**Spring Core – Spring Boot – Micro Services**

**By Mr. Raghu**

**Spring Framework introduction:**

**Spring framework** is a RAD framework for Java based enterprise application.

**RAD**: Rapid Application Development.(Faster & less lines of code)

**Framework** 🡪 **Technologies** + **Design Patterns** (Solution to common problems with high performance)

**Case-1:** Web Application 🡨 Technologies (Servlet/JSP), Database(JDBC), dependencies, container/server…etc

🡨 3 months and 10k lines of code

🡨 Less performance

**Case-2:** Web Application 🡨 Framework

🡨 1 month and 3k line of code

🡨 High performance

**Spring Core:** In simple, it provides all rules and guidelines to work with Spring Framework and Spring Boot based applications.

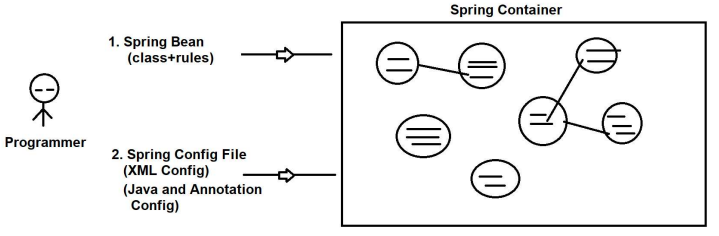
**Spring Container:** It is a Pre-defined Application that does

1. Scan/Find classes. Called Component Scanning
2. Create Object to detected class.
3. Provide data to variables and link objects (HAS-A) called Dependency Injection
4. Finally destroy the objects (When we stop Application/Server)

**Programmer has to provide two inputs to Spring Container:**

1. **Spring Bean**: class + Rules given by Spring container
2. **Spring Configuration**: (objName and link details)

**XML Config** | **Java Config** | \*\*\***Annotation Config**



**Spring Bean:**  is a class that follows the below rules given by spring framework.

* Package statement must be given.
* Class must be public with any name
* private variable (if exist/optional)
* Provide zero-param constructor **and/or** N-params constructor [sometimes both are used]
* Provide Mutators (setters, getters) 1 variable = 1 set / 1 get methods
* We can override 3 methods from object class mainly. **toString()**, **hashCode() +** **equals()** methods which are

[non-private, non-static and non-final]

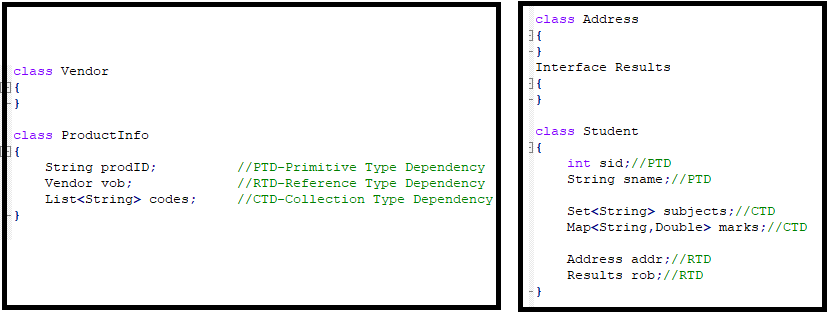
* Inheritance Rule: One Spring bean can extend another Spring f/w or our spring classes, but not other classes. Only Serializable(I) is allowed.
* Annotations: Only Spring Annotations allowed (some integrations we use :spring framework, integration such as Lombok, JPA[Hibernate], AOP ...etc)

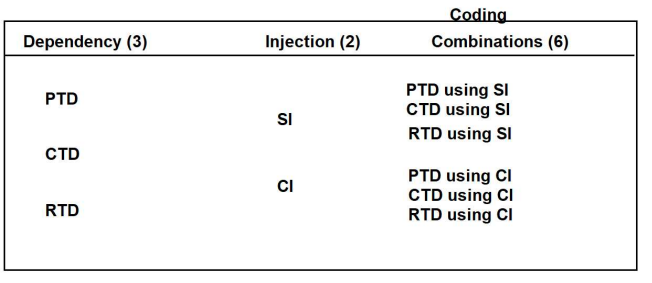
|  |  |
| --- | --- |
| **Theory** | **Programming** |
| OOPs | Core Java |
| ORM | JPA/Hibernate |
| DI | SpringF/W( Container/IoC) |

**Dependency Injection [Theory]**

**Dependency / Dependencies** (3): A variable created inside class (Spring Bean) is called Dependency.

* **Primitive Type Dependency(PTD)[8+1] :** If a variable is created using the following data types such as byte, short, int, long, float, double, boolean, char + String [Even Wrapper classes]
* **Collection Type Dependency (CTD)[4] :** If a variable is created using : List, Set, Map or Properties(C) [all are from java.util package]
* **Reference Type Dependency(RTD)[no count] :**If a variable is created using class or interface (another Spring bean) ( HAS-A Relation)

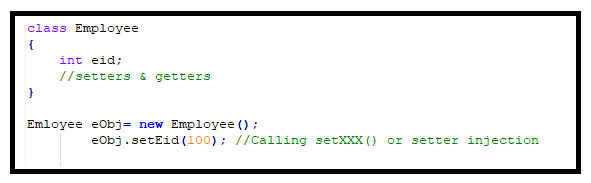




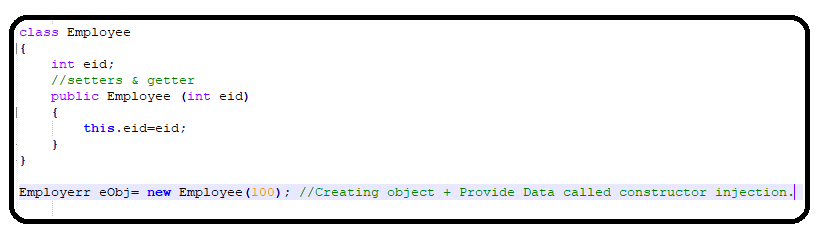
**Injection (4)/ (3):** Provide data to variable after creating object.

1. Setter Injection (SI)
2. Constructor Injection (CI)
3. Lookup Method Injection (LMI)
4. Interface Injection (not exist in Spring/Boot)

**1. Setter Injection (SI):** Provide data to variable after creating object by using setters.



**2. Constructor Injection (CI):**  Provide data to variable using PARAMETERIZED CONSTRUCTOR.

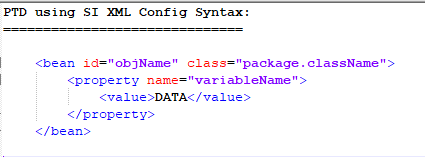


**3. Lookup Method Injection (LMI):**  It depends on spring bean scopes. If parent bean is singleton and child bean is prototype then we use this injection.

**4. Interface Injection:**  (not exist in Spring/Boot). Creating object to interface and link to variable.

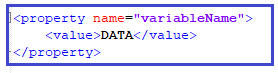
**Sring XML Configuration using DI Theory**

|  |  |
| --- | --- |
| **<bean>** | Container creates object |
| **<property>** | Calling set method(Setter Injection) |
| **<constructor-arg>** | Calling parameterized constructor(Constructor Injection) |
| **<value>** | Primitive type dependency(PTD) |
| **<list> , <set>, <map> , <props>** | Collection type dependency(CTD) |
| **<ref/>** | Reference type dependency(RTD) |

****

**We can write XML Config for Primitive in 3 different ways:**

**1. Value as Tag:**

****

**2. Value as Attribute**

****

**3. p:schema or p:namespace 🡪\*** Here, p is short format of property tag (setter)

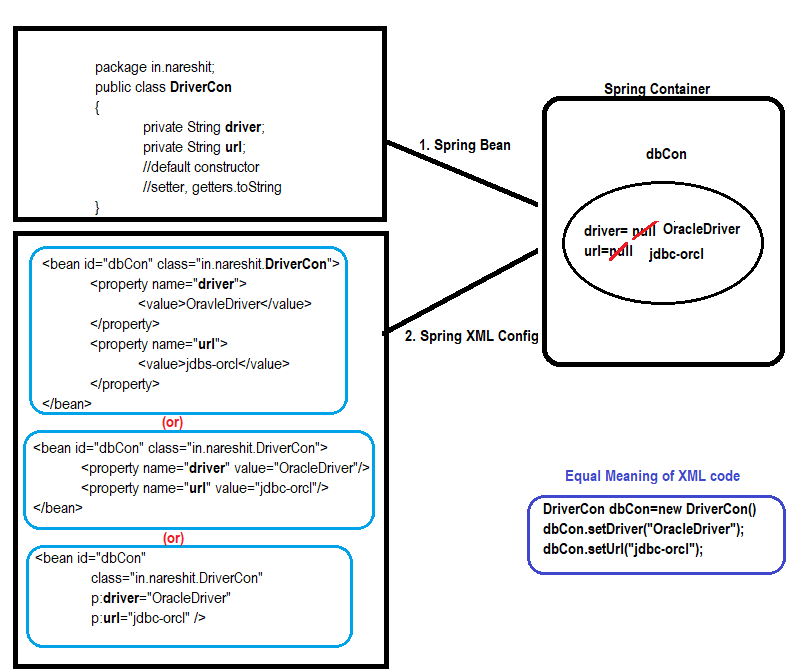
****

* If any “**p:**variableName” unidentified error then need to add the below schemaLocation in xml config file.

