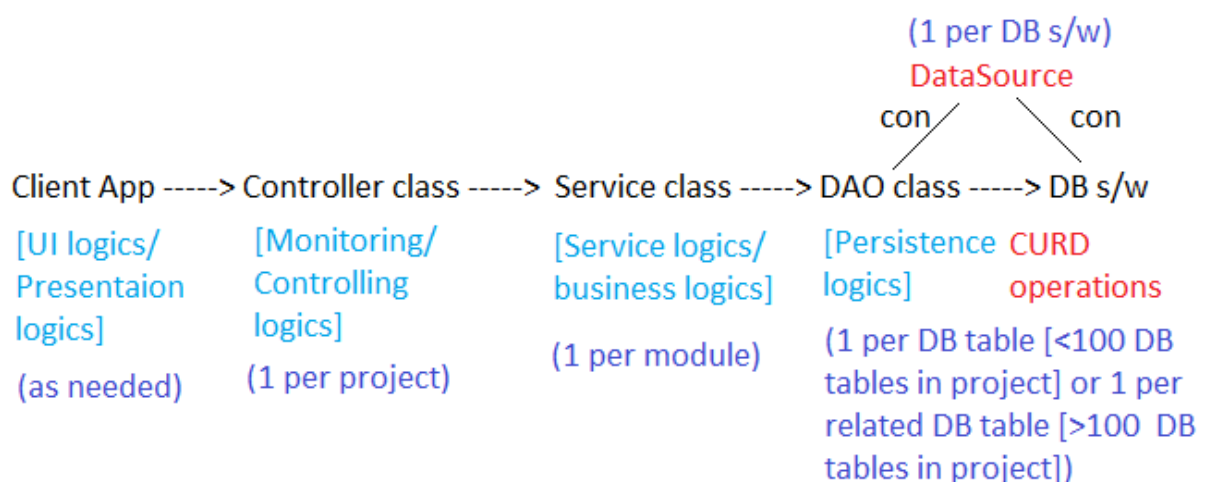
public class DriverManagerDataSource extends AbstractDriverBasedDataSource:

- Simple implementation of the standard JDBC DataSource interface, configuring the plain old JDBC DriverManager via bean properties, and returning a new Connection from every getConnection call.
- This class is not an actual connection pool, it does not actually pool Connections. It just serves as simple replacement for a full-blown connection pool, implementing the same standard interface, but creating new Connections on every call.

- This DriverManagerDataSource class was originally designed alongside Apache Commons DBCP and C3PO, featuring bean-style BasicDataSource / ComboPooledDataSource classes with configuration properties for local resource setups. For a modern JDBC connection pool, consider HikariCP instead, exposing a corresponding HikariDataSource instance to the application.

- ✚ Any DataSource object is the object of Java class implementing the common javax.sql.DataSource (I).

JDBC Connection Pool	DataSource class name
C3PO	<pkg>.C3PODataSource (C)
Apache DBCP	<pkg>.BasicDataSource (C)
Proxool	<pkg>.ProxoolDataSource (C)
HikariCP	<pkg>.HikariDataSource (C)



While developing the above app in Spring/ Spring Boot environment we can perform

- ✚ DS/ DataSource class object can be injected to DAO class object (DataSource can be taken as dependent Spring bean to DAO class target Spring bean).
- ✚ DAO class object can be injected to Service class object (DAO class can be taken as dependent Spring bean to Service class target Spring bean).
- ✚ Service class object can be injected to Controller class object (Service class can be taken as dependent Spring Bean get to Controller class target Spring bean).
- ✚ Client application can create/ get IoC container and can use ctx.getBean(-) method on Controller class object to invoke the business methods.

Java Bean

- Java Bean is a Java class that is developed by following standards.
- Java bean is always used helper classes to carry and pass multiple values across the main classes of same project or different Java projects.

Standards:

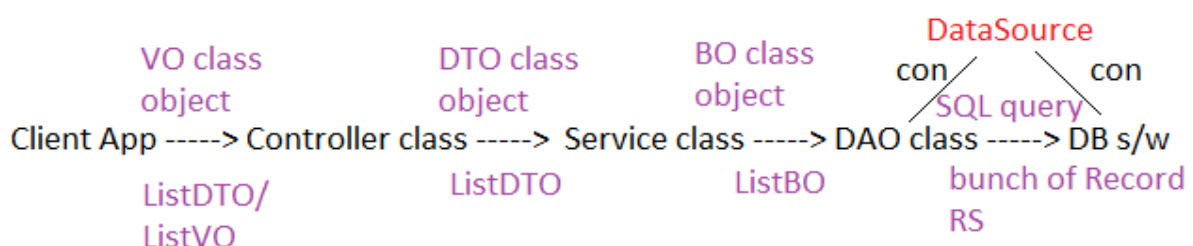
- Class must be public
- Recommended to implement `java.io.Serializable (I)`
- All member variables (bean properties) must be private and non-static
- Every bean property must have setter and getter method
- Must have 0-param constructor directly (given by programmer) or indirectly (generated by the Java compiler as the default constructor).

Based on kind of data we store there are 3 types of Java Beans:

1. **VO class (Value Object class):** The Java bean class object that holds either inputs or outputs (generally as string properties).
2. **DTO class (Data Transfer Object class):** The Java bean class object that holds Shippable/ transferable data either with in the Java project or across the multiple Java projects.
3. **BO class (Business Object class)/ Entity class/ Model class/ Persistence class:** The Java bean class object that holds persistable data (To be inserted to DB table as record) or persistent Data (collected from DB table record) is called BO class/ Entity class/ Model class. Generally, take this class on 1 per DB table having compatibility between Java bean class properties and DB table columns.

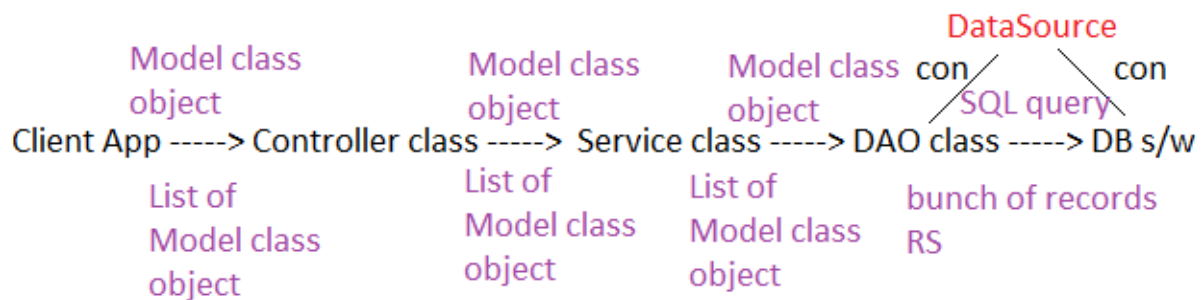
Note:

- ✓ If we take support of VO class, DTO class, BO class to carry data across the multiple layers of a project either for sending inputs or for retrieving outputs we will be having freedom to increase/ decrease inputs/ outputs in each layer.



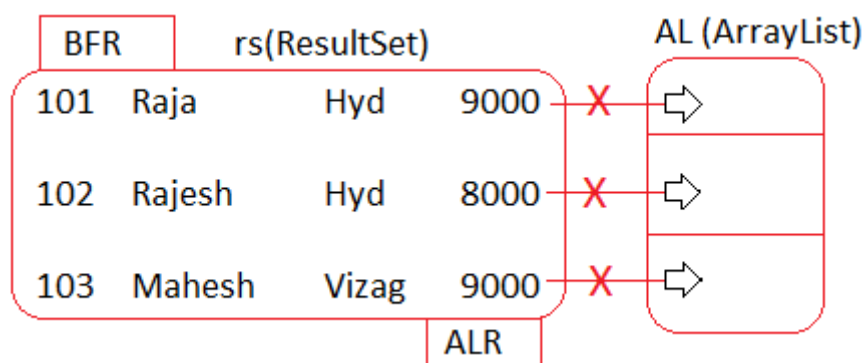
- ✓ If there is no need of increasing/ decreasing inputs/ outputs in each layer then we can take single java bean class (model class) to carry the

across the multiple layers (bit comprised).



Copying ResultSet Object records to ListBO collection:

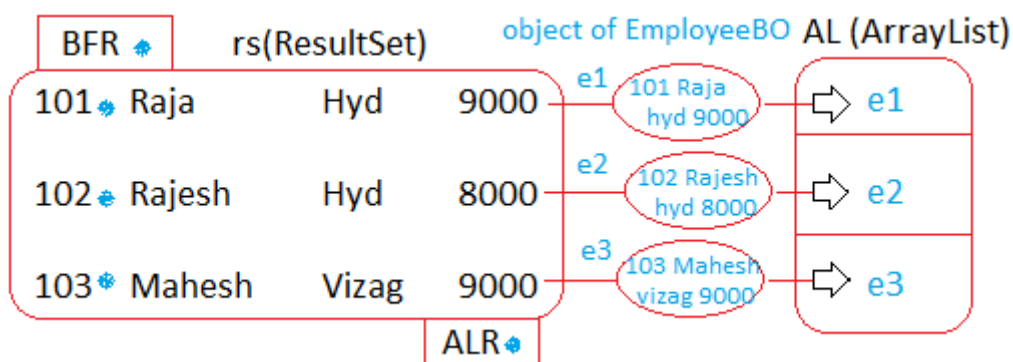
Problem:



Note:

- ✓ We cannot copy the records of RS to List collection directly because each record of RS contains multiple values (in fact multiple objects and multiple simple value) but each element of List collection can hold only one object.
- ✓ To overcome the above problem, copy each record of RS to an object of BO class and place that object in List collection.

Solution:



Sample code in DAO class

```
PreparedStatement ps = con.prepareStatement("SELECT EMPNO, ENAME,  
                                           EADD, SALARY FROM EMPLOYEE_INFO");
```

```
ResultSet rs = ps.executeQuery();
```

```
//copy RS records to ArrayList collection
```

```
List<EmployeeBO> listBO = new ArrayList<> ();
```

```
Employee ebo = null;
```

```
while (rs.next()) {
```

```
    //copy each record of RS to each object of BO class
```

```
    ebo = new EmployeeBO ();
```

```
    ebo.setEmpno(rs.getInt(1));
```

```
    ebo.setName(rs.getString(2));
```

```
    ebo.setEadd(rs.getString(3));
```

```
    ebo.setSalary(rs.getDouble(4));
```

```
    //add each object BO class to ListBO
```

```
    listBO.add(ebo);
```

```
}
```

@Data (getters + setters + toString () + hashCode () + equals (-) + param
constructor)

```
public class EmployeeBO {
```

```
    private int empno;
```

```
    private String ename;
```

```
    private String eadd;
```

```
    private double salary;
```

```
    //setters (4) && getters (4)
```

(Do not generate manually use Lombok API)

```
    .....
```

```
}
```

✚ If we add spring-boot-starter-jdbc (part of Spring boot starters) to Spring Boot as dependency then we get the following spring beans through auto configurations.

- DataSource object (HikariDataSource object)

Get inputs from application.properties/ yml to creates JDBC connection object in JDBC connection pool represented by DataSource object.

- JdbcTemplate object
- NamedParameterJdbcTemplate object
- DataSourceTransactionManager object, etc.

(Part of Spring
JDBC API)

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

Ans.

- If Spring/ Spring Boot project is standalone application then use third party JDBC connection pools/ DataSource like Apache DBCP, C3PO, HikariCP (best) and etc.
- If Spring/ Spring Boot project is web application or Distributed application which is deployable application in the servers then use the underlying Server Managed JDBC connection pool.
E.g., Tomcat managed JDBC connection pool, Wildfly managed JDBC connection pool and etc.

Note: Do not use "DriverManagerDataSource", "SingleConnectionDataSource" based JDBC connection pool because they do not pool the JDBC connections.

Q. Can we use third party JDBC connection pools like HikariCP in the server managed applications like web applications and distributed applications?

Ans. Yes, we can use but not recommended.

Q. Can we use Server managed JDBC connection pool in standalone app?

Ans. Yes, but not recommended because taking web server/ application server only for JDBC connection pool is meaningless.