xmlns:c=*"http://www.springframework.org/schema/c"*

xmlns:p=*"http://www.springframework.org/schema/p"*>

**Example#1**

****

**Spring Container Types :**

**1. Old Container**

(BeanFactory) [I]

**XmlBeanFactory** (C) [It supports only XML Configuration]

**2. New Container**

(ApplicationContext) [I]

**ClasPathXmlApplicationContext**(C)

**AnnotationConfigApplicationContext**(C)

**ServletWebServerApplicationContext**(C)

..etc

* [It supports XML, Java and Annotation Configuration]

**New Container:**

**ApplicationContext(I)**

**IS-A**

**ClassPathXmlApplicationContext(C)**

ClassPath = src/main/resources folder

Xml = XML File

ApplicationContext = Spring Container

ClassPathXmlApplicationContext = Create Spring Container, by reading XML file which exist in location src/main/resources folder

**getBean("objName")**: Object This method is used to read one object from container.

**Setup:**

1. Download and install JDK : 8/11/Latest
2. Download and install STS(Spring tool suite)
3. Basics of Maven(Learn)
4. Basics of Lombok(Learn)

**ScApp1\_XMLConfig\_SI Project (Ref)**

1. Create maven project
2. Add compiler plugin, spring-context in pom.xml(Search from web)
3. Create Spring bean
4. Create SpringConfig.xml file & SpringConfig2.xml update accordingly. (Search spring xml configuration from web)
5. Create Test class: to check bean is created by Spring Container or not?

ApplicationContext ac = new ClassPathXmlApplicationContext("SpringConfig.xml","SpringConfig2.xml");

**Spring Configuration- Annotation**

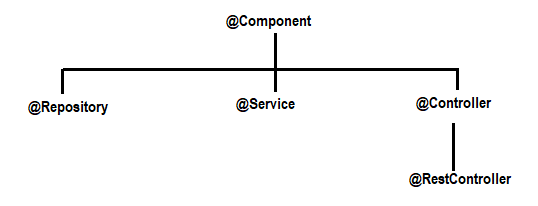
**Annotation Configuration:**

* Annotation configurationis fast execution comparing to xml/ java based configuration.
* It applicable for only programmer defined spring classes and should apply in source files(\*.java).

**Types of Annotation Configuration:**

**1. Stereo Type Annotations:** This type of annotations is used for object creation.

1. **@Component :** Creating object to our class
2. **@Repository :** Creating objects to our class + Database operation
3. **@Service :** Creating objects to our class + Transaction management, logics, calculation, sorting…. etc
4. **@Controller :** Creating objects to our class + Web Application – MVC (http request)
5. **@RestController :** Creating objects to our class + Web Services (http) + Data conversions(Json,xml)

****

**2. Basic /Support Annotation:** These are used to provide data and link other objects.

1. **@Value :** Called as Field Injection, no need of setters/constructor)

Provides default/hardcoded values**|** reads data from properties files**|** Provides SpEL data to variables in spring bean.

1. **@Autowired**, **@Qualifier**, **@Primary**
2. **@PropertySource**
3. **@ComponentScan:**  It is used to provide package[s] information to spring container for finding/scanning spring beans.

Here, Spring Container scans beans in specified package and its all sub packages.

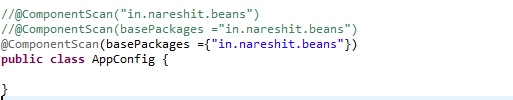
1. **@Scope**

..etc

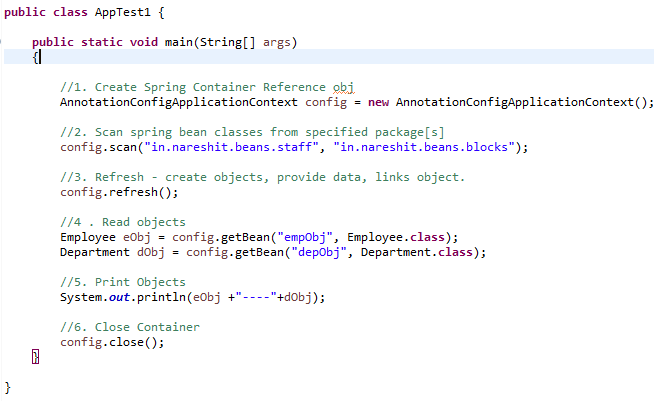
* Spring Boot has not given support for XML Config (Zero XML Config)
* In Spring Boot, we use mostly Annotation Config (99%) and Java Config (1%).

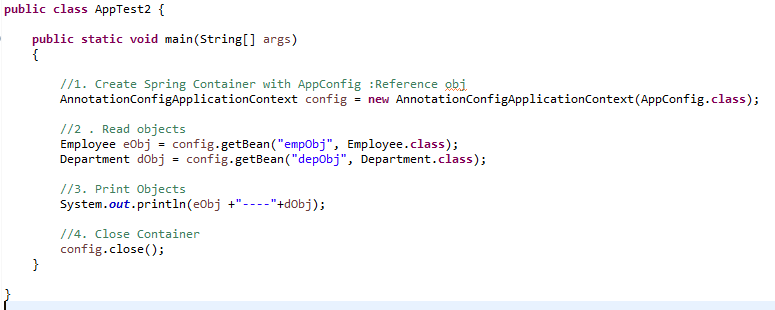
**ScApp2\_AnnotationConfig Project (Ref)**

1. Create maven project
2. Add compiler plugin, spring-context dependency in pom.xml(Search from web)
3. Create Spring beans with @Component and @Value annotations inside.
4. Create AppConfig call to provide basePackage information



1. Create Test classes





**basePackage:** This is one package name, given to Spring container find/scan classes from given package and subpackages.

**Setup Data:** Data comes at runtime from end-user (Form data such as login form, register form….)

Even developer/testing team provide some data before running/starting application is called setup data.

**Ex:** Database connection, connection pooling, security configuration, Web MVC prefix/suffix, Email properties, Logging properties, ORM properties…. Etc.

Such data is given form

1. Properties files
2. YAML files
3. VM/System Args
4. Cmd line Args

**Properties file:**

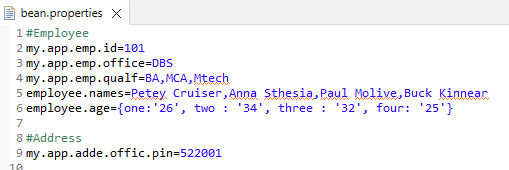
* <anyFileName>**.properies** and it should be created in **src/main/resources** [classpath]
* Data must be **key=value** format. Key can have dot (.), dash(-) , underscore(\_)

**Ex:** my.app-driver\_class=ABCD

* **Keys** are case-sensitive & **Value** is string type by default , auto-parsing supported.
* **‘#’** indicates comment.

**Working with Properties file(steps):**

1. Create properties files and have data in key=value format.

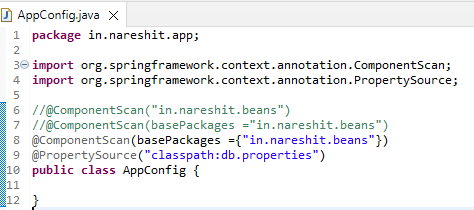


1. Load properties files in Spring Container by **@PropertySource**(“**classpath:**abcd.properties”)

Container loads properties file and created object to **Environment (I)** 🡪 **StandardEnvironment(C)**

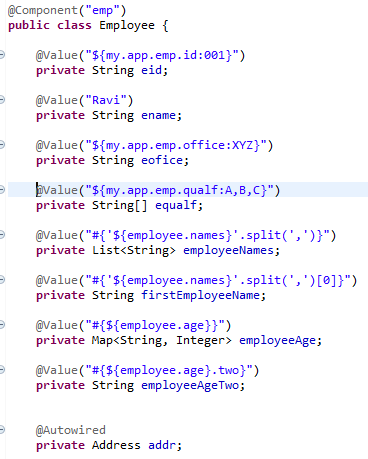
Stores all key=values

Note: word **classpath:** is optional for in spring non-web applications.



1. Read data from properties files.

**@Value( “ ${keyName} ”)**

****

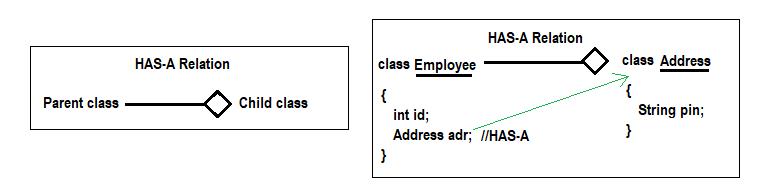
**Association (HAS-A) Relation/Mapping:** Using one class(child) object(object reference) inside another class(parent) is called as HAS-A relation.

Using child as a Data type inside parent and creating one variable is called as Association/HAS-A relation.

* **Wiring :** creating link between two objects which are in spring container.
* **Autowired:** Link between two objects is created by spring container.
* **@Autowired** annotation is used on top of HAS-A relation.
* If **@Autowired** annotation is not applied on top HAS-A relation then the link will not be created and holds the default value ‘null’.
* If **@Autowired** is applied on HAS-A variable but child object is not created(i.e no **@Component** on child class) then the spring container throws exception ‘NoSuchBeanDefinitionException…..’, Parent object will not be created. Application itself stops.
* **@Autowired(required=false):** If object is there link it else set ‘null.