Storyboard for Spring Boot application

Develop a Spring Boot Standalone Layered Application that gives Employee details based on given "n" Designations:

application.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
(Keys are fixed)
```

DAO Layer:

IEmployeeDAO.java

```
public interface IEmployeeDAO {
    public List<EmployeeBO> getEmpsByDesg(String condition) throws
                                                Exception;
}
```

EmployeeDAOImpl.java

```
@Repository("empDAO")
public class EmployeeDAOImpl implements IEmployeeDAO {
    @Autowired
    private DataSource ds; //HAS-A property
                                (o)
    public List<EmployeeBO> getEmpsByDesg(String condition)throws
                                                Exception {
        .....
        (p) ..... //logic to get RS and converting it to ListBO class object
        .....
        return listBO; (q)
    }
}
```

Note: Spring Boot Autoconfiguration gives certain pre-defined Java class objects as Spring beans based on the starter, we have whereas @Autowired can be used to inject such Spring bean class object to other Spring bean properties.

Service Layer:

IEmployeeMgmtService.java

```
public interface IEmployeeMgmtService {
    public List<EmployeeDTO> fetchEmpsByDesgs(String desgs[]) throws
                                                Exception;
}
```

EmployeeMgmtServiceImpl.java

```
@Service("empService")
public class EmployeeMgmtServiceImpl implements IEmployeeMgmtService {
    @Autowired
    private IEmployeeDAO dao;
                                (m)
    public List<EmployeeDTO> fetchEmpsByDesgs(String desgs[]) throws
                                                Exception {
        //use DAO
        (r) List<EmployeeBO> listBO=dao.getEmpsByDesg(condition); (n)
        //convert listBO to listDTO
        .....
        .....
    }
}
```

```

        return listDTO; (s)
    }
}

Controller Layer:
MainController.java controller class
@Controller ("controller")
public class MainController {
    @Autowired
    private IEmployeeMgmtService service;

    public List<EmployeeVO> displayEmpsByDesgs(String desgs[])throws
                                                Exception {

        //use Service
        (t) List<EmployeeDTO> listDTO =
                                service.fetchEmpsByDesgs(desgs); (l)
        //convert listDTO to listVO
        .....
        .....
        return listVO; (u)
    }
}

```

Note: VO, DTO, BO classes will not be configured as Spring beans. So, their objects will be created in different layers manually by using new operation.

Client App

```

@SpringBootApplication
public class RealtimeDITest {
    (a)
    public static void main (String [] args) {
        //get IoC container
        (d) Pre-instantiation of Singleton scope beans and injection
        (f) ApplicaitonContext ctx = (b) (c) AutoConfiguration
            SpringApplication.run(RealtimeDITest.class, args);

        //read inputs using Scanner
        .....
        ..... //desgs count and desg names as array
        .....
    }
}

```



```

// get Controller class object (g)
(i?) MainController controller =
        ctx.getBean("controller",MainController.class);
try {
(v)    List<EmployeeVO> listVO =
        controller.displayEmpsByDesgs(desgs); (j)
        listVO.forEach(System.out::println); (w)
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    //close IoC container
    ctx.close();
}
}

```

(h?) Internal Cache of IoC Container (e)

controller	MainController class object reference
empService	EmployeeMgmtServiceImpl class object reference
empDAO	EmpDAOImpl class object reference
hikariDataSource	HikariDataSource class object reference

(c) **AutoConfiguration** – HikariDataSource, JdbcTemplate, NamedParameterJdbcTemplate and etc. class objects we get in the part of AutoConfiguration based on Spring boot JDBC starters.

(d) **Pre-instantiation** – DAO, Service, Controller class object in pre-instantiation and injected also.

Procedure to develop Mini Project/ Layered application

Step 1: Keep the following software setup ready

- Oracle 11g+
- Eclipse 2020 + with STS plugin

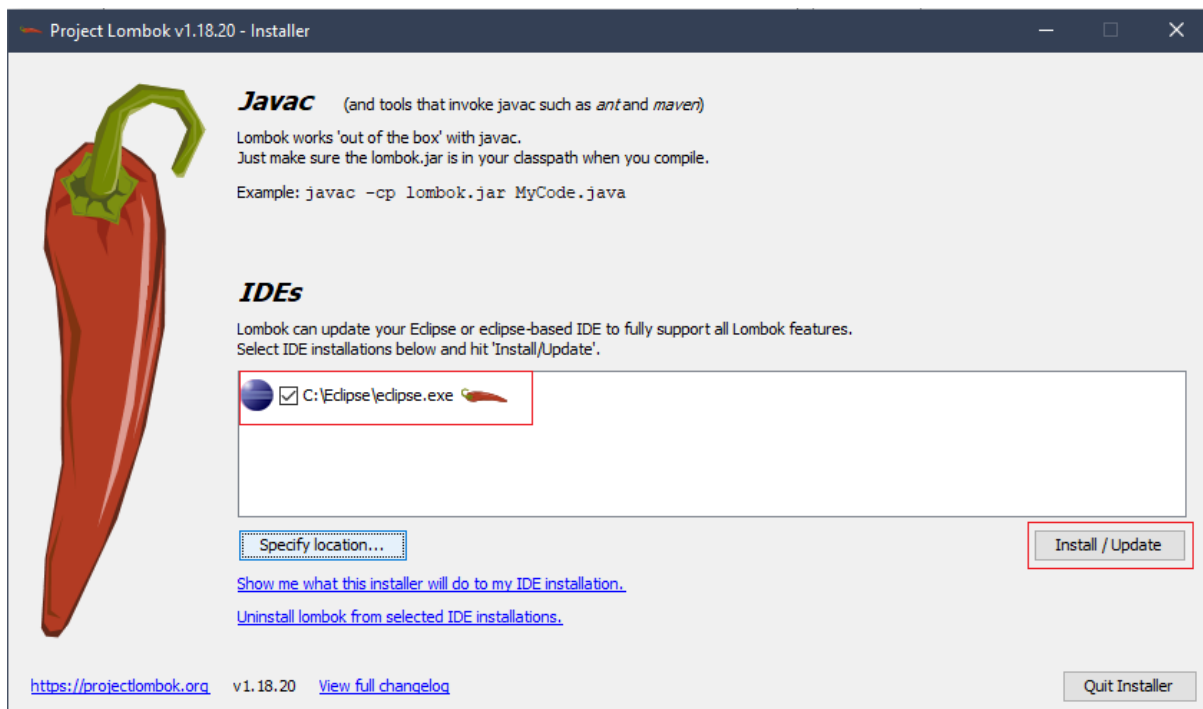
Step 2: Make sure that Oracle DB s/w is having "emp" readymade DB table. otherwise copy and paste SQL queries from scott.sql.

(c:\oracle\app\oracle\product\11.2.0\server\rdbms\admin) file on SQL

prompt to get the default table of scott user.

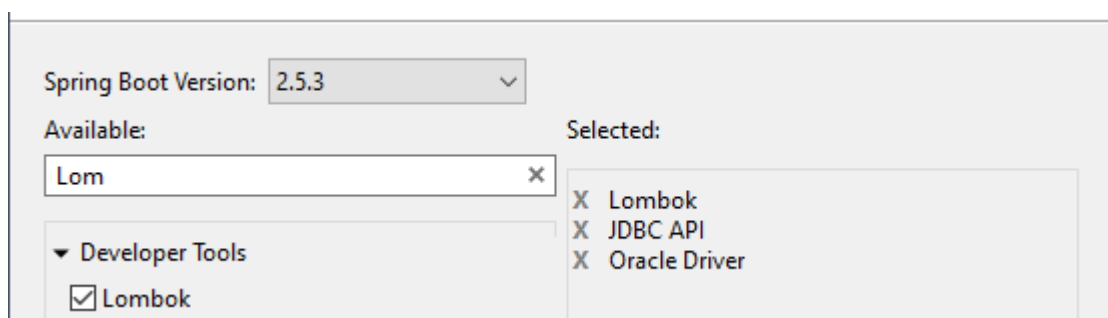
Step 3: Make sure that Lombok API is configured with Eclipse IDE.

- Download lombok-<ver>.jar file [\[Download\]](#).
- Launch the jar by clicking on that and configure with Eclipse IDE by choose your IDE and click Install/ Update, then will got successful message, then click on Quit Installer.
- Restart/ Lunch your Eclipse IDE freshly.



Step 4: Create Spring Starter Project and add the following stater/ dependencies while creating the Project itself.

- JDBC API (To get spring-boot-starter-JDBC starter) (gives spring jdbc, HikariCP and relevant jar files)
- lombok (gives lombok-<ver>.jar)
- Oracle driver (gives ojdbc8.jar)



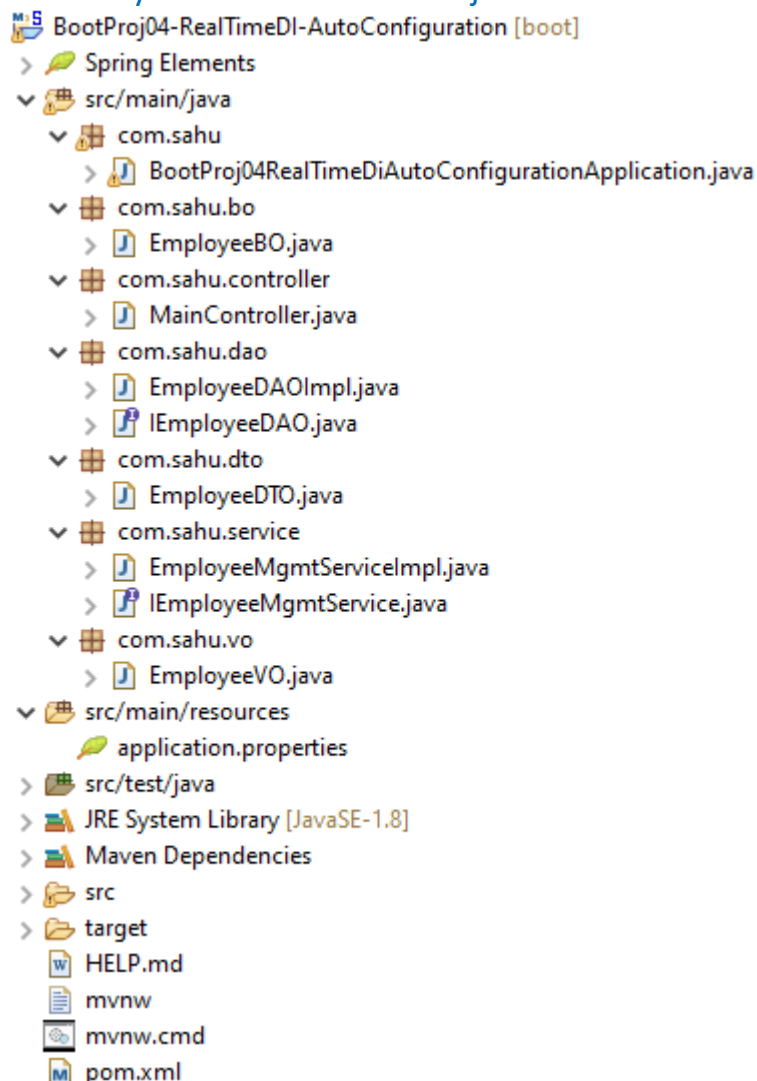
Step 5: Create packages and develop the application based on the story board knowledge.

- com.sahu.vo
- com.sahu.dto
- com.sahu.bo
- com.sahu.dao
- com.sahu.service
- com.sahu.controller

Step 6: After development had completed, then run as Spring Boot application.

Note: While developing Java Beans it recommended to take wrapper data type (Like Integer, Long, Double and etc.) bean properties rather simple data types (int, long, double and etc.) because simple type numeric variables hold "0" as default values whereas wrapper data type properties hold "null".

Directory Structure of BootProj04-RealTimeDI-AutoConfiguration:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also.
- During creation following starter project is required
 - JDBC API
 - Oracle Driver
 - Lombok
- Then use the following code with in their respective file.

EmployeeVO.java

```
package com.sahu.vo;

import lombok.Data;

@Data
public class EmployeeVO {
    private String srNo;
    private String empNo;
    private String ename;
    private String job;
    private String sal;
    private String deptNo;
    private String mgrNo;
}
```

EmployeeDTO.java

```
package com.sahu.dto;

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

@Data
public class EmployeeDTO implements Serializable {
    private Integer srNo;
    private Integer empNo;
    private String ename;
    private String job;
    private Double sal;
    private Integer deptNo;
    private Integer mgrNo;
}
```

EmployeeBO.java

```
package com.sahu.bo;

import lombok.Data;

@Data
public class EmployeeBO {
    private Integer empNo;
    private String ename;
    private String job;
    private Double sal;
    private Integer deptNo;
    private Integer mgrNo;
}
```

application.properties

```
#DataSource inputs
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
#The no.of idle connection objects that JDBC connection pool should
maintain
spring.datasource.hikari.minimum-idle=10
#The maximum JDBC connection objects that JDBC connection pool have
spring.datasource.hikari.maximum-pool-size=100
#Idle time out period for JDBC connection object in the JDBC connection
pool
spring.datasource.hikari.idle-timeout=20
```

IEmployeeDAO.java

```
package com.sahu.dao;

import java.util.List;
import com.sahu.bo.EmployeeBO;

public interface IEmployeeDAO {
    public List<EmployeeBO> getEmpsByDesgs(String condition) throws
Exception;
}
```

EmployeeDAOImpl.java

```
package com.sahu.dao;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

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

import com.sahu.bo.EmployeeBO;

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

    private static final String GET_EMPS_BY_DEGS = "SELECT EMPNO,
ENAME, JOB, SAL, DEPTNO, MGR FROM EMP WHERE JOB IN";

    @Autowired
    private DataSource ds;

    @Override
    public List<EmployeeBO> getEmpsByDesgs(String condition) throws
Exception {
        List<EmployeeBO> listBO = null;
        try ( // Get Pooled JDBC connection
            Connection con = ds.getConnection();
            // Statement Object
            Statement st = con.createStatement();
            // Send and execute SQL query in DB s/w
            ResultSet rs =
st.executeQuery(GET_EMPS_BY_DEGS + condition + "ORDER BY JOB");) {
            // Convert RS to ListBO
            listBO = new ArrayList<>();
            EmployeeBO bo = null;
            while (rs.next()) {
                bo = new EmployeeBO();
                bo.setEmpno(rs.getInt(1));
                bo.setEname(rs.getString(2));
                bo.setJob(rs.getString(3));
                bo.setSal(rs.getInt(4));
                bo.setDeptno(rs.getInt(5));
                bo.setMgr(rs.getInt(6));
            }
        }
        return listBO;
    }
}
```

```

        bo.setEmpNo(rs.getInt(1));
        bo.setEname(rs.getString(2));
        bo.setJob(rs.getString(3));
        bo.setSal(rs.getDouble(4));
        bo.setDeptNo(rs.getInt(5));
        bo.setMgrNo(rs.getInt(6));
        // Add each object of Employee BO to LisBO
        listBO.add(bo);
    }

    } catch (SQLException se) {
        se.printStackTrace();
        throw se;
    } catch (Exception e) {
        e.printStackTrace();
        throw e;
    }
    return listBO;
}
}

```

IEmployeeMgmtService.java

```

package com.sahu.service;

import java.util.List;

import com.sahu.dto.EmployeeDTO;

public interface IEmployeeMgmtService {
    public List<EmployeeDTO> fetchEmpsByDegr(String degra[]) throws
    Exception;
}

```

EmployeeMgmtServiceImpl.java

```

package com.sahu.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.sahu.bo.EmployeeBO;
import com.sahu.dao.IEmployeeDAO;
import com.sahu.dto.EmployeeDTO;

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

    @Autowired
    private IEmployeeDAO dao;

    @Override
    public List<EmployeeDTO> fetchEmpsByDegs(String[] desgs) throws
Exception {
        // Convert desg[] into SQL IN clause String Condition
        StringBuffer condition = new StringBuffer("(");
        for (int i = 0; i < desgs.length; i++) {
            if (i == desgs.length - 1)
                condition.append("'" + desgs[i] + "'");
            else
                condition.append("'" + desgs[i] + "',");
        }
        // Use DAO
        List<EmployeeBO> listBO =
dao.getEmpsByDegs(condition.toString());
        // Convert ListBO to ListDTO
        List<EmployeeDTO> listDTO = new ArrayList<>();
        listBO.forEach(bo -> {
            EmployeeDTO dto = new EmployeeDTO();
            BeanUtils.copyProperties(bo, dto);
            dto.setSrNo(listDTO.size() + 1);
            // add each object of DTO class into listDTO
            listDTO.add(dto);
        });
        return listDTO;
    }
}

```


MainController.java

```
package com.sahu.controller;

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

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

import com.sahu.dto.EmployeeDTO;
import com.sahu.service.IEmployeeMgmtService;
import com.sahu.vo.EmployeeVO;

@Controller("mainController")
public class MainController {

    @Autowired
    private IEmployeeMgmtService service;

    public List<EmployeeVO> showEmpsByDesgs(String desgs[]) throws
Exception {
        //User service
        List<EmployeeDTO> listDTO = service.fetchEmpsByDegr(desgs);
        //Convert listDTO to listVO;
        List<EmployeeVO> listVO = new ArrayList<>();
        listDTO.forEach(dto->{
            EmployeeVO vo = new EmployeeVO();
            BeanUtils.copyProperties(dto, vo);
            vo.setSrNo(String.valueOf(dto.getSrNo()));
            vo.setEmpNo(String.valueOf(dto.getEmpNo()));
            vo.setSal(String.valueOf(dto.getSal()));
            vo.setDeptNo(String.valueOf(dto.getDeptNo()));
            vo.setMgrNo(String.valueOf(dto.getMgrNo()));
            //add vo to ListVO
            listVO.add(vo);
        });
        return listVO;
    }
}
```

BootProj04RealTimeDiAutoConfigurationApplication.java

```
package com.sahu;

import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

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

import com.sahu.controller.MainController;
import com.sahu.vo.EmployeeVO;

@SpringBootApplication
public class BootProj04RealTimeDiAutoConfigurationApplication {

    public static void main(String[] args) {
        // Read inputs from user
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Desgination count : ");
        int count = sc.nextInt();
        String desgs[] = null;
        if (count >= 1) {
            desgs = new String[count];
        } else {
            System.out.println("Invalid desgination count");
            return;
        }
        for (int i = 0; i < count; i++) {
            System.out.print("Enter designation " + (i + 1) + " : ");
            desgs[i] = sc.next();
        }
        // Get IoC container
        ApplicationContext ctx =
        SpringApplication.run(BootProj04RealTimeDiAutoConfigurationApplication.
        class, args);
        //Get controller class object
        MainController controller = ctx.getBean("mainController",
        MainController.class);
    }
}
```

```

        //Invoke method
        try {
            List<EmployeeVO> listVO =
controller.showEmpsByDesgs(desgs);
            System.out.println(Arrays.toString(desgs));
            listVO.forEach(System.out::println);
        }
        catch (Exception e) {
            e.printStackTrace();
            System.out.println("Some internal problem :
"+e.getMessage());
        }
        //close IoC container
        ((ConfigurableApplicationContext) ctx).close();
    }
}

```

Q. How do you propagate the exception raised in DAO class to client application (in standalone App) or DAO class to Servlet component/ JSP component/ browser (in Web application)?

Ans.

While developing Standalone Layered application (Both Spring and Non-Spring environment):

- The DAO, Service, Controller classes/ layers should not just catch and handle the exception rather they should propagate the exception using throw, throws keywords whereas client application should catch and handle exception to display non-technical error messages to end-user.

While developing Web based Layered application (Servlet, JSP based):

- The DAO, Service layers/ classes should not just catch and handle the exception rather they should propagate the exception using throw, throws keywords whereas Controller Servlet component should catch and handle exception to display non-technical error messages to end-users through error HTML pages / JSP pages.

While developing Spring Web MVC/ Spring Boot Web MVC Layered web application:

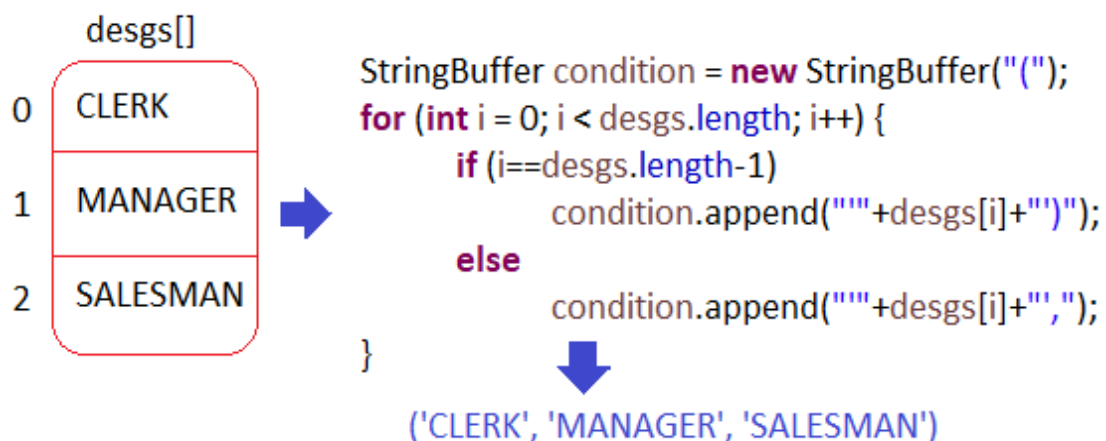
- The DAO, service layers/ classes should not just catch and handle the exception rather they should propagate the exception using throw,

throws keywords where controller/ handler classes should have special logical in handler methods to delegate the control to error pages to display error messages as the non-technical guiding messages.

Q. Where did you use StringBuffer/ StringBuilder in your project?

Ans.

- String constants and String class objects are immutable in Java (By default thread safe).
- StringBuffer class objects are mutable in Java (By default thread safe because all the methods are synchronized prefer in web applications for thread safe).
- StringBuilder class objects are mutable in Java (By default not thread safe because all the methods are non-synchronized prefer in web applications towards achieve performance).
- Instead of doing multiple modifications on same String directly which immutable, it is recommended to use StringBuffer or StringBuilder and perform all the modifications and convert into String lastly.



Q. Where did you use Simple Statement dealing with static SQL query in your project?

Ans. While Employee details based dynamic count of given employee desgs/ Jobs we need to work with Simple Statement having static SQL query.

Q. Where did you use try, catch, throws, throw and finally in a single method definition?

(or)

Q. Where did you use try with resources, catch, throws, throw y in a single method definition?

Ans. While developing layered applications having DAO class code in plain JDBC then we need to use the above said multiple things.

Note:

- ✓ Checked exceptions do not support exception propagation by default because it makes the programmer to catch and handle the exceptions. In that process Exception will be caught and eaten in the same method definition (Exception suppressed). So, we need to use throws, throw statement explicitly to perform Exception propagation.
- ✓ Unchecked exceptions support exception propagation by default because it makes the programmer not to catch and handle the exceptions. In that process Exception will not be caught and eaten in the same method definition (Exception suppressed) because programmer never catches and handles the exception. So, the raised exception will passed/ propagated to caller (another layer).

✚ In Service class we generally convert listBO to listDTO as shown below.

//Use DAO

```
List<EmployeeBO> listBO = dao.getEmpsByDesgs(condition.toString());
```

//Convert ListBO to ListDTO

```
List<EmployeeDTO> listDTO = new ArrayList<> ();
```

```
listBO.forEach(bo -> {
```

```
    EmployeeDTO dto = new EmployeeDTO();
```

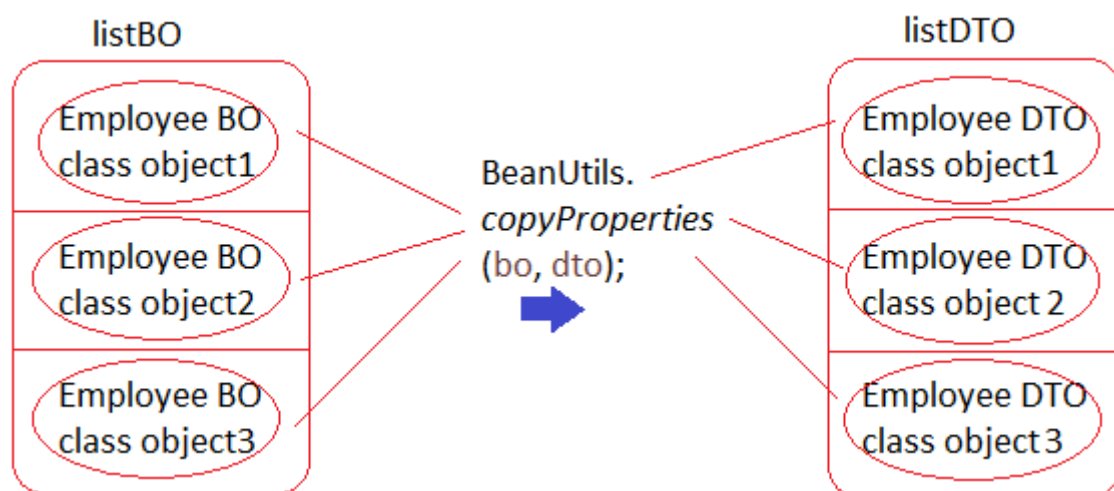
```
    BeanUtils.copyProperties(bo, dto);
```

```
    dto.setSrNo(listDTO.size()+1);
```

//add each object of DTO class into listDTO

```
listDTO.add(dto);
```

```
});
```



Q. When spring-boot-starter-jdbc starter is added to Spring Boot application can you explain how AutoConfiguration is taking on multiple Spring beans like

HikariDataSource/ <other DataSource>, JdbcTemplate and etc.?

Ans. When `springApplication.run(-, -)` method is called from main class cum configuration class multiple things will happen. One aspect/ operation is AutoConfiguration of Spring beans based on starters/ jars that are added CLASSPATH. This AutoConfiguration takes place as shown below

- IoC container searches in all jar files added to CLASSPATH for `spring.factories` file and finds in multiple jar files, in those multiple `spring.factories` files it looks for the key `"org.springframework.boot.autoconfigure.EnableAutoConfiguration"` and finds in `META-INF/spring.factories` files of `spring-boot-autoconfigure-<ver>.jar` and collects all the values added to that key which are multiple readymade configuration classes like this.

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=/  
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfigura  
tion,\br/>org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfigu  
ration,\br/>org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfi  
guration,\br/>org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfig  
uration,\br/>org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionM  
anagerAutoConfig,\
```

- Takes each Configuration class and its nested configuration classes and imported (`@Import`) Configuration classes and calls all the `@Bean` methods to make certain readymade classes like `HikariDataSource`, `JdbcTemplate` and etc. classes Spring beans through instantiation process.

Short Answer:

- IoC Container searches for `spring.factories` files in all jar files added to CLASSPATH gets different Configuration (`@Configuration`) classes against the key `org.springframework.boot.autoconfigure.EnableAutoConfiguration` and executes all the `@Bean` methods of those Configuration classes to make java classes of currently added starter jar files as Spring beans.