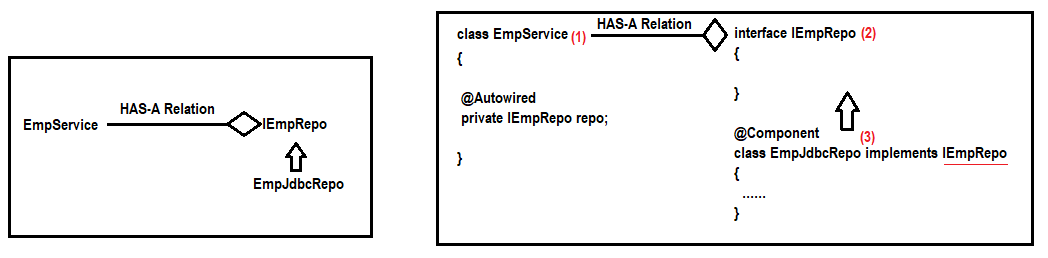
By default @Autowired internally works with ‘required=true’. It means child object must be provided, else Exception.

**Ref: ScApp3\_Autowired (project)**

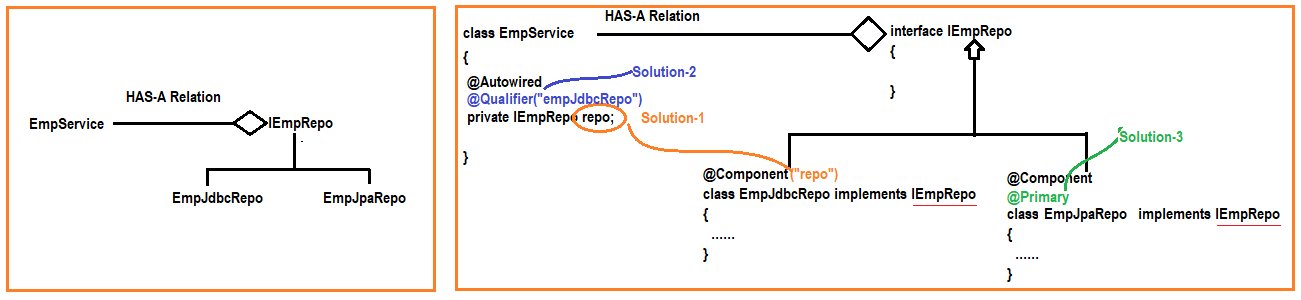


**Autowired using child interface:** If child is an interface in HAS-A relation then Autowired will search for implemented class object for child interface. Same injected into parent object.

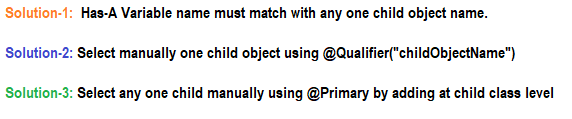
* We must not write @Component on top of interface. It will give runtime error Object can not create for interface.
* If Child interface has not implemented class or no @Component on top of child implemented class level then Spring Container throws **NoSuchBeanDefinitaionException**.



**Child Interface having multiple implementations class and objects:**



If child interface is having multiple Implementation classes and their objects are created Spring Container throws **NoUniqueBeanDefinitaionException.**

****

* Use **@Qualifier** annotation at HAS-A variable level to select one child object in-case of multiple found. [ Use this if child are predefined we have jar files or .class files. i.e no .java file]
* Use **@Primary** annotation at any one child Implementation class level, that says if multiple objects are exist then choose current class only.
* **@Qualifier** compares given objectname with every object in container(bit slow), **@Primary** says to Autowire “Do not search any other, Link it this”.

**Q) Can we use @Value for Object linking in Spring F/W?**

**A)** Yes. @Value(“#{objectName}”) on top of HAS-A variable

**Q) What is the difference between @Autowired and @Value?**

**ScApp3\_Autowired Project (Ref)**

**ScApp3a\_Autowired\_childIntrface Project (Ref)**

**ScApp3b\_Autowired\_multi\_Child Project (Ref)**

**Spring Bean – Lifecycle Method**

**Life Cycle Methods:** A method is called by container in between object creation and destroys.

=> In Servlet API 3 Life Cycle methods exist:

**init()** : Can be called only once after creating object.

**service()** : Can be called for every request

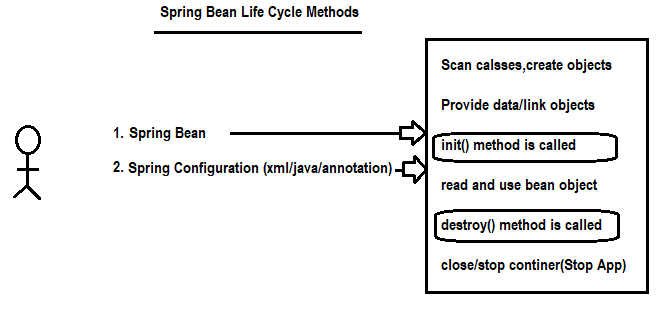
**destroy()** : Can be called only once before deleting the object by container

=> In Spring container 2 Life Cycle methods exist: (Optional to use)

init ()

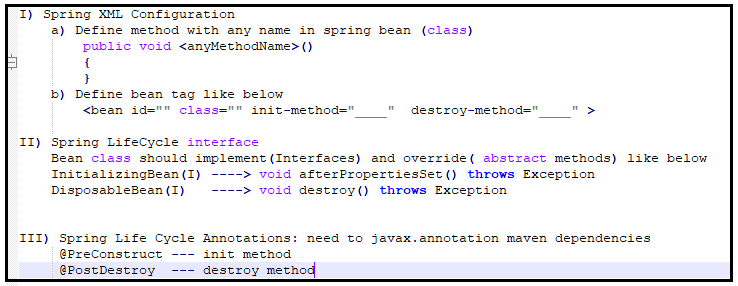
destroy()

\* destroy method will not destroy the object. It’s logic should be closes connection/files/streams.. etc.



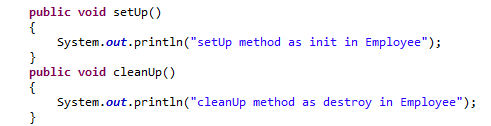
Life Cycle Method Coding:

1. Spring XML Configuration
2. \*\*\* Spring Life Cycle Interfaces
3. Spring Life Cycle Annotation(JEE)

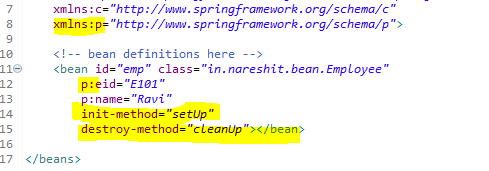


**ScApp4a\_LifeCycle\_XML Project (Ref)**

1. Create Maven project and add javasource compiler plugin, spring-contect dependencies from Web.
2. Create Spring Bean with two public void methods with any name(setup, cleanUp).



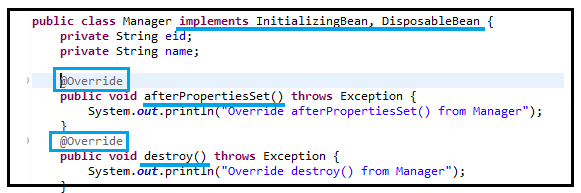
1. Create Spring XML config file and copy the structure from web and add **init-method, destroy-method** properties for bean tag



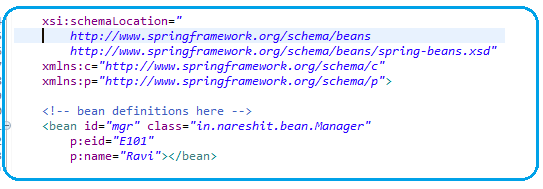
1. Create Test class then check the bean lifecycle methods flow.

**ScApp4b\_LifeCycle\_Interface Project (Ref)**

1. Create Maven project and add javasource compiler plugin, spring-contect dependencies from Web.
2. Create Spring Bean by implementing InitializingBean, DisposableBean interfaces.



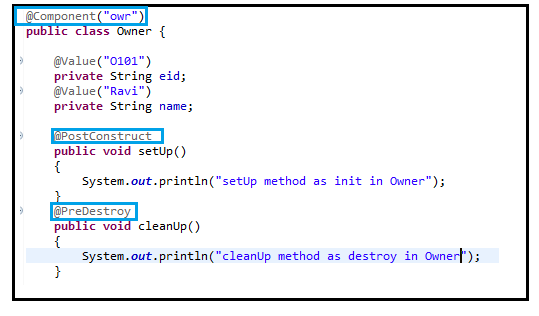
1. Create Spring XML config file and copy the structure from web and add bean tag



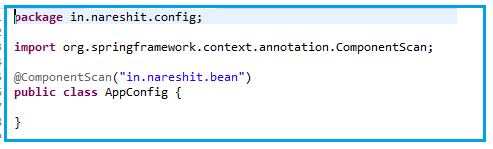
1. Create Test class then check the bean lifecycle methods flow.

**ScApp4c\_LifeCycle\_Annotation Project (Ref)**

1. Create Maven project and add javasource compiler plugin, spring-contect, **javax.annottion** dependencies from Web.
2. Create Spring Bean as follow with annotation based config.