JdbcTemplateConfiguration.java

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(JdbcOperations.class)
class JdbcTemplateConfiguration {

    @Bean
    @Primary
    JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties
properties) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcProperties.Template template = properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int)
template.getQueryTimeout().getSeconds());
        }
        return jdbcTemplate;
    }
}
```

Note: By default, every Spring Boot Application of 2.5 version tries to perform AutoConfiguration on all Spring beans (@Bean methods) that are there 131 readymade Configuration classes. (These classes will multiple nested configuration classes and imported Configuration classes to participate in AutoConfiguration).

- ✚ As of now Spring Boot 2.5 is using two DataSource as part AutoConfiguration if we add "spring-boot-starter-jdbc"
 - HikariCP DataSource (HikariDataSource) (default)
 - Apache DBCP2 DataSource (BasicDataSource) (only when HikariCP jars are not there in CLASSPATH)

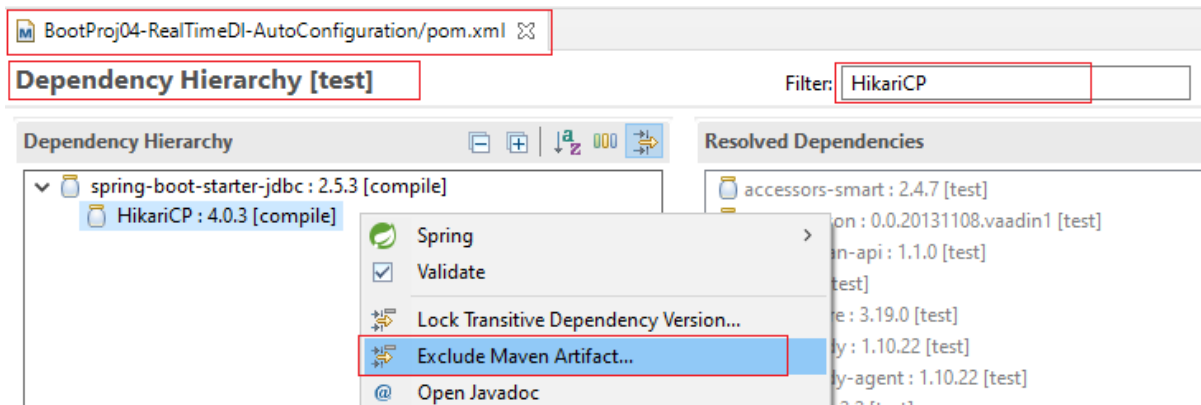
Q. How can Spring boot 2.x app working with Apache DBCP2 DataSource?

Ans.

Step 1: Exclude HikariCP jar file from list of dependent jar files (transitive dependents) that comes for "spring-boot-starter-jdbc "

Go to Dependency Hierarchy of pom.xml then search HikariCP in Filter

then right click on HikariCP, now click on Exclude Maven Artifact then it will exclude automatically.



Step 2: Now add Apache DBCP 2 jar dependency in dependencies tag of pom.xml file.

[pom.xml](#)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
  <exclusions>
    <exclusion>
      <groupId>com.zaxxer</groupId>
      <artifactId>HikariCP</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
</dependency>
```

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.8.0</version>
</dependency>
```

Step 3: Run the application

- ✚ Spring Boot 2.x (tested in Spring Boot 2.5) DataSource Algorithm towards AutoConfiguration is,
 - a. HikariCP (Default part of spring-boot-starter-jdbc jar file)
 - b. Tomcat CP (If HikariCP jar is not available and Tomcat-JDBC jar is placed)
 - c. Apache DBCP2 (If HikariCP, TomcatCP jars not available and Apache-DBCP2 jar is placed)
 - d. Oracle UCP (If HikariCP, TomcatCP, Apache DBCP2 jars not available and Oracle UCP jar is placed)

Note:

- ✓ Tomcat CP can be used as standalone connection pool (in standalone applications) and also Server Managed JDBC connection pool (in web applications).
- ✓ Oracle UCP DataSource can interact any DB s/w based on JDBC properties (driver class name, URL, DB user, DB password) that we are passing from application.properties file.

- ✚ In Spring Boot 1.x, the DataSource Algorithm was
 - a. Tomcat CP
 - b. HikariCP
 - c. Apache DBCP2
 - d. Apache DBCP

- ✚ To break this priority order/ algorithm to make our choice DataSource class directly participating Autoconfiguration then use "spring.datasource.type" property in application.properties file.

application.properties

```
#To use ApacheDBCP2
spring.datasource.type=org.apache.commons.dbcp2.BasicDataSource

#To use Oracle UCP
#spring.datasource.type=oracle.ucp.jdbc.PoolDataSource

# To use C3PO (try with Spring Boot <=2.2/2.3), will raise error in Spring
Boot 2.4 versions)
#spring.datasource.type=com.mchange.v2.c3pO.ComboPooledDataSource

#At a time, you can take only one DataSource collection type.
```

Note: This feature to work with new JDBC connection pools given by different third parties or user-defined JDBC connection pool because they are not part of Basic DataSource Configuration algorithm.

- ✚ To work with server Managed JDBC connection pool in Spring Boot web app or Distributed app (generally Spring Boot MVC/ Spring Boot Rest)

application.properties

```
# Server Managed JDBC con pool
spring.datasource.jndi-name=DsJndi

#Jndi name of DataSource object refer that is placed Jndi registry
```

Q. Can we disable AutoConfiguration of certain Spring beans even though starters are added?

Ans. Yes, we can do by using exclude param of @SpringBootApplication

BootProj04RealTimeDiAutoConfigurationApplication.java

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class,
JdbcTemplateAutoConfiguration.class})
public class BootProj04RealTimeDiAutoConfigurationApplication {
}
```

In the above situation we can use @Bean methods to create your choice classes objects and to make them Spring beans either in @Configuration class or in @SpringBootApplication class.

application.properties

```
#DataSource inputs
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
```

Note:

- ✓ And you have the required DataSource connection pool jar dependency in your pom.xml file.
- ✓ IoC container created special internal object (Environment) holding the

messages collected from the application.properties files and other user-defined properties files also.

PersistenceConfig.java

```
package com.sahu.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.env.Environment;

import com.mchange.v2.c3p0.ComboPooledDataSource;

@Configuration
public class PersistenceConfig {

    @Autowired
    private Environment env;
    @Bean
    public ComboPooledDataSource createDS() throws Exception {
        System.out.println("PersistenceConfig.createDS()");
        ComboPooledDataSource c3PODS = new
        ComboPooledDataSource();

        c3PODS.setDriverClass(env.getProperty("spring.datasource.driver-
        class-name"));
        c3PODS.setJdbcUrl(env.getProperty("spring.datasource.url"));

        c3PODS.setUser(env.getProperty("spring.datasource.username"));

        c3PODS.setPassword(env.getProperty("spring.datasource.password"))
;
        return c3PODS;
    }
}
```

Q. When should we use @Bean methods by disabling AutoConfiguration on Spring beans?

Ans. If you are, not happy with the classes that are participating AutoConfiguration and you want to use our choice class as Spring bean, then go for the @Bean methods like above.

Example scenario 1: In Spring Boot 2.4/ 2.5 there is no provision to work with

other DS algorithm connection pools related DataSource classes by specifying DataSource class name is "spring.datasource.type" property. In that situation disable AutoConfiguration on DataSource and go for @Bean methods in @Configuration class (Specially to use C3P0, Proxool, ViburCP and etc. JDBC connection pool).

Q. If DataSource is coming through AutoConfiguration and also configured using @Bean method then what DataSource object will be used in the execution of the application?

Ans. The Explicitly configuration @Bean method-based DS will be taken.

Q. What is the industry standard to pick up and use DataSource/ JDBC connection pool in Spring Boot Application?

Ans.

- If the application is standalone application, then prefer working with AutoConfiguration based HikariCP given multiple starters like spring-boot-starter-jdbc and spring -boot-starter-data-jpa and etc.
- If the application is Spring Boot MVC application/ Spring Boot Rest then prefer working underlying Server Managed JDBC connection pool like Tomcat managed JDBC connection pool, Wildfly Managed JDBC connection pool and etc. (or) use AutoConfiguration based HikariCP.

Note: Only in exceptional cases we go for other than HikariCP/ Server Managed JDBC connection pool.

Spring Boot Starter Parent

- ✚ The maven pom.xml of Spring Boot project contains multiple details
 - Parent starter/ project details (Maven inheritance)
 - Current project details
 - Properties overriding parent project details (For main jar files)
 - Dependencies (For dependent jar files)
 - Maven build plugins
- ✚ It is ready made Spring Boot project/ starter that is placed in maven central repository having multiple common things required for every new Spring Boot project created by the programmers. It is useful to reduce burden on the programmer towards configuring common things.
- ✚ spring-boot-starter-parent takes care of the following common things
 - a. Dependencies version based on Spring Boot parent version (jar version)

- b. Common properties
- c. Common maven builds plugins.

Dependencies version based on Spring Boot parent version:

- When we select Spring Boot starters while creating Spring Boot project using any IDE, we get those starters info in dependencies tag of pom.xml file but we get them without <version> tag because their version is decided based on spring-boot-starter-parent version.
- This helps to add the jars having balance and compatibility towards their versions.

pom.xml

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.5.3</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- If want to change version of specific Spring Boot Starter then we can specify <version> for that spring boot starter dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <version>2.5.3</version>
</dependency>
```
- In order to change the versions of transitive dependencies (child jar files) that given various spring boot starters we can use <dependency-

management> tag as shown below [Only these child jar files versions will be changed].

pom.xml

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>c3p0</groupId>
      <artifactId>c3p0</artifactId>
      <version>0.9.1.2</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <version>2.5.3</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Common properties:

- spring-boot-start-parent's pom.xml is giving multiple common properties like Java version, source encoding and etc.
- If want override these properties or if want to add new properties then we can use <properties> section of our project's pom.xml

```
<properties>
  <java.version>15</java.version>
</properties>
```

Common maven builds plugins:

- Actually, to run Spring Boot project in maven and to pack Spring Boot application into war/ jar file, to generate some reports we need multiple maven plugins like maven-surefire-plugin, maven-site-plugin and etc. but we do not add them in our project's pom.xml because they are inherited from spring-boot-starter-parent project.

@Value Annotation

- It is Spring supplied data annotation.
- It is given for 3 operations.
 - a. To inject hard coded value to Spring bean property

```

@Component
public class DBOperations {
    @Value ("system")
    private String dbuser;
}

```

- b. To inject values gathered from properties files

application.properties

db.user=system

```

@Component
public class DBOperations {
    @Value ("${db.user}")
    private String dbuser;
}

```

`${db.user}`: key of the properties file

- c. To write SPEL (Spring Expression Language) expression performing arithmetic and logical operations towards injecting values.

```

@Component("bill")
public class BillGeneration {
    @Value ("#{info.idlyPrice + info.dosaPrice}")
    private float total;
}

```

```

@Component("info")
public class ItemsInfo {
    @Value ("100.0")
    private float idlyPrice;
    @Value ("200.0")
    private float dosaPrice;
}

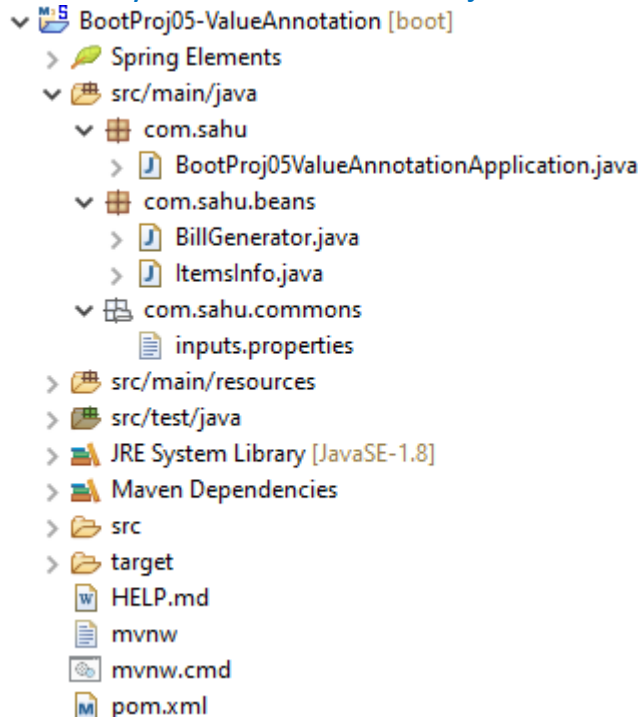
```

Note:

- ✓ To configure user-defined properties file we can `@PropertySource (-)` annotation.
- ✓ We can add only pre-defined properties or only user-defined properties or both in `application.properties`.
- ✓ The values collected from properties files/ yml files, system properties and environment variables like `PATH` will be first stored into the IoC

container and created internal Environment class object as key-value pairs and later will be injected to specified bean properties based key name matching in @Value annotations.

Directory Structure of BootProj05-ValueAnnotation:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also no starter project required.
- Then use the following code with in their respective file.

ItemsInfo.java

```
package com.sahu.beans;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.stereotype.Component;

@Component("info")
@PropertySource("com/sahu/commons/inputs.properties")
public class ItemsInfo {
    @Value("${idly.price}")
    public float idlyPrice;
    @Value("${dosa.price}")
```



```

    public float dosaPrice;
    @Value("${poha.price}")
    public float pohaPrice;

    @Override
    public String toString() {
        return "ItemsInfo [idlyPrice=" + idlyPrice + ", dosaPrice=" +
dosaPrice + ", pohaPrice=" + pohaPrice + "]";
    }
}

```

BillGenerator.java

```

package com.sahu.beans;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("bill")
public class BillGenerator {
    @Value("#{info.dosaPrice+info.idlyPrice+info.pohaPrice}")
    private float total;
    @Value("Paradise")
    private String hotelName;
    @Autowired
    private ItemsInfo items;

    @Override
    public String toString() {
        return "BillGenerator [total=" + total + ", hotelName=" +
hotelName + ", items=" + items + "]";
    }
}

```

input.properties