1. Create Spring annotation based config to scan components



1. Create Test class then check the bean lifecycle methods flow.

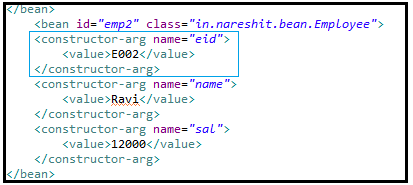
**Spring XML Config using Constructor Injection**

When we use <bean>, <properties> tags then Spring container uses default constructor to create object to class then provides data by using setter methods.

**Constructor Injection:** If spring container uses param-contructor to create object to class then it is called CI.

* 1 <constructor-arg> tag 🡪 1 parameter in constructor
* If class has 3 variables/3 setters, we can call them in any order (or) we can set even few values (need not to set all). But not in constructors, must pass all params in given order.
* **In Spring Boot, We use Autowiring concept with runtime data(properties & @value) there is no constructor injection in Spring Boot.**

**ScApp5\_XMLConfig\_CI Project (Ref)**

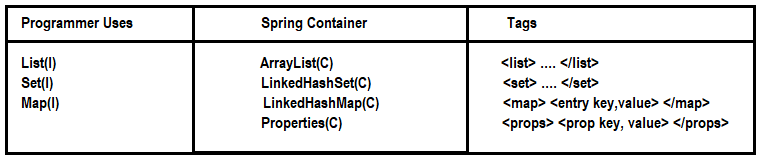
****

**Dependency – Collction Type(LIST,SET,MAP & Properties)**

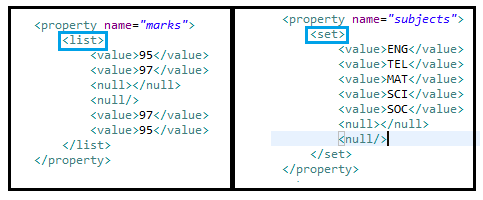
Dependencies:

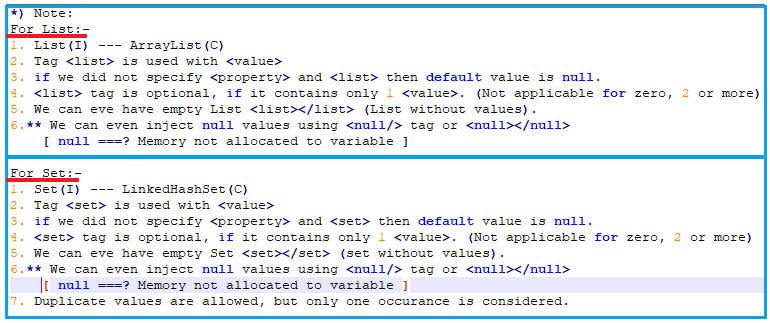
1. Primitive Dependency (8+1): byte, short, int, long, float, double, char, Boolean & String…
2. Collection Type Dependency(4): List(I), Set(I), Map(I) & Properties(C)
3. Reference Type Dependency:

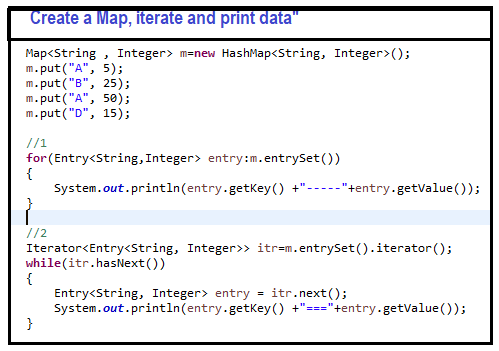
* Java/Spring says WORK WITH INTERFACES, I’LL CHOOSE IMPL CLASSES. Tomorrow if better IMPL class comes your code remains same.

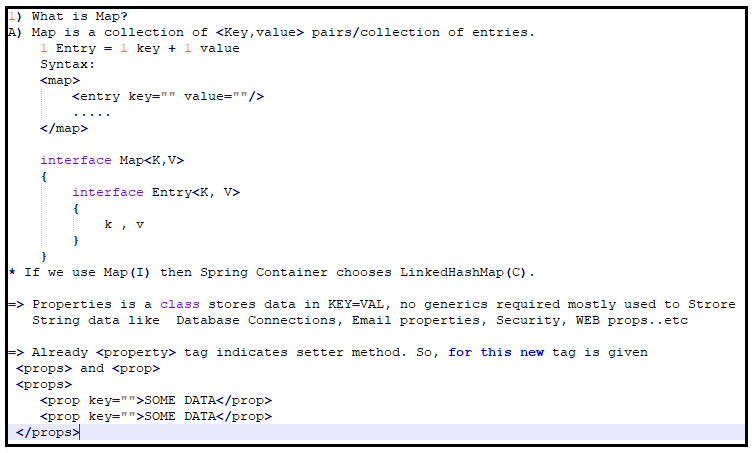


XML Configuration for List and Set:

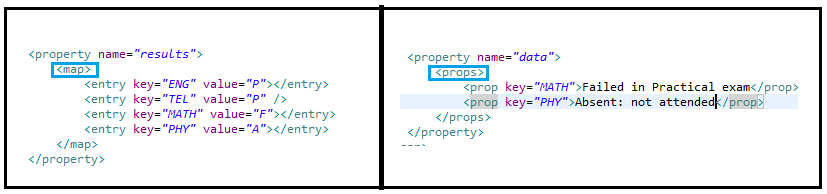


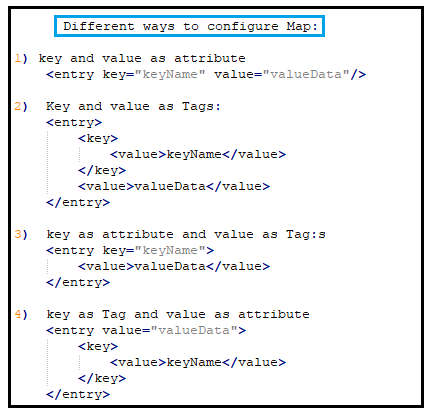






XML Configuration for Map and Properties:







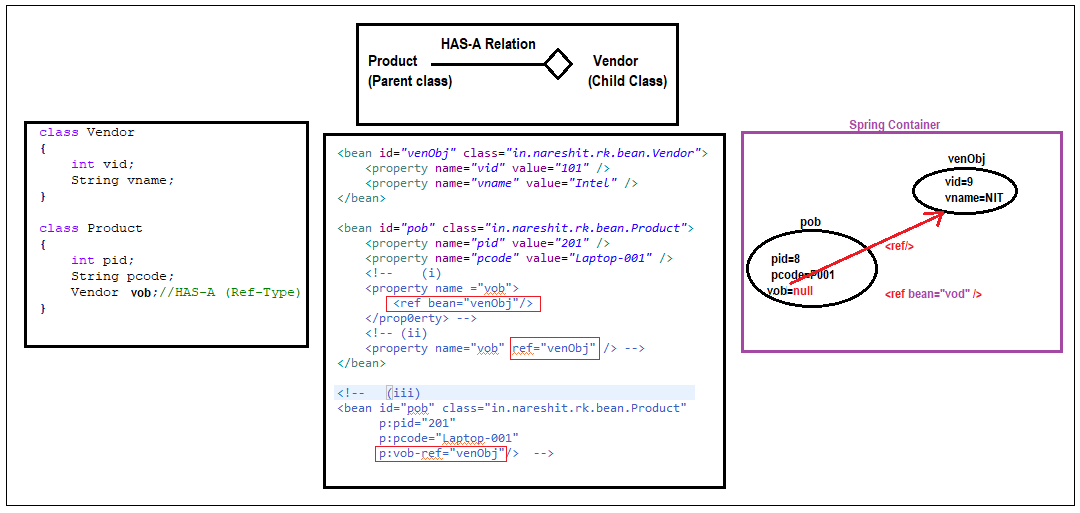
Ref: **ScApp6\_XmlConfig\_SI\_List\_Set\_Map\_Properties**

**Spring Core – Reference Tag - <ref/>**

HAS-A or Association Mapping: Creating a variable inside a class by using class/interface as data type.

* If two classes are connected using HAS-A relation then their objects are linked using **<ref/>** tag.
* The process of linking by using **<ref/>** Tag is called **wiring.** If **<ref/>** Tag is not used then default value is null(assigned). i.e no link between two objects.
* **3 Ways to define <ref/> Tag:** as marked red in diagram

1. **Ref as a Tag**
2. **Ref as attribute**
3. **p:schema-ref**

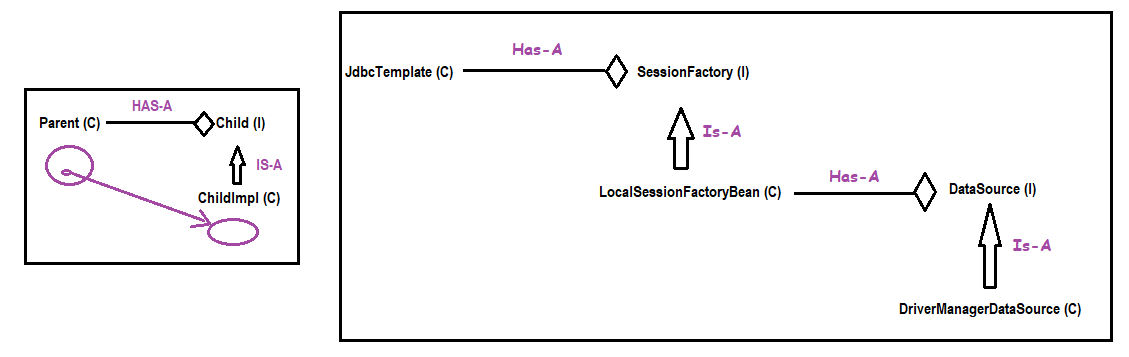


Ref: ScApp7\_XmlConfig\_RefTag

**Ref Tag using interface**

If Child is an interface then parent class object links to Child’s implementation class by using <ref/> tag.