```

#User-defined message
dosa.price=300
idly.price=200
poha.price=100

```

BootProj05ValueAnnotationApplication.java

```
package com.sahu;

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

import com.sahu.beans.BillGenerator;

@SpringBootApplication
public class BootProj05ValueAnnotationApplication {

    public static void main(String[] args) {
        //Get IoC container
        ApplicationContext ctx=
SpringApplication.run(BootProj05ValueAnnotationApplication.class, args);
        //Get target Spring bean class
        BillGenerator bill = ctx.getBean("bill", BillGenerator.class);
        System.out.println(bill);
        //Close container
        ((ConfigurableApplicationContext) ctx).close();
    }
}
```

Note: application.properties file data/ values will override the data/ values of user-defined properties file, if both properties files are having same keys with different values.

Q. Can we change name or location of application.properties/ application.yml?

Ans. Yes, we can change but it must be configured explicitly using @PropertySource

Some Important points:

- We can pass input values to Spring bean properties from different places like DB s/w, HTML or JSP forms, properties file/ yml file and etc.
- By default, every Spring Boot application contains 3 files in the setup given by STS plugin as part of starter project
 - a. Main class/ Configuration and client class/ Starter class (class with @SpringBootApplication + main (-))

- b. application.properties/ yml (src/main/resources folder)
- c. pom.xml/ build.gradle (build file to place build configurations)

Note: Also gives lots of dependencies (jar files) as maven libraries.

- We need not to configure application.properties/ yml using `@PropertySource` because it part Spring Boot application echo system (SpringApplication.run(-) internally locates and recognizes as part of application startup activities).
- Any other user-defined properties placed any location must be configure explicitly using `@PropertySource` annotation.
- In application.properties/ yml file 99% we work with readymade/ pre-defined keys very rarely we place user-defined keys.
- application.properties/ yml file contains
 - DataSource configurations
 - Spring Boot banner configurations
 - Spring Boot log messages configurations
 - Spring Boot web application configurations
 - Spring security configurations
 - Spring batch configurations
 - Spring actuator configurations
 - Spring mail configurations
 - InMemory DB configurations
 - Messaging configurations
 - Zuul proxy configurations
 - Gateway's configurations
 - and etc.
- These pre-defined keys can be collected from [\[All key properties\]](#).
- These pre-defined keys based messages we can place both in application.properties/ yml and also in user-defined additional properties/ yml files.
- If we change the name or location or extension of application.properties/ yml file then it must be configure explicitly using `@PropertySource` annotation.
- If the key place holder (`${....}`) of `@Value` annotation is not matching with the keys available in all the configured properties files/ yml files through Environment object then we get [java.lang.IllegalArgumentException](#): Could not resolve placeholder 'dosa.price' in value "\${dosa.price}".

- We can inject values Spring bean properties from properties/ yml files we can use two types annotations
 - a. `@Value` (given by Spring and can be used Spring Boot)
 - To inject each value to each Spring bean property separately.
 - Does not support bulk injection (To inject 20 values of properties/ yml file to 20 Spring bean properties we need to use `@Value` annotation for 20 times).
 - b. `@ConfigurationProperties` (given by Spring Boot and cannot be used in Spring applications)
 - Supports bulk injection (i.e., we need to write `@ConfigurautinProperties` only 1 time on the top of Spring bean class).
 - All keys in properties file must have same prefix and Spring bean property names must be there in keys as last words (matching with each other).
 - Taking common prefix work for keys of properties/ yml file in mandatory.

Example application

application.properties

```
org.sahu.name=NareshIT
org.sahu.type=IT
org.sahu.location=Hyd
```

Using @Value

```
@Component ("company")
public class CompanyDetails {
    @Value("${org.nit.name}")
    private String title;
    @Value("${org.nit.type}")
    private String type;
    @Value("${org.nit.location}")
    private String addr;

    //toString ()
    .....
}
```

Using @ConfigurationProperties

```
@Component ("company")
@Data
@ConfigurationPropertis(prefix="
org.sahu")
public class CompanyDetails {
    private String name;
    private String type;
    private String location;

    //toString ()
    .....
}
```

To implement place the below code in their respective files in above project.

CompanyDetails.java

```
package com.sahu.beans;

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

import lombok.Data;

@Component("company")
@ConfigurationProperties(prefix = "org.sahu")
@Data
public class CompanyDetails {
    private String name;
    private String type;
    private String location;

    @Override
    public String toString() {
        return "CompanyDetails [name=" + name + ", type=" + type + ",
location=" + location + "];"
    }
}
```

CompanyDetails1.java

```
package com.sahu.beans;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("company1")
public class CompanyDetails1 {
    @Value("${org.sahu.name}")
    private String title;
    @Value("${org.sahu.type}")
    private String type;
    @Value("${org.sahu.location}")
    private String addr;

    @Override
    public String toString() {
```

```
        return "CompanyDetails [title=" + title + ", type=" + type + ",  
        addr=" + addr + "];  
    }  
}
```

application.properties

```
#User-defined message  
org.sahu.name=NareshIT  
org.sahu.type=IT  
org.sahu.location=Hyd
```

BootProj05.java

```
package com.sahu;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ConfigurableApplicationContext;  
  
import com.sahu.beans.CompanyDetails;  
import com.sahu.beans.CompanyDetails1;  
  
@SpringBootApplication  
public class BootProj05 {  
    public static void main(String[] args) {  
        //Get IoC container  
        ApplicationContext ctx=  
SpringApplication.run(BootProj05.class, args);  
        //Get target Spring bean class  
        CompanyDetails company = ctx.getBean("company",  
CompanyDetails.class);  
        System.out.println(company);  
        System.out.println("-----");  
        CompanyDetails1 company1 = ctx.getBean("company1",  
CompanyDetails1.class);  
        System.out.println(company1);  
        //Close container  
        ((ConfigurableApplicationContext) ctx).close();  
    }  
}
```

Q. What is the difference between @Value and @ConfigurationProperties annotations?

Ans.

@Value	@ConfigurationProperties
a. Given by Spring f/w so, it can be used in both Spring application and Spring Boot application.	a. Given by Spring Boot f/w so, it can be used only in Spring Boot applications.
b. Supports single value injection to Spring bean property.	b. Supports bulk injection to Spring bean properties.
c. Can perform field level injection i.e., setters and getters not required in Spring bean class.	c. Performs setter injection internally so; setters and getters are mandatory in Spring bean class.
d. Supports SPEL expression while injecting the values.	d) Does not support SPEL expressions (because there is no place to write SPEL expressions).
e. Common prefix for all keys in properties/ yml file is not required.	e. Required.
f. Keys in properties file and Spring bean property name need not to match.	f. Spring bean property names and the last word in keys (after matching prefix) must be matched otherwise injection on that Spring bean property will be ignored.
g. This field level annotation (We generally use).	g. This is class level, method level annotation (@Bean Method).
h. If specified key is not found in the properties file, then we will get IllegalArgumentException.	h. If matching key is not found then injection will be ignored on Spring bean property.

Note: While working @ConfigurationProperties we can takes prefixes in the properties file with respect to each Spring bean class.

application.properties

my.app.title=NIT

my.app.location=Hyd

org.sahu.name=NareshIt

org.sahu.addrs=hyd

```
@Component("comp1")
@Data
@ConfigurationProperties(prefix="my.app")
public class Company1 {
    private String title;
    private String location;
}
```

```

@Component("comp2")
@Data
@ConfigurationProperties(prefix="org.nit")
public class Company2 {
    private String name;
    private String addr;
}

```

Note: Both @Value and @ConfigurationProperties injects values to Spring bean property though IoC container created InMemory Environment object.

Q. If we try inject two different values to same Spring bean property using both @Value (Field level) and @ConfigurationProperties annotations?

Ans. Since @ConfigurationProperties performs setter injection so it overrides the field level injection value given by @Value annotation.

Injecting values to different types of Spring bean properties from properties file

- Different types Spring bean properties means Simple, Array, List, Set, Map, HAS-A property.
- It is recommended to add the following dependency in pom.xml while working with @ConfigurationProperties annotation to get META DATA (more info/ more details) about Spring bean properties.
- Simply we get suggestions/ hint box in application.properties towards user-defined Spring bean properties.

```

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

```

you can add this through suggestion

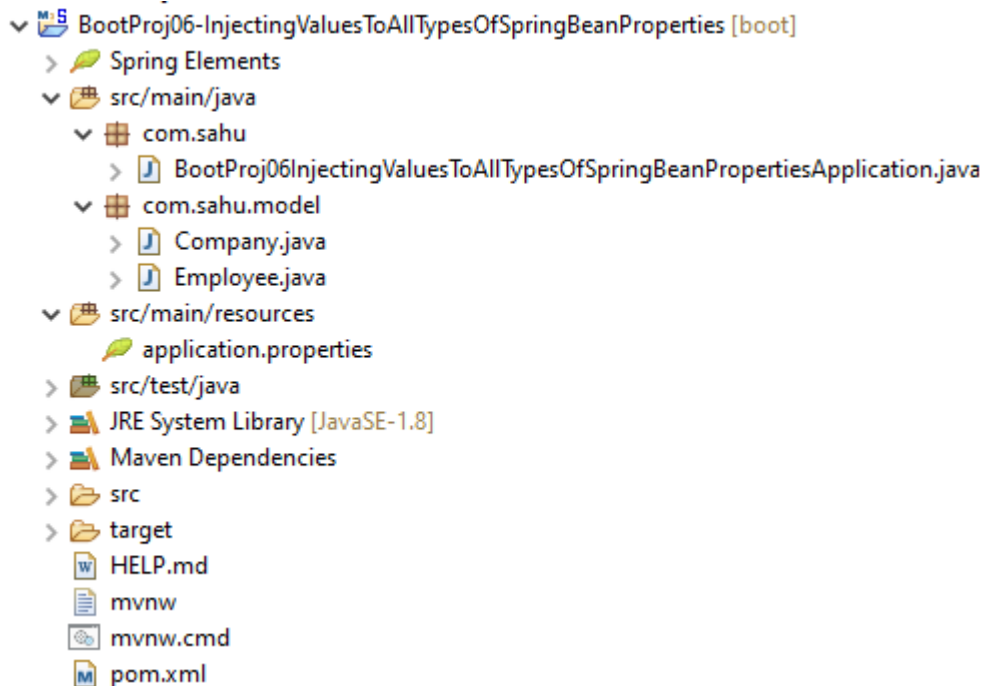
- The allowed or recommended special characters in keys of properties are ".", "-", "[", "]"

Directory Structure of

BootProj06-InjectingValuesToAllTypesOfSpringBeanProperties:

- Develop the below directory structure using Spring Starter Project option and create the package and classes also Lombok API starter project is required.

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



application.properties

```
#Simple properties - prefix.prop=value
emp.info.id=1001
emp.info.name=Rajesh

#Array type property - prefix.prop[index]=value
emp.info.nick-names[0]=China
emp.info.nick-names[1]=Kana
emp.info.nick-names[2]=Munna

#List type property - prefix.prop[index]=value
emp.info.team-members[0]=Ravi
emp.info.team-members[1]=Srinu
emp.info.team-members[2]=Ravi

#Set type property - prefix.prop[index]=value
emp.info.phone-numbers[0]=9999999999
emp.info.phone-numbers[1]=8888888888
emp.info.phone-numbers[2]=7777777777

#Map type property - prefix.prop.key=value
emp.info.id-details.aadharNo=3455678790976
emp.info.id-details.panNo=3RTY697X
```

emp.info.id-details.voterId=847364345

#Has-A property - prefix.prop.subprop=value
emp.info.comp-details.title=HCL technologies
emp.info.comp-details.location=Hyderbad
emp.info.comp-details.size=200

Company.java

```
package com.sahu.model;

import org.springframework.stereotype.Component;

import lombok.Data;

@Data
@Component("company")
public class Company {
    private String title;
    private String location;
    private int size;
}
```

Employee.java

```
package com.sahu.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("employee")
@ConfigurationProperties(prefix = "emp.info")
public class Employee {
    private long id;
    private String name;
```

```

    private String[] nickNames;
    private List<String> teamMembers;
    private Set<Long> phoneNumbers;
    private Map<String, Object> idDetails;
    private Company compDetails;
}

```

BootProj06InjectingValuesToAllTypesOfSpringBeanPropertiesApplication.java

```

package com.sahu;

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

import com.sahu.model.Employee;

@SpringBootApplication
public class
BootProj06InjectingValuesToAllTypesOfSpringBeanPropertiesApplication {

    public static void main(String[] args) {
        //Get IoC container
        ApplicationContext ctx =
SpringApplication.run(BootProj06InjectingValuesToAllTypesOfSpringBeanPr
opertiesApplication.class, args);
        //Get Spring bean class object
        Employee emp = ctx.getBean("employee", Employee.class);
        System.out.println(emp);
        //close IoC container
        ((ConfigurableApplicationContext) ctx).close();
    }
}

```

Note: While working with array, list, set type bean properties we must provide sequential index to elements

```

#Set type property- <prefix>.<prop>[index]=value
emp.info.phoneNumbers[0]=9999999999
emp.info.phone-numbers[1]=8888888888

```

```
emp.info.phone-numbers[2]=7777777777  
emp.info.phone-numbers[4]=6677777777
```

// gives error because indexing not sequential

Property: emp.inFo.phone-numbers[4]

Value: 667777777777

Origin: class path resource [application.properties] - 19:27

Reason: The elements [emp.info.phone-numbers[4]] were left unbound.

- The symbol in each key of properties file represents one level or node or key or sub property that depends on how and where we are using them.

YML/ YAML

- ✚ Yet Another Markup language.
- ✚ YAMLing language.
- ✚ Yiant Markup language
- ✚ The file can have either yml or yaml extension.
- ✚ The biggest limitation of properties files is the nodes/ levels will be repeated in multiple keys especially while working with common prefix concept, collections, HAS-A properties to support bulk injection using @ConfigurationProperties.
- ✚ Spring framework is not supporting yml only Spring Boot is supporting yml.

application.properties

```
emp.info.name=raja  
emp.info.id=1001  
emp.info.location=hyd
```

application.yml

```
emp:  
  info:  
    name: raja  
    id: 1001  
    location: hyd
```

- ✚ Spring Boot application internally converts every yml content properties content before using the content.
- ✚ To convert yml content to properties content and to parse, process yml documents Spring Boot internally use snakeyaml API (snakeyaml-<ver>.jar)

While writing yml files from properties file:

- a. Same nodes/ levels in the keys should not be repeated.
- b. Replace "." each node/ level with ":" symbol and write new node in the next line having proper indentation (minimum single space is required).

- c. Replace “=” symbol with “:” before placing value having minimum single space.
- d. To place array/ list/ set elements use “-” (hyphen).
- e. Take map collection keys and "HAS-A" property sub keys as the new nodes/ levels.
- f. Use # symbol for commenting.

Directory Structure of

BootProj07-InjectingValuesToAllTypesOfSpringBeanPropertiesYML:

- Copy and paste the previous project and rename that project.
- Delete the application.properties file and create application.yml at that location.
- And place the following code with their respective files.

application.yml

```
#Using YML
emp:
  info:
    #Simple Properties
    id: 1001
    name: raja
    #Array
    nickNames:
      - Kana
      - Muna
      - Ravi
    teamMembers:
      - Ravi
      - Ramesh
      - Harish
    phoneNumbers:
      - 9999999999
      - 8888888888
      - 7777777777
    #Map
    idDetails:
      aadharNo: 6987456334
      panNo: JA643HY
      voterId: 987465445
```

```
#HAS-A property
compDetails:
  title: HCL
  location: Hyderabad
  size: 2000
```

Q. Can we place values in properties file/ yml file in a single line for array/ list/ set collection type properties?

Ans. Yes, possible

application.properties

```
emp.info.nick-names[0]=chinna
emp.info.nick-names[1]=kanna
emp.info.nick-names[2]=munna
```

is equal to

```
emp.info.nick-names=chinna,kanna,munna (In line formatting)
```

application.yml

```
emp:
  info:
    nick-names:
      - chinna
      - kanna
      - munna
```

is equal to

```
emp:
  info:
    nick-names: [chinna,kanna,munna]
(In line formatting)
```

- ✚ We can take user-defined yml files, but we must develop one PropertySourceFactory class to process that yml file, while configuration that user-defined yml file using @PropertySource we should specify yml location and that factory class.

```
@Configuration
@ConfigurationProperties(prefix = "emp.info")
@PropertySource(value = "classpath:foo.yml", factory =
YamlPropertySourceFactory.class)
public class YamlFooProperties {
    .....
    .....
    .....
}
```

For more information refer: [\[Visit\]](#)

Q. If we place both application.properties and application.yml having same keys and different values can we tell me what happens?

Ans. The values given application.properties will override the values given in application.yml

Some Examples:

application.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
```

application.yml

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

application.properties

```
my.nit.name=raja
my.nit.id=1001
my.nit.one.profile=SE
my.nit.one.exp=20
my.nit.two.qlfy=Engineer
```

application.yml

```
my:
  nit:
    name: raja
    id: 1001
    one:
      profile: SE
      exp: 20
    two:
      qlfy: Engineer
```

Note:

- ✓ The nodes/ level in the keys of properties file or yml file are not case-sensitive.

- ✓ Once we have properties file in any eclipse Spring/ Spring Boot project, then it can be converted into yml easily using wizard supplied by STS plugin right click on properties file --> convert to yaml file.

Q. What is the difference between Properties file and YML file?

Ans.

Properties File	YML File
a. There is no specification providing rules and guidelines to develop properties file's it is just keys=values.	a. There is specification providing rules and guidelines to develop the yml file (www.yml.org).
b. Can be used only in java.	b. Can be used java, python, ruby, groovy and etc.
c. No way related to JSON format.	c. Super set of JSON.
d. Can be used in both Spring f/w and Spring Boot.	d. Cannot used in Spring f/w i.e., supported only in Spring Boot.
e. Nodes/ levels in the keys may have duplicates.	e. Same nodes/ levels will not be repeated (no duplicates).
f. It is not hierarchal data.	f. It is hierarchal data.
g. Custom properties file can be configured in Spring Boot App directly by using @PropertySource and no PropertySourceFactory class is required.	g. Custom yml file can be configured in Spring Boot App by using @PropertySource and by developing, specifying PropertySourceFactory class is required.
h. While working with profiles in Spring/ Spring Boot we cannot place multiple profiles in single properties file.	h. We can place multiple profiles in single yml file having separation with "---".
i. Spring or Spring Boot App directly loads and reads the properties content.	i. Every yml file will be converted to properties file before loading and reading.
j. Use properties file when number of keys are less and the nodes/ levels in keys are not repeating.	j. Use yml file when number of are keys are more and the nodes/ levels in keys are repeating.

Profiles in Spring Boot

- Both Spring and Spring Boot support profiles.
- In all approaches of Spring app development, we can use profiles
 - XML driven configuration,
 - Annotation driven configuration
 - 100% code driven configuration,
 - Spring Boot

- ✚ Environment means the setup the required for executing the application or project, it contains
compiled code + servers + DB software + Jar files + properties/ yml and etc.
- ✚ From the development to production of the project we come across the multiple environments where code needs to be executed from different purposes. This environment is also called as profile.
 - a. Dev environment/ Dev profile
 - b. Test or QA environment/ Test or QA profile
 - c. UAT environment/ UAT profile
 - d. Prod environment/ Prod profile
 - and etc.

Dev env.

- MySQL DB software
- Apache DBCP DS
- Tomcat server
- and etc.

Test env.

- MySQL DB software
- Oracle UCP DS
- Tomcat server
- and etc.

Software
company side
(may be Local/
cloud)

UAT env.

- Oracle DB software
- C3p0 DS
- Glassfish server
- and etc.

Prod env.

- MySQL DB software
- HikariCP DS
- Wildfly server
- and etc.

All client side or
Organization
side (May be
local/ cloud)

Note: The testing on released project done by client organization people and its respective is called UAT.

In our Spring App/ project:

- To make stereotype annotation-based spring class participating only on certain profile activation use @Profile on the top of Spring bean class like @Profile("dev") or @Profile({"dev","test"}).

Note: If no @Profile is specified the spring bean is available for all the profiles.

```
@Repository("mysqlEmpDAO")
@Profile({"dev", "test"})
public class MysqlEmployeeDAOImpl implements IEmployeeDAO {
```

```

        .....
        .....
    }

    @Repository("oraEmpDAO")
    @Profile({"uat" , "prod"})
    public class OracleEmployeeDAOImpl implements IEmployeeDAO {
        .....
        .....
    }

```

- To make @Bean method of @Configuration class returned java class object related Spring bean, working for certain profile then @Profile annotations on the top of @Bean methods.

```

@Configuration
public class PersistenceConfig {
    @Bean(name="dsDBCP2")
    @Profile("dev")
    public Datasource createDsUsingDBCP2() {
        .....
        .....
    }
    @Bean(name="dsOraUCP")
    @Profile("test")
    public Datasource createDsUsingOracleUCP() {
        .....
        .....
    }
}

```

- To keep application.properties/ application.yml information specific to one profile then we create separate properties file for that profile as shown below

[application.properties](#)

application-dev.properties/yml

application-uat.properties/yml

application-prod.properties/yml application-<profile>.properties/yml

Note: While working with yml we have option of placing multiple profiles in single yml file (We don't need any factory class, though we are working with multiple yml files in profiles environment).

wrong file names while working with profiles

dev-application.properites/yml

test-application.properites/yml

How to activate specific Profile

a. In application.properties/ yml file

spring.profiles.active=prod

(In both web application and standalone application)

Note: If any given profile related properties file is not available then it will use application.properties file as fallback properties file.

b. Using system property with the value

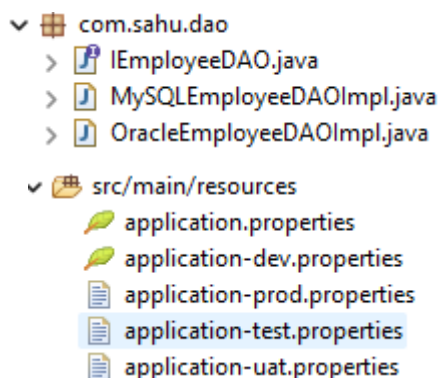
-Dspring.profiles.active=dev (valid)

-Dspring.profiles.active=application-dev.properties X (invalid)

(Only in standalone environment)

Directory Structure of BootProj08-RealTimeDI-AutoConfiguration-Profiles:

- Copy and paste the BootProj04-RealTimeDI-AutoConfiguration project and rename that project.
- Create the following file in their respective location.



```
com.sahu.dao
├── IEmployeeDAO.java
├── MySQLEmployeeDAOImpl.java
├── OracleEmployeeDAOImpl.java
└── src/main/resources
    ├── application.properties
    ├── application-dev.properties
    ├── application-prod.properties
    ├── application-test.properties
    └── application-uat.properties
```

- In the both DAOImpl class the code is same only change you will get from below. And to create the DAOs just need to copy and paste then rename the class and files names
- Delete the com.sahu.config package
- And place the following code with their respective files.

BootProj04RealTimeDiAutoConfigurationApplication.java

```
@SpringBootApplication(exclude = {DataSourceAutoConfiguration.class,
JdbcTemplateAutoConfiguration.class})
public class BootProj04RealTimeDiAutoConfigurationApplication {

    @Autowired
    private Environment env;

    @Bean(name = "cds")
    @Profile("uat")
    public ComboPooledDataSource createC3P0DS() throws Exception {
        ComboPooledDataSource comboPooledDataSource = new
        ComboPooledDataSource();

        comboPooledDataSource.setDriverClass(env.getProperty("spring.data
source.driver-class-name"));

        comboPooledDataSource.setJdbcUrl(env.getProperty("spring.datasou
rce.url"));

        comboPooledDataSource.setUser(env.getProperty("spring.datasourc
e.username"));

        comboPooledDataSource.setPassword(env.getProperty("spring.datas
ource.password"));
        return comboPooledDataSource;
    }
    .....
}
```

OracleEmployeeDAOImpl.java

```
@Repository("empDAO")
@Profile({"uat", "prod"})
public class OracleEmployeeDAOImpl implements IEmployeeDAO {
```

MySQLEmployeeDAOImpl.java

```
@Repository("empDAO")
@Profile({"dev", "test"})
public class MySQLEmployeeDAOImpl implements IEmployeeDAO {
```

application-dev.properties

```
#MySQL DB s/w with Apache DBCP2
#DataSource inputs
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///ntspbms714db
spring.datasource.username=root
spring.datasource.password=root

#To use Apache DBCP2
spring.datasource.type=org.apache.commons.dbcp2.BasicDataSource
```

application-prod.properties

```
#DataSource inputs
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-test.properties

```
#MySQL DB s/w with Oracle UCP
#DataSource inputs
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///ntspbms714db
spring.datasource.username=root
spring.datasource.password=root

#To use Oracle UCP
spring.datasource.type=oracle.ucp.jdbc.PoolDataSource
```

application-uat.properties

```
#Oracle DB s/w with C3P0
#DataSource inputs
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.properties

```
#Active Spring Profile  
spring.profiles.active=uat
```

pom.xml

```
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-jdbc</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>org.apache.commons</groupId>  
    <artifactId>commons-dbcp2</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>c3p0</groupId>  
    <artifactId>c3p0</artifactId>  
    <version>0.9.1.2</version>  
  </dependency>  
  <dependency>  
    <groupId>com.oracle.database.jdbc</groupId>  
    <artifactId>ojdbc8</artifactId>  
    <scope>runtime</scope>  
  </dependency>  
  <dependency>  
    <groupId>org.projectlombok</groupId>  
    <artifactId>lombok</artifactId>  
    <optional>true</optional>  
  </dependency>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-test</artifactId>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

Q. Can we activate multiple profiles at time?

Ans. Yes, that may lead to ambiguity problems. More over activating multiple profiles at a time are meaning less operation because our project can stay only in one environment/ profile at a time.

application.properties