If Two classes are connected using interface reference is called loose-coupling.



**Spring Bean -Scope**

**Scope**: lifetime/period of object in memory. \* When it is created, and How long data exist in memory

**Core Java Scopes**

1. local scope --> local variables

2. object scope --> instance variables

3. global scopre --> static variables

**Servlets Scopes in Web application)**

1. request scope : Data stored at server until response given

2. session scope : Data stored at server from login to logout

3. context scope : Data stored at server from app start to stop.

**Q) When bean/object is created in Container and how long it exists?**

A) It depends on the scope is used in programming.

Spring Bean scopes

1. singleton –default scope
2. prototype
3. request [web application]
4. session [web application]
5. global context [Removed]
6. **singleton** : For every configuration only one object is created by spring container when app started or container is created and exist until app stopped or container destroyed.

1 <bean> 1 object is created inside container.

1 @Component 1 object is created inside container.

1 @Bean 1 object is created inside container.

**Ex:** DataBase connection, Service/logic objects, ViewResolver…etc

**Realtime:** Controller, Service, DAO are recommended to be singleton

1. **prototype :** For every access/use one new object is required. Object is created by container for every request.

**Ex:** Token

1. **request:** Object is created when request comes and destroy once response committed.

**Ex:** Form Data(Register,Login,Payment….etc)

1. **session:** When we login is successfully a new object is created and stored until logout.

**Ex:** User data like role, name,id…. Etc

Coding syntax:

1. **XML Configuration :**

*<****bean*** id=”…” class=”…” ***scope*=”…”** />

1. **Annotation Configuration:**

***@Component(“…”)***

***@Scope(“…”)***

public class ClassName

{

…

}

1. **Java Configuration:**

***@Configuration***

public class AppConfig

{

**@Bean**

***@Scope(“..”)***

public <ClassName> <objName>()

{

….

}

}

**Ref: ScApp8\_SpringBeanScopes project**

**Lookup Method: Method level Injection**

Spring could inject dependencies between the beans in the application by following way of injection types:

1) Setter Injection

2) Constructor Injection

3) Field Injection (@Autowired at field)

4) Method Injection

Spring Beans have mainly two types of scope

1. Singleton (Instantiate only one object)

2. Prototype (Instantiate a new object every time).

In Spring application there are many beans injected to each other for a goal.

There is no problem when injected beans have same scope of beans like singleton bean injected with other singleton beans.

Sometimes in Spring, Problems arise when you need to **inject a prototype-scoped bean in a singleton-scoped bean**.

Since singletons are created (and then injected) during context creation it’s the only time the Spring context is accessed and thus prototype-scoped beans are injected only once.

**Lookup or Method Injection**

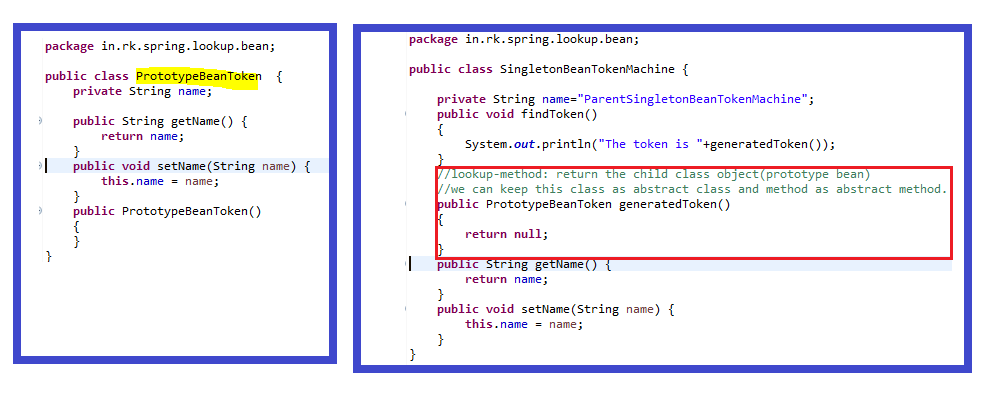
Spring provides another way for injection of beans, It is called method injection.

It is solution of above problem in injecting different scoped beans.

It works as that since singleton beans are instantiated at context creation, but it changes the way prototype-scoped are handled, from injection to created by an abstract method.

**ScApp8a\_LookupMethodInjection Project (Ref)**

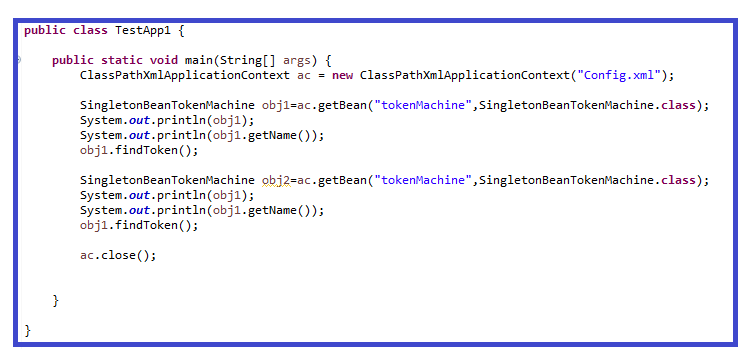
1. Create Maven project and add javasource compiler plugin, spring-contect dependencies from Web.
2. Create Spring bean classes **child(prototype)** and **parent(singleton)**



1. Create **config.xml** xml based spring configuration

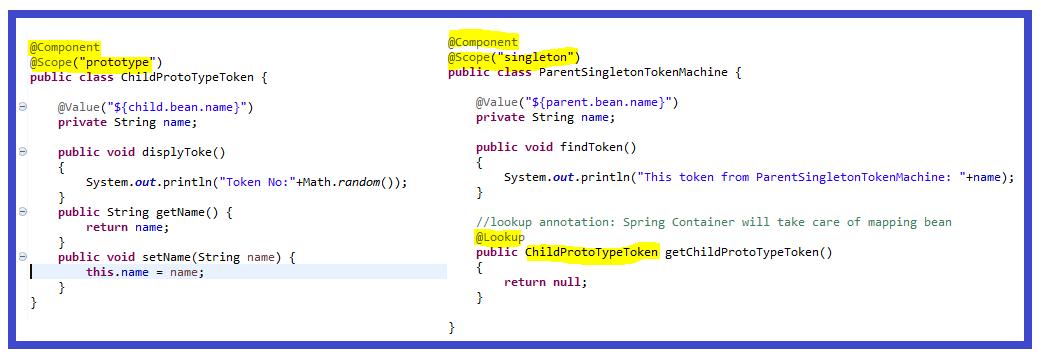


1. Create Test class then check the lookup methods flow.

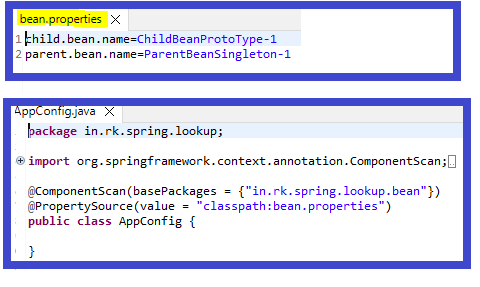


**ScApp8a\_LookupMethodInjectionAnnotation Project (Ref)**

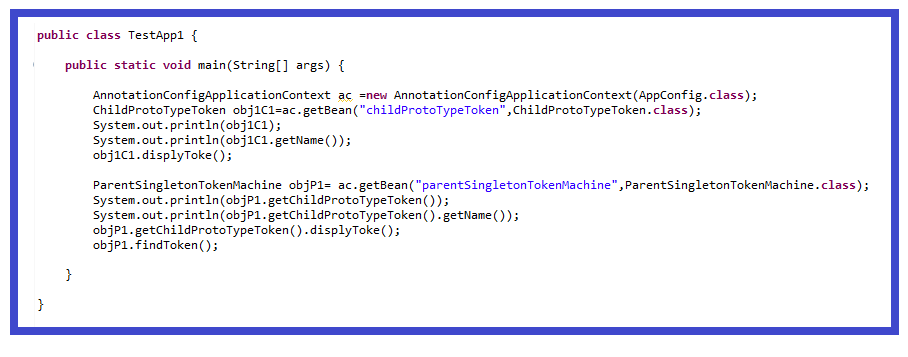
1. Create Maven project and add javasource compiler plugin, spring-contect dependencies from Web.
2. Create Spring bean classes **child(prototype)** and **parent(singleton)**



1. Create Spring bean properties files and AppConfig file



1. Create Test class and check the lookup method injection



**Spring Boot Introduction**

Spring Boot is a spring based project provided by **Pivotal** **team**. It behaves like abstract and reduces the generic configuration ( Zero- XML configuration) compared to spring framework.

All predefined configurations are provided by jars (dependencies) known as starters.

ex: spring-boot-starter-jdbc

spring-boot-starter-web

spring-boot-starter-mail

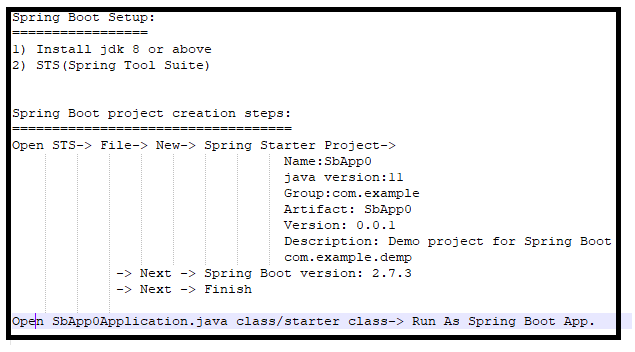
spring-boot-starter-security ..etc

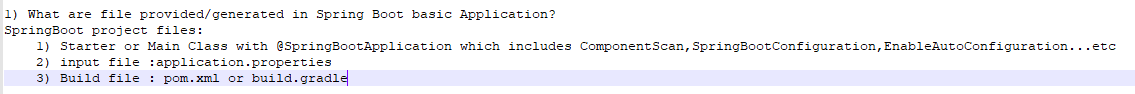
Our project is called as Child Project of one of the spring boot starter projects that must be created either using Maven or using Gradle.

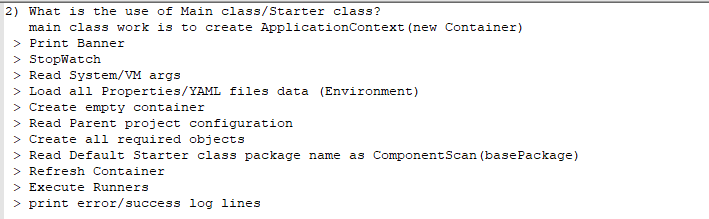
A lot of predefined properties are available in Spring Boot and can provide our own properties.

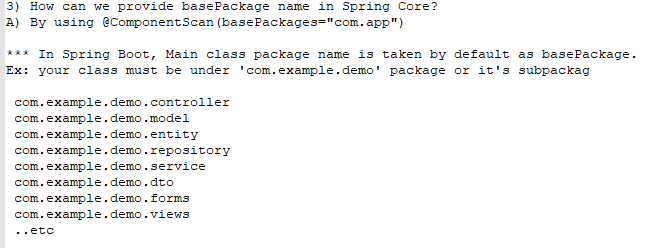
[Common Application Properties (spring.io)](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html#appendix.application-properties)

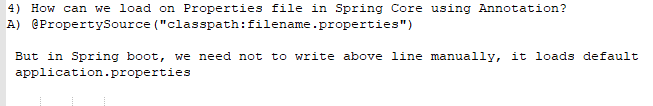
Official: [Spring Boot Reference Documentation](https://docs.spring.io/spring-boot/docs/current/reference/html/index.html)

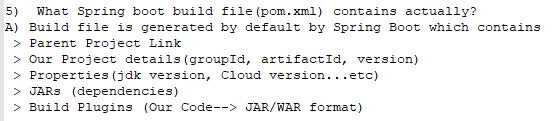


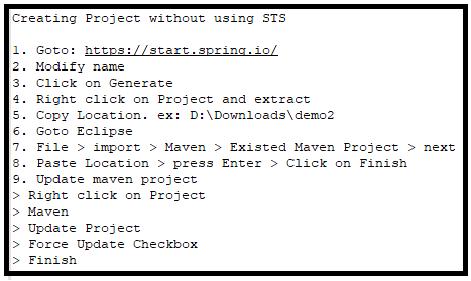












Ref: **ScApp0**

**Spring Boot Runners**

**Runner:** Some logic or block of code that should be executed only once when application is started.

We can have multiple runners in our application. All will get executed while application is started.

**What type of code/logic?**

1. Testing Objects
2. Profile Management
3. Setup SQL/DB Schema

Incase multiple Runners are created then all are executed in Name sorting order(Unicode naming order).

Ex: We have 3 Runners

MessageRunner, EmailRunner, SqlDbSetupRunner

Then the order could be 1) EmailRunner 2) MessageRunner 3) SqlDbSetupRunner

**Order of execution based on ClassName in alphabetic order.**

The execution order can be controlled by **@Order** or **@Order(someNumber)** annotation. Smallest number 1st execution order.

Execution order for Runners @Order(**-5**)-->@Order(**3**)-->@Order(**39**)-->@Order

Ex: Test logic.

Ctrl+Shift +T : open any predefined file

Ctrl+Shilf+R : open any file exist in our workspace

Ctrl+O : to view class member

Ctrl+L : Go to Line number

**Types of Runner:**

1) CommandLineRunner(FI)-SB(1.0): void run(String... args) throws Exception;

2) ApplicationRunner(FI)-SB(1.3): void run(ApplicationArguments args) throws Exception;

**Create Runner class:**

1) Public class with any name + @Component and/or @Order(-)

2) Implement CommandLineRunner or ApplicationRunner interface and override run(-) method.

3) Run-->Run Configuration--> Arguments --> code test deploy --port=8989 --code=ORCL --loc=abc --loc=mno --spring.app.name=FROM\_CMD\_ARGs

4) Run Main class.

**CommandLineRunner**

package com.example.demo.runners;

import java.util.Arrays;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

@Component

@Order(5)

public class CmdRunnerMessageRunner implements CommandLineRunner {

    @Value("${spring.app.name}")

    private String name;

    @Override

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

        System.out.println(this.getClass().getName()+"-Execution only once.");

        System.out.println("CmdLine args  if any:"+Arrays.asList(args));

        System.out.println("Server Name"+name);

    }

}

**ApplicationRunner**

package com.example.demo.runners;

import java.util.Arrays;

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

import org.springframework.boot.ApplicationArguments;

import org.springframework.boot.ApplicationRunner;

import org.springframework.core.annotation.Order;

import org.springframework.stereotype.Component;

@Component

@Order(56)

public class AppRunnerMessageRunner implements ApplicationRunner {

    @Value("${spring.app.name}")

    private String name;

    @Override

    public void run(ApplicationArguments args) throws Exception {

        System.out.println(this.getClass().getName()+"-Execution only once.");

        System.out.println("AppRunner args  if any:"+Arrays.asList(args));

        System.out.println("NonOptionArgs:---->"+args.getNonOptionArgs());

        System.out.println("OptionArgs:---->");

        for(String each:args.getOptionNames())

        {

            System.out.println(each+" ===> "+args.getOptionValues(each));

        }

        if(args.containsOption("port"))

        {

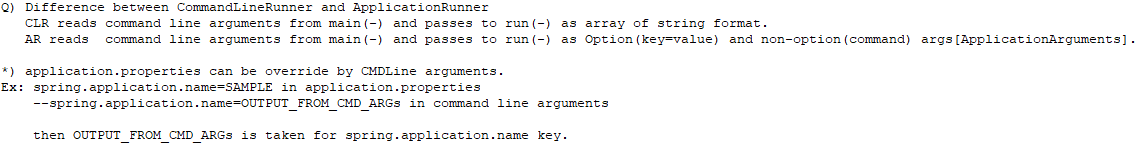
            System.out.println("Server port="+args.getOptionValues("port"));

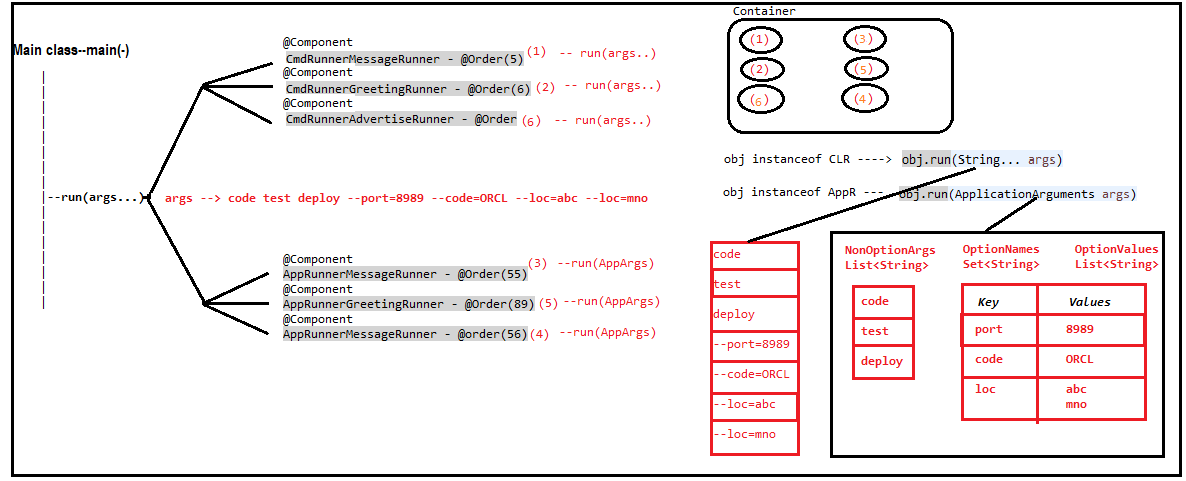
        }

        System.out.println("Server Name"+name);

    }

}





Ref: **SbApp1\_Runners (Project)**

Ref: **SbApp1a\_Runners\_AppConfig\_Java8Ways (Project)**

@Configuration

public class AppConfig {

    @Bean

    public CommandLineRunner getCLMessageRunner() {

        //Anonymous interface

        return new CommandLineRunner() {....};

        //or

        //Lambda Expression

        //or

        return (args) -> {...};

        //Method reference

        return RunnerUtility:: getCLAdvertiseRunner;

    }

}

public class RunnerUtility {

    public static void getCLAdvertiseRunner(String... data )

    {

            System.out.println("RunnerUtility.getCLAdvertiseRunner()--Execution only once.");

            System.out.println("CmdLine args  if any:" + Arrays.asList(data));

    }

}

**\*) No Runners are required for UI based applications(Rest,Cloud,...etc).**

**\*) Runners logic can be placed in main(-) itself. But not recommended (Code modularity)**

**\*) Main class can implement Runners interface but not recommended.**

**Read or Load properties data into variables**

**Spring Core**

1) All key=value properties in **someProperties.properties** files under **resources** folder

2) AppConfig with **@ComponentScan**(basePackages ={"in.onepackage","in.twopackage"}), **@PropertySource**("classpath:someProperties.properties")

3) **@Value**("${propertyNameOrKey} at variable level and **@Component** at class level

**Spring Boot**

1) All key=value properties in application.properties file under resources

2) **@Value**("${propertyNameOrKey} at variable level and @Component at class level

\* No need of **@PropertySource, @ComponentScan** annotation

\* All classes must be under starter/main class package or its subpackage.