```
#Active Spring Profile
spring.profiles.active=test,dev,prod
```

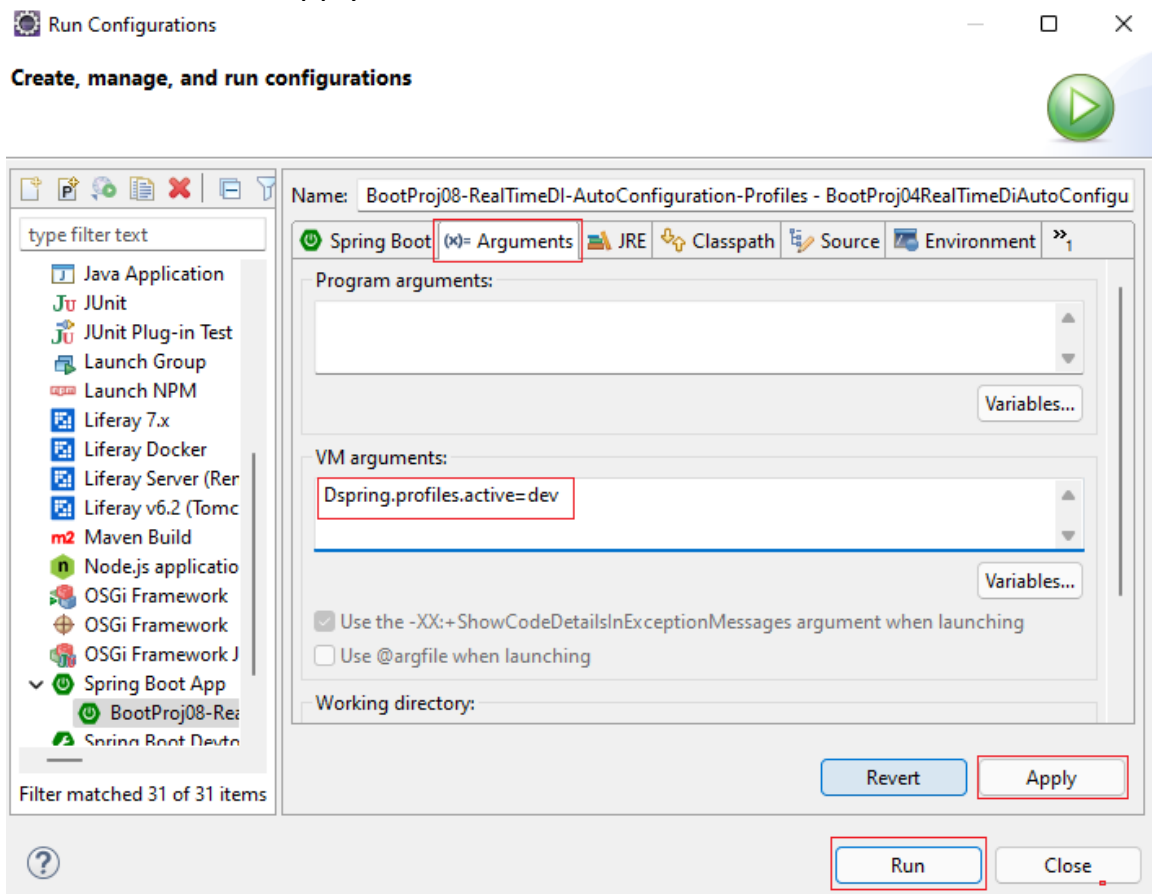
Q. How many ways are there to activate profiles in Spring Boot application?

Ans. There are 4 ways,

- Using application.properties/ yml (Best)
- Using system property (-Dspring.active.profiles)
- Using maven pom.xml
- Using programmatic approach (application.setAdditionalProfile(-))

Using System Property:

- Right click on Project then -> click on Run As -> Run Configurations then
- Click on Arguments the got to VM arguments the give the information then click on Apply then Run



Note:

- ✓ Program arguments means Command line arguments
- ✓ VM arguments means System properties

Using Programmatic approach:

- Since application.properties/ yml is not available in 100% code driven, annotation driven and XML configurations driven Spring applications development. So, we use the following code activate profile.

BootProj04RealTimeDiAutoConfigurationApplication.java

```
// Get IoC container
ApplicationContext ctx =
SpringApplication.run(BootProj04RealTimeDiAutoConfigurationApplication.
class, args);

//Get access to Environment object
ConfigurableEnvironment env = (ConfigurableEnvironment)
ctx.getEnvironment();
env.setActiveProfiles("uat");
```

- In Spring Boot application place the following code in runner/ main class.

BootProj04RealTimeDiAutoConfigurationApplication.java

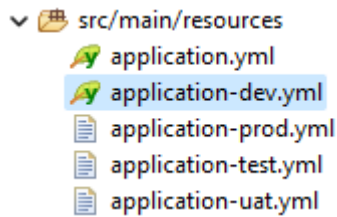
```
// Get SpringApplication object
SpringApplication application = new
SpringApplication(BootProj04RealTimeDiAutoConfigurationApplicatio
n.class);
application.setAdditionalProfiles("uat");

// Get IoC container
ApplicationContext ctx = application.run(args);
```

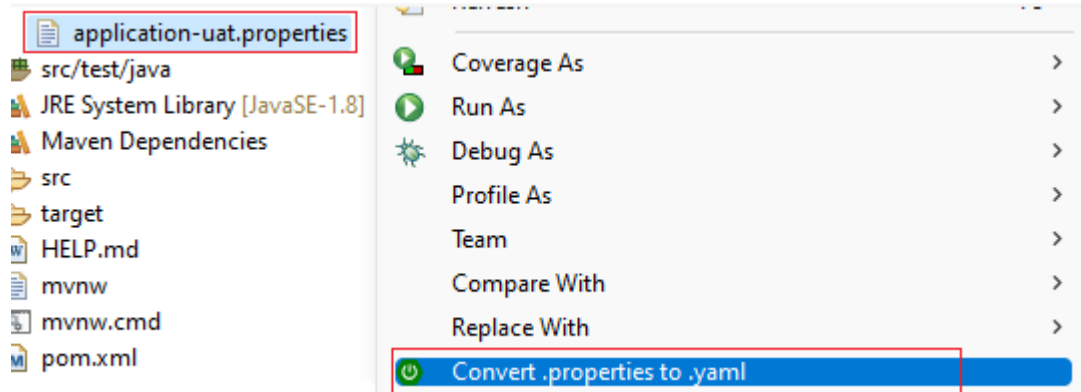
Using YML file:

Directory Structure of BootProj09-RealTimeDI-AutoConfiguration-Profiles-yml:

- Copy and paste the BootProj08-RealTimeDI-AutoConfiguration-Profiles project and rename that project.
- Now just convert property files to yml



- Right click on properties file then click on Convert .properties to .yaml



Note:

- ✓ While working with yaml, we can place multiple profiles info in single yaml file. We must have to use "---" as separator.
- ✓ snakeyaml API used by Spring Boot internally converts multiple profiles placed in single yaml file into multiple properties files by taking "---" symbol as delimiter.

Directory Structure of

BootProj10-RealTimeDI-AutoConfiguration-Profiles-singleyaml:

- Copy and paste the BootProj09-RealTimeDI-AutoConfiguration-Profiles-yaml project and rename that project.
- Remove all the yaml file except application.yaml
- And place the following code in their respective file.

application.yaml

```
spring:
  profiles:
    active: uat

---
#DEV
spring:
  datasource:
```

```
driver-class-name: com.mysql.cj.jdbc.Driver
password: root
type: org.apache.commons.dbcp2.BasicDataSource
url: jdbc:mysql:///ntspbms714db
username: root
```

#PROD

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

#TEST

```
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    type: 'oracle.ucp.jdbc.PoolDataSource '
    url: jdbc:mysql:///ntspbms714db
    username: root
```

#UAT

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

Q. What is fallback profile?

Ans. If active profile related properties/ yml file is not available in physically in our project then it looks for relevant required information from the application.properties/ yml file. So, we can say application.properties/ yml file as fallback file/ profile.

Q. If we activate different profiles using application.properties/ yml, system property and programmatic approach then what will happen?

Ans. There is possibility of getting ambiguity problem because multiple profiles are activated belonging to different environments.

Q. How can we change from one dependent Spring bean to another dependent Spring bean dynamically at run time without touching the source code of Spring beans?

Ans.

- a. Using <alias> tag of Spring bean configuration file in support with @Qualifier annotation.
- b. Using Spring profiles (Best).

Child Profiles or Profiles Include

- ✚ While taking application.properties file as fallback properties file having fallback profile information instead of writing repeated keys and values in application.properties which are already there in another profile specific properties file.
- ✚ We can include that profile name in application.properties file as shown below.

Directory Structure of

BootProj11-RealTimeDI-AutoConfiguration-ProfilesInclude:

- Copy and paste the BootProj08-RealTimeDI-AutoConfiguration-Profiles project and rename that project.
- And place the following code in their respective file.

application.yml

```
#Active Spring Profile
spring.profiles.active=staging
```

```
#Include multiple keys from other profile specific properties file
spring.profiles.include=prod
```

(This inclusion makes application.properties related fallback profile as child profile for prod profile.)

Note:

- ✓ We do not have application-staging.properties file so it will take application.properties as the fallback file, but in that file we are not keys

- and values directly rather we getting by including another profile (prod).
- ✓ We can include other profiles info only in application.properties i.e., not possible in profile specific properties files.

Q. What is the meaning of @Profile("default")?

Ans.

- If no active profile is specified then all the beans of "default" profile will be instantiated and used.
- It is recommended to have complete setup of all Spring beans to execute in "default" profile i.e., when no active profile is specified.
- We can make certain Spring bean working for specific profiles and also for default Profile as shown below.

MySQLEmployeeDAOImpl.java

```
@Repository("empDAO")
@Profile({"dev", "test", "default"})
public class MySQLEmployeeDAOImpl implements IEmployeeDAO {
```

BootProj04RealTimeDiAutoConfigurationApplication.java

```
@Bean(name = "tcp")
@Profile("default")
public DataSource createTomcateCPDS() throws Exception {

    System.out.println("BootProj04RealTimeDiAutoConfigurationApplicat
ion.createTomcateCPDS()");
    DataSource tcpds = new DataSource();
    tcpds.setDriverClassName("com.mysql.cj.jdbc.Driver");
    tcpds.setUrl("jdbc:mysql:///ntspbms714db");
    tcpds.setUsername("root");
    tcpds.setPassword("root");
    return tcpds;
}
```

To make app running without active profiles we just need to comment.

application.yml

```
#Active Spring Profile
#spring.profiles.active=dev
```

Note: "default" is not profile name, it is the environment or setup of Spring/ Spring Boot that executes when no active profiles are specified.

Q. What is the difference b/w fallback profile and default profile and also child profile?

Ans.

- Fallback Profile: The profile that executes when requested active profile is not available.
- Default Profile: The profile that executes when there no activate profile in the application.
- Child Profile/ Profile Include: The Profile that is included in application.properties by specifying the profile name to reuse keys and values specific profile in application.properties indirectly as fallback profile.

Q. Can we keep multiple Spring beans of same category/ type (having common super class/ implementing interface) in default profile?

Ans. No, not possible. Conflicts/ ambiguity problems may come.

- Spring/ Spring Boot gives [org.springframework.beans.factory.NoUniqueBeanDefinitionException](#): no qualifying bean of 'javax.sql.DataSource' expected single matching bean but found 2: cds, tcp
- This error will come when more dependent of same type found to inject to target bean.
- Spring/ Spring boot gives [org.sf.beans.factory.NoSuchBeanException](#) if the given bean id Spring bean is not available.

Q. What is difference b/w not adding @Profile for Spring bean and adding @Profile("default") for Spring bean?

Ans.

- If @Profile is not added for Spring bean that will work all for profiles when profiles activated it will even work if no profiles are activated.
- @Profile("default") based Spring bean will work only when active profiles are not specified.
- In real projects we keep DS classes, DAO classes in specific profile or in default profile because persistence logic changes DB s/w to DB s/w because SQL queries are DB s/w dependent.
- In real projects, we do not keep controller, service classes in any profile because they are common for all profiles and not dependent to any DB s/w.

Runners in Spring Boot

- Runners' java classes cum Spring beans of Spring Boot app implementing XxxRunner (I) directly or indirectly.
- Every runner class contains run (-) dealing with onetime executing logics and those logics will execute when SpringApplication.run(-) is about to complete all its startup activities (right after creating ApplicationContext object and completing pre-instantiation and injections)
- The run (-) of every Runner class will be executed automatically by IoC container only for 1 time as part of application startup process that takes place in SpringApplication.run(-) method.

Q. What is the difference b/w Static block and Runner with respect to Spring Boot application?

Ans.

Static Block of Spring Bean	Spring Bean acting Runner class
a. Executes when IoC container loads this class for instantiation.	a. run (-) this class will be called automatically by SpringApplication.run(-) as part of application startup activities.
b. We cannot pass inputs (args) while working with static block.	b. We can get CMD Line args as inputs in run (-) method as parameter values.
c. Static block allows only static variables/ data to use.	c. run (-) of Runner classes can deal with both static and non-static variables/ data.
d. It is feature of java. So, can be used in Spring and Non-Spring environment.	d. Features of Spring Boot. So, can be used only in Spring Boot application to place one time executing logics.

Two types of Runners in Spring Boot:

- CommandLineRunner (I)
- ApplicationRunner (I)

Both these are two independent interfaces

Q. What is the difference between CommandLineRunner and ApplicationRunner?

Ans. Both the runners are giving run (-) as the callback method (because it will be called by underlying IoC container automatically). Both runner's run (-) methods get command line args (program args) as the parameter values but the way they get them is different.

org.springframework.boot.CommandLineRunner (I)

| -> public void run (String... args)

gives command line args as raw values (same as main (-) args []
or ...args)

org.springframework.boot.ApplicationRunner (I)

| -> public void run (ApplicationArguments args)

Allows to categories and get command line args value as
option args (start with ...) non-option args from
ApplicationArguments object

Methods of ApplicationArguments class:

All Methods	Instance Methods	Abstract Methods
Modifier and Type		Method and Description
boolean		containsOption(String name) Return whether the set of option arguments parsed from the arguments contains an option with the given name.
List<String>		getNonOptionArgs() Return the collection of non-option arguments parsed.
Set<String>		getOptionNames() Return the names of all option arguments.
List<String>		getOptionValues(String name) Return the collection of values associated with the arguments option having the given name.
String[]		getSourceArgs() Return the raw unprocessed arguments that were passed to the application.

- Both are Functional Interfaces (Java 8) because both of them contains only one method declaration i.e., run (-) method.
- The interface that contains only one abstract method declaration directly or indirectly is called functional interface.
- To mark interface as functional interface we use @FunctionalInterface.

Q. What is the use of Runners in Spring Boot apps?

Ans.

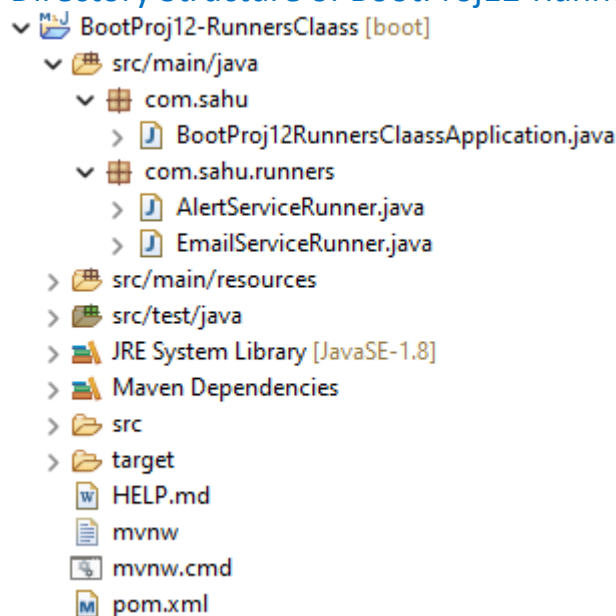
- We generally place initial logics or startup logics to execute in the run (-) method of Runner class.
- The logics are like,
 - Executing DB script and keeping DB tables with rows ready.
 - Performing some testing activities related to AutoConfiguration and its injections.
 - Logging activities related to AutoConfiguration and startup activities.
 - To send email or SMS alter messages to admins when important services are started during the startup process of Spring Boot

application, like start of server, initialization of SMS Gateway, Payment Gateway and etc.

- Though there are multiple modules in project, if u want to activate only certain modules, we take the support of runners.
- To create dummy/ admin email ids and accounts during the startup process automatically we take support of runners and etc.

Note: In real time projects the main class (configuration class cum main class that contains @SpringBootApplication) main (-) method contain only SpringApplication.run(-) method call. The remaining all logics like getting spring bean class object from IoC container, invoking business method and etc. will be done from the run (-) methods of runner classes through various injections.

Directory Structure of BootProj12-RunnersClass:



- Develop the above directory structure using Spring Starter Project option and create the package and classes also, no starter project required.
- Then place the following code with in their respective files.

BootProj12RunnersClaassApplication.java