\* **@ComponentScan** is set to main class package by default.

**Project creation steps:**

1) Create Spring project: jdk-11, lombok

2) Create SomePojo Employee class with eid, ename,sal,... and @Value at variable level and lombok annotations

3) Update Application class: Get ApplicationContext and get required bean.

4) provide application.properties data

**Employee.java**

package com.example.demo.beans;

import java.util.List;

import java.util.Map;

import java.util.Set;

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

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

import org.springframework.stereotype.Component;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@Component("empObj")

public class Employee {

    @Value("${my.app.emp.id:10001}")

    private int id;

    @Value("${my.app.emp.name:abcName}")

    private String name;

    @Value("${my.app.emp.sal:100000.00}")

    private double sal;

    @Value("${my.app.emp.qualfs:Btech,MBA}")

    private String[] qualfs;

    @Value("${my.app.emp.mgrs:X-Manager X,Y-Manager Y}")

    private List<String> mgrs;

    @Value("${my.app.emp.mnos:8888888888,9999999999}")

    private Set<String> mnos;

    @Value("#{${my.app.emp.exps}}")

    private Map<String, String> exps;

    @Value("#{${my.app.emp.exps}.DBS}")

    private String currentExp;

    @Autowired

    private Address addr;

}

**Address.java**

package com.example.demo.beans;

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

import org.springframework.stereotype.Component;

import lombok.Data;

@Data

@Component

public class Address {

    @Value("8-1-67")

    private String hno;

    @Value("${my.app.emp.addr.pin:100001}")

    private String pin;

}

**application.properties**

#Employee

my.app.emp.id=101

my.app.emp.name=Ravi

my.app.emp.sal=25000.00

my.app.emp.qualfs=BA,MCA

my.app.emp.mgrs=Petey Cruiser,Anna Sthesia,Paul Molive,Buck Kinnear

my.app.emp.mnos=7777777777,6666666666

my.app.emp.exps={DBS:'3yrs', GL : '3.3yrs', HCL : '3.3yrs'}

#Address

my.app.emp.addr.pin=522001

**SbApp2aLoadDataValueApplication.java**

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ConfigurableApplicationContext;

import com.example.demo.beans.Employee;

@SpringBootApplication

public class SbApp2aLoadDataValueApplication {

    public static void main(String[] args) {

        ConfigurableApplicationContext conext=SpringApplication.run(SbApp2aLoadDataValueApplication.class, args);

        Employee eObj=conext.getBean("empObj",Employee.class);

        System.out.println(eObj);

    }

}

**Ref: SbApp2a\_LoadData\_Value (Project)**

**Q) What if @Value(-) is unable to find given key from properties files?**

\* if key is not present Spring core will take the ${...} or null. But in Spring Boot **IllegalArgumentException:Could not resolve placeholder.**

\* If duplicate keys are present latest one will be taken.

**Read or Load properties bulk data into variables**

**@ConfigurationProperties(prefix=”somePrefixKey”)**

@ConfigurationProperties/bulk Reading or loading at a time based on one common prefix.

When spring bean has a lot of variables/keys then keeping @Value at each variable level is not recommended.

**Q) What if prefix is not matched with properties files.**

No data is injected. Object is created with default values.

**Q) Is prefix must be given in code?**

At max we can define for one class with empty prefix.(But not recommended).

Ex: my.email.pwd=emalpwd

my.db.pwd=dbpwd

package com.example.demo.beans;

import java.util.LinkedHashMap;

import java.util.List;

import java.util.Set;

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

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

import org.springframework.stereotype.Component;

import lombok.Data;

import lombok.NoArgsConstructor;

@Data

@NoArgsConstructor

@Component("empObj")

@ConfigurationProperties(prefix="my.app.emp")

public class Employee {

    private int id;

    private String name;

    private double sal;

    private String[] qualfs;

    private List<String> mgrs;

    private Set<String> mnos;

    private LinkedHashMap<String, String> exps;

    @Autowired

    private Address addr;

}

package com.example.demo.beans;

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

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

import org.springframework.stereotype.Component;

import lombok.Data;

@Data

@Component

@ConfigurationProperties(prefix="my.app.emp.addr")

public class Address {

    @Value("8-1-67")

    private String hno;

    private String pin;

}

**application.properties**

#Employee

my.app.emp.exps.DBS=3yrs

my.app.emp.exps.GL=3.3yrs

my.app.emp.exps.HCL=3.3yrs

my.app.emp.mnos=7777887777,6666677666

my.app.emp.sal=29000.00

#my.app.emp.qualfs=BA,MCA

my.app.emp.qualfs[1]=MCA

my.app.emp.qualfs[0]=BCA

my.app.emp.id=206

my.app.emp.name=Ravi

my.app.emp.mgrs=Petey Cruiser,Anna Sthesia,Paul Molive,Buck Kinnear

#Address

my.app.emp.addr.pin=533001

**Project creation steps:**

1) Create Spring project: jdk-11, lombok

2) Create SomePojo Employee class with eid, ename,sal,... and @Component, @ConfigurationProperties at class level and lombok annotations

3) Update Application class: Get ApplicationContext and get required bean.

4) Provide application.properties data

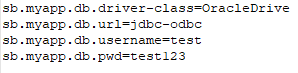
**Ref: SbApp2b\_LoadBulkData\_ConfigurationProperties (Project)**

**YAML(Yet Another Markup Language)**

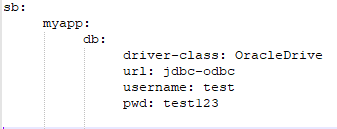
It is new way of key-value representation format.

It reduces duplicate levels/words in keys & We can represent a lot of keys with minimal spaces.

**application.properties**

****

**application.yml**

****

**Rules to work with yml.**

1. File extension is **.yml** in Spring Boot.
2. Must have any prefix/level only once in file [No duplicate level]
3. For every level use colon(:) in place of dot (.) and go to next line to start next level.

dot(.) = colon(:) + new line

1. Must provide few **(two)** spaces for new level but same level of keys must have the same spaces count.
2. Between last level and value **(after colon) must give one space.**

\*) All yaml properties are converted into key=value of Environment type through snake-yaml api(jar).

Reads YAML file

Parse into key=value

Loads data into Environment (memory)

Key=Values never follow order to define in Properties/YAML

\*) We can find some online tool to convert application.properties to application.yml and vice versa.

**application.yml**

my:

  app:

    emp:

      exps:

        DBS: '3yrs'

        GL: '3.3yrs'

        HCL: '3.3yrs'

      mnos: 7777887777,6666677666

      sal: '29000.00'

      qualfs[1]: MCA

      qualfs[0]: BCA

      id: '206'

      name: Ravi

      mgrs: Petey Cruiser,Anna Sthesia,Paul Molive,Buck Kinnear

      addr:

        pin: 533001

**Project creation steps:**

1) Create Spring project: jdk-11, lombok

2) Create SomePojo Employee class with eid, ename,sal,... and @Component, @ConfigurationProperties at class level and lombok annotations

3) Update Application class: Get ApplicationContext and get required bean.

4) provide application.yml data

**Ref: SbApp2c\_LoadBulkData\_Yaml (Project)**

**\* ) If we forgot to provide space between key: and its value then exception will be thrown**

ERROR org.springframework.boot.SpringApplication - Application run failed

org.yaml.snakeyaml.scanner.ScannerException: while scanning a simple key

\*) If we have properties and yml file in project with same key in both.

It is not good approach but it gives **high priority from properties file.**

**Spring Boot - Profiles Concept**

Realtime Environment:

1. Developer environment- DEV
2. Test/QA environment- SIT,UAT
3. Production environment-PROD

SpringBoot application could be moved from one environment to another environment such as DEV,SIT,UAT,PROD.

Application is same but properties are varying from one env to another such as db, endpoint for other systems.

Spring boot provided a concept called Spring Profiles concept where env related properties can be configured.

Profiles: to handle multiple properties files for env based is called as profiles.

Syntax: **application-[profileName].properties**

Example:

application.properties is **default profile**

application-dev.properties

application-qa.properties

application-sit.properties

application-uat.properties

application-prod.properties

\*) When the profiles are activated?