```
package com.sahu;

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



```

@SpringBootApplication
public class BootProj12RunnersClaassApplication {

    public static void main(String[] args) {

        SpringApplication.run(BootProj12RunnersClaassApplication.class,
args);
    }

}

```

AlertServiceRunner.java

```

package com.sahu.runners;

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

@Component
public class AlertServiceRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("AlertServiceRunner.run()");
        for (String arg : args) {
            System.out.println(arg);
        }
    }

}

```

EmailServiceRunner.java

```

package com.sahu.runners;

import java.util.Arrays;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;

```

```

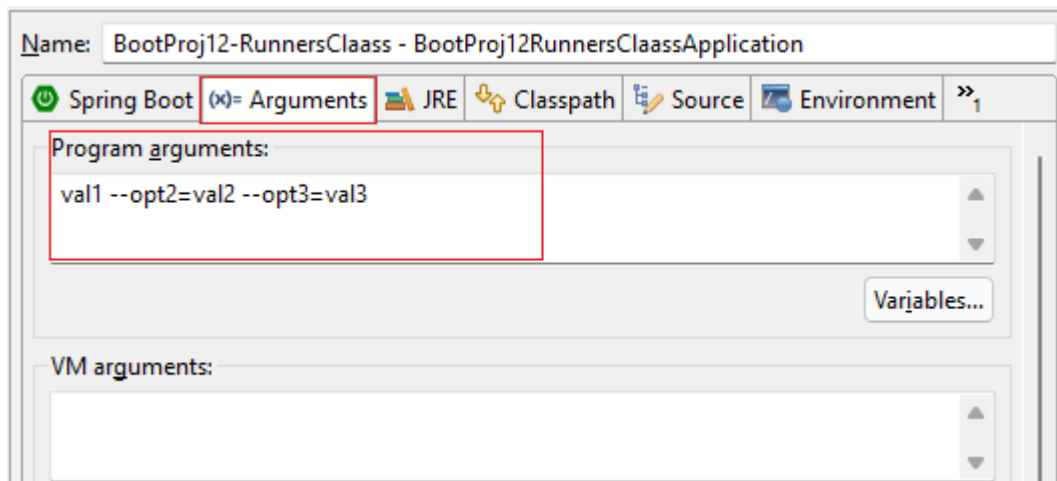
@Component
public class EmailServiceRunner implements ApplicationRunner {

    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("EmailServiceRunner.run()");
        System.out.println("Non-option args -
"+args.getNonOptionArgs());
        System.out.println("Option Names - "+args.getOptionNames());
        System.out.println("Option values -
"+args.getOptionValues(null));
        System.out.println("Source Args -
"+Arrays.toString(args.getSourceArgs()));
    }
}

```

While running the application provide command line args:

Run As -> Run Configuration... -> Arguments -> Give the Program arguments
Then click on Apply and Run



- --opt2=val2, --opt3=val3 these two are option args having value.
- val1 is non-option args (only value).
- Program args are command line args like args[0], args[1] and etc.
- VM args given to pass system properties application using -D option
 - Dspring.profiles.active=dev
 - Duname=raja
 - Dpremgensize=34mb

Note: Runners really useful in projects of type web applications and distributed applications which runs in production environment for long time without shutdown. In those applications if want to execute some startup logics or initialization logics then go for Runner Support.

Q. What is the difference b/w Starter and Runner?

Ans.

- Starters are jar files acting as libraries supporting Spring Boot's autoconfiguration activities by collecting inputs from application.properties/ yml.
- Runners are Spring beans having callback method (run (-)) and contains application startup or initial logics as discussed in above use-cases.

More about Runner:

- ✓ In one Spring Boot application we can place multiple Runners, if you do not want to execute specific runner then remove @Component on top of that runner class.
- ✓ If multiple Runners are placed in a Spring Boot application, first it will execute all ApplicationRunner and then it will execute all CommandLineRunner.
- ✓ By default, the multiple ApplicationRunner and CommandLineRunner execute in their alphabetic order (A-Z). [First A-Z alphabetic order with in ApplicationRunner classes, next A-Z alphabetic order with in CommandLineRunner classes].
- ✓ To provide our choice execution order to these multiple Runner classes we can use either @Order(<n>) [new] or Ordered (I) [old means legacy] implementation on Runner classes.
- ✓ Once @Order is specified having priority value or Ordered (I) is implemented having priority value then the Runners will execute according to the priority value without worrying about Runner type (ApplicationRunner or CommandLineRunner).
- ✓ High value indicates low priority.
- ✓ Low Indicates high priority
- ✓ If same priority value is given then they will go by alphabetic order of class names.

Note: The default priority value is Integer max Value (2147483647) (Integer.MAX_VALUE).

- Create more Runner classes and place the flowing code with their files.

- ▼ com.sahu.runners
 - > AlertServiceRunner.java
 - > EmailServiceRunner.java
 - > JdbcServiceRunner.java
 - > SecurityServiceRunner.java

JdbcServiceRunner.java

```
package com.sahu.runners;

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(1)
public class JdbcServiceRunner implements CommandLineRunner {

    @Override
    public void run(String... args) throws Exception {
        System.out.println("JdbcServiceRunner.run()");
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

SecurityServiceRunner.java

```
package com.sahu.runners;

import java.util.Arrays;

import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(15)
public class SecurityServiceRunner implements ApplicationRunner {

    @Override
```

```

    public void run(ApplicationArguments args) throws Exception {
        System.out.println("SecurityServiceRunner.run()");
        System.out.println("Non-option args -
"+args.getNonOptionArgs());
        System.out.println("Option Names - "+args.getOptionNames());
        System.out.println("Option values -
"+args.getOptionValues(null));
        System.out.println("Source Args -
"+Arrays.toString(args.getSourceArgs()));
    }
}

```

AlertServiceRunner.java

```

@Component
@Order(10)
public class AlertServiceRunner implements CommandLineRunner {

```

AlertServiceRunner.java

```

@Component
@Order(2)
public class EmailServiceRunner implements ApplicationRunner {

```

<u>Runner</u>	<u>@Order value</u>	<u>Runner</u>	<u>@Order value</u>
XRunner	10 (iv)	XRunner	10 (ii)
YRunner	-20 (i)	YRunner (AR)	<no value> (iii)
ZRunner	1 (ii)	ZRunner	1 (i)
ARunner	6 (iii)	ARunner (CR)	<no value> (iv)

<u>Runner</u>	<u>@Order value</u>	<u>Runner</u>	<u>@order value</u>
XRunner (AR)	10 (ii)	XRunner (CR)	10 (v)
Yrunner (CR)	10 (iii)	YRunner (AR)	-20 (i)
ZRunner (CR)	10 (iv)	ZRunner (AR)	<no value>(vii)
ARunner (AR)	10 (i)	ARunner (CR)	6 (iii)
		BRunner (CR)	6 (iv)
		CRunner (AR)	<no vlaue> (vi)
		DRunner (AR)	6 (ii)

Conclusion priority order:

- Low value to high value of @Order including negative values
 - [If multiple runners having same priority, then A-Z alphabetic Order first in ApplicationRunner then in CommandLineRunner].
- No @Order Runners
 - [A-Z alphabetic Order first in ApplicationRunner then in CommandLineRunners].

Ordered (I):

- We can also give the priority value by implementing Ordered (I) as shown below

AlertServiceRunner.java

```
package com.sahu.runners;

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

@Component
public class AlertServiceRunner implements CommandLineRunner, Ordered
{

    @Override
    public void run(String... args) throws Exception {
        System.out.println("AlertServiceRunner.run()");
        for (String arg : args) {
            System.out.println(arg);
        }
    }

    @Override
    public int getOrder() {
        return -10;
    }

}
```

Q. If we provide two different priority values using @Order and Ordered (I) for a Runner class then which will be taken?

Ans. Ordered (I)'s getOrder() method returned value will be taken as the priority value.

AlertServiceRunner.java

```
package com.sahu.runners;

import org.springframework.boot.CommandLineRunner;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;

@Component
@Order(-20)
public class AlertServiceRunner implements CommandLineRunner, Ordered
{

    @Override
    public void run(String... args) throws Exception {
        System.out.println("AlertServiceRunner.run()");
        for (String arg : args) {
            System.out.println(arg);
        }
    }

    @Override
    public int getOrder() {
        return 15;
    }
}
```

Before JDK 1.7:

```
interface Arithmetic {
    public int add (int a, int b);
}

class ArithmeticImpl implements Arithmetic {
    public int add (int a, int b) {
        return a + b;
    }
}
```

In client code/ test code: `Arithmetic ar = new ArithmeticImpl();`
`ar.add(20, 20);`

JDK 1.8 and above: (LAMDA Expressions)

```
interface Arithmetic {  
    public int add (int a, int b);  
}
```

Lambda expression for interface gives implementation class + object creation

```
Arithmetic ar = (int a, int b) -> {return a + b};  
ar.add(20, 30)
```

(or)

```
Arithmetic ar = (int a, int b) -> return a + b;  
    [method body {} optional for 1 statement]  
Arithmetic ar = (int a, int b) -> a + b;  
    [when {} are not their return statement is optional]  
Arithmetic ar = (a, b) -> a+b;  
    [Data types for params are optional]
```

Note:

- ✓ If there is a single param then “()” are optional.
- ✓ This LAMDA expressions-based interface programming is possible in only for Functional interface.

✚ In Spring Boot application both runners are functional interfaces, so we develop runners' logics directly in main class without taking separate classes.

BootProj12RunnersClassApplication.java

```
package com.sahu;  
  
import org.springframework.boot.ApplicationArguments;  
import org.springframework.boot.ApplicationRunner;  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
import org.springframework.context.annotation.Bean;  
import org.springframework.core.annotation.Order;  
  
@SpringBootApplication
```



```

public class BootProj12RunnersClaassApplication {

    @Bean
    @Order(-20)
    public CommandLineRunner createSchedulingRunner() {
        CommandLineRunner commandLineRunner = (String... args) ->
    {
        System.out.println("Scheduling Runner");
    };
    return commandLineRunner;
}

    @Bean
    @Order(50)
    public ApplicationRunner createTimeRunner() {
        ApplicationRunner commandLineRunner =
    (ApplicationArguments args) -> {
        System.out.println("Time Runner");
    };
    return commandLineRunner;
}

    public static void main(String[] args) {

        SpringApplication.run(BootProj12RunnersClaassApplication.class,
    args);
    }

}

```

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