Profile is activated while running application by passing one command line argument.

**--spring.profiles.active=qa**

Here -- indicates input to container.

\*) How can we add Lombok after project creation?

1) Add manually Lombok dependency in pom.xml

2) Right click on project🡪Spring🡪Add starter🡪Search Lombok 🡪 Select Lombok 🡪Next 🡪Choose pom.xml 🡪Finish

**Project creation steps:**

1) Create Spring project: jdk-11, lombok

2) Create MsgCmdRunner class by implementing CommandLineRunner wth @Component and one @Value by reading properties key.

3) Update application.properties files

4) Run the App.

package com.example.demo.runner;

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

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

@Component("msgcmdrunner")

public class MsgCmdRunner implements CommandLineRunner {

    @Value("${my.app.db}")

    private String db;

    @Override

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

        System.out.println(db);

    }

}

**application.properties**

my.app.db=FROM DEFAULT

**application-qa.properties**

my.app.db=FROM QA

**application-sit.properties**

my.app.db=FROM SIT

**application-uat.properties**

my.app.db=FROM UAT

**application-prod.properties**

my.app.db=FROM PRODUCTION

**Ref: SbApp3a\_ProfilesCmdRunner (Project)**

**Execution Steps:**

1 > Open Main class > Run As > Spring Boot Application Output: FROM DEFAULT

o/p: FROM DEFAULT

2 > Open Main class > Run As > Run Configurations > Click on Arguments tab > Under program Arguments --**spring.profiles.active**=sit > Apply > Run

o/p: FROM SIT

3 > Do maven install and got target location then run the below command from cmd prompt.

java -jar SbApp3a\_ProfilesCmdRunner-0.0.1-SNAPSHOT.jar --spring.profiles.active=sit

**Q) If given profile is not found then what will happen?**

A) Spring Boot will search for given profile name first.

If not found then it will goto : default profile [ application.properties ]

\*) Profiles can be handled using YAML Files too

**application-qa.yml**

**application-prod.yml**

**Q) What if we pass two profile names to activate?**

ex: --spring.profiles.active=qa,prod

1. It is not a good approch. But first qa profile is loaded and later prod profile overrides key=vals.

**Spring Boot – Spring Java based configuration**

Spring has 3 types of configuration

1. XML [Removed in Spring Boot]
2. Annotation [@Component,@Value, @Autowired…. Etc]
3. Java [@Bean, @Configuration]

\*) Annotation Configuration can be applied on user defined classes. i.e source code.

Not applicable for pre-defined classes.

\*) Java Configuration can be applied for pre-defined classes.

**Q) When to choose Java and Annotation Configuration?**

A) You want to create object to a class

If class is pre-defined then use Java Configuration

If class is programmer-defined then use Annotation Configuration

**Java configuration coding steps:**

1. Define a public class with any name.
2. Apply @Configuration annotation over class.[indicated inputs to container]
3. Define one method for one object as below

public <ClassName> objectName()

{

//logic and return objectName

}

1. Apply **@Bean**  annotation over the method[that indicated object creation]

**Q) Can we use/Write Java Configuration for Programmer defined class? (or user defined class)?**

A) Yes possible. But not recommended.

Q ) Can we use @Configuration for a pre-defined class? Why?

A) No, we must have source code and able to compile code then only we can use annotation configuration.

Q) Which Configuration is used for pre-defiend classes? provide steps.

A) Java Configuration

>> Define one public class

>> apply @Configuration

>> Define one method for one object creation

>> apply @Bean

package com.example.demo.config;

import org.springframework.boot.info.JavaInfo;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class AppConfig {

    @Bean

    public JavaInfo javaInfoObj()

    {

        return new JavaInfo();

    }

}

package com.example.demo;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.info.JavaInfo;

import org.springframework.context.ConfigurableApplicationContext;

@SpringBootApplication

public class SbApp4JavaConfigurationBeanApplication {

    public static void main(String[] args) {

        ConfigurableApplicationContext context=SpringApplication.run(SbApp4JavaConfigurationBeanApplication.class, args);

        JavaInfo obj=context.getBean("javaInfoObj",JavaInfo.class);

        System.out.println("Java version:"+obj.getVersion());

    }

}

**Ref: SbApp4\_JavaConfigurationBean (Project)**

Note: The above code is manual java configuration. We use Auto Configuration most of the time.

**Spring Bean -Scope**

**Scope**: lifetime/period of object in memory.

\*\*\* When it is created, and How long data exist in memory

**Core Java Scopes**

1. local scope --> local variables

2. object scope --> instance variables

3. global scopre --> static variables

**Servlets Scopes in Web application)**

1. request scope : Data stored at server until response given

2. session scope : Data stored at server from login to logout

3. context scope : Data stored at server from app start to stop.

**Q) When bean/object is created in Container and how long it exist?**

A) It depends on the scope is used in programming.

Spring Bean scopes

1. singleton –default scope
2. prototype
3. request [web application]
4. session [web application]
5. global context [Removed]
6. **singleton** : For every configuration only one object is created by spring container when app started or container is created and exist until app stopped or container destroyed.

1 <bean> 1 object is created inside container.

1 @Component 1 object is created inside container.

1 @Bean 1 object is created inside container.

**Ex:** DataBase connection, Service/logic objects, ViewResolver…etc

**Realtime:** Controller, Service, DAO are recommended to be singleton

1. **prototype :** For every access/use one new object is required. Object is created by container for every request.

**Ex:** Token

1. **request:** Object is created when request comes and destroy once response committed.

**Ex:** Form Data(Register,Login,Payment….etc)

1. **session:** When we login is successfully a new object is created and stored until logout.

**Ex:** User data like role, name,id…. Etc

Coding syntax:

1. **XML Configuration :**

*<****bean*** id=”…” class=”…” ***scope*=”…”** />

1. **Annotation Configuration:**

***@Component(“…”)***

***@Scope(“…”)***

public class ClassName

{

…

}

1. **Java Configuration:**

***@Configuration***

public class AppConfig

{

**@Bean**

***@Scope(“..”)***

public <ClassName> <objName>()

{

….

}

}

**Ref: ScApp8\_SpringBeanScopes project**

**Ref: SbApp5a\_SpringScopesSingletonePrototype project**

1. Create spring starter project : java 11 & Lombok
2. Create Two pojos DataBaseConnection(singleton scope) and TokenService(prototype) using annotation configuration
3. Update application class to get beans multiple times.

* If we did not specify the scope then default scope is singleton.
* If we provide any wrong scope name then Spring container will throw IllegalStateException: No Scope Register…
* **request** and  **session**  scopes works only in of web applications. i.e Spring web dependency is required.
* **If scope is singleton then object is created for class when container is created but if scope is prototype then the object is created when object is accessed/used.**
* **All singleton beans are created while Container is created.**
* **We can make singleton scoped object creation in Container Lazy by using *@Lazy* annotation.**

**Q) Servlet, What is loan-on-startup? Why it is used?**

**A)**  Servlet, by default lazy.

1 servlet--- 1 object is created when 1st request comes by servlet container.

***<load-on-startup>+ve value</load-on-startup>*** gives inputs to servlet container to create servlet object when container started.

**Q) Spring, Why is eager/early loading for singleton? How can we make it lazy?**

**A)** When Spring container is about to create all singleton objects are created: is call eager/early loading in spring.

***@Lazy***  *on class gives indication to spring container to create the singleton object when 1st accessed/used.*

**Spring Boot Scheduling**

**Scheduling:** to execute any logic over a **period of time** or **point of time**.

PERIOD of TIME = Time Gap (4days, 6hrs, 35 mis..etc)

POINT of TIME = Exact Date and Time. ( 12 Jan 9AM, 26th Dec 1PM… etc)

Ex: Report Generation daily, weekly, monthly for online application

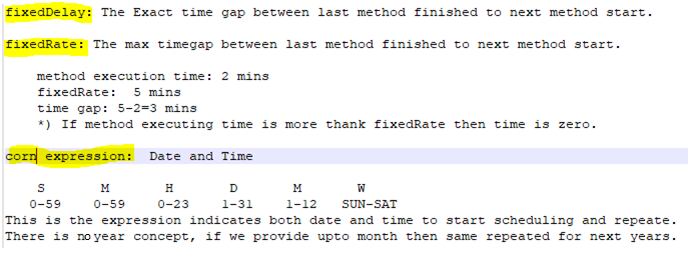
Monthly Bank or Credit card statements.

Monthly Salary Pay slip

B’Day Remainders, EMI remainders, TAX Returns… etc

Scheduling Programming:

1. **fixedDelay** : Exact time gap between old task finish to next task to be started/executed.
2. **fixedRate** : The Max time gap between old task finish to next task to be started/executed.
3. **\*\*\* cron expression:** Configured date and time.



Programming Steps:

Two annotations required to work with Spring Boot scheduling

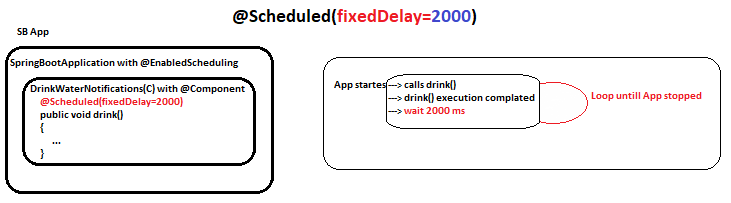
1. **@EnableScheduling** at Application class level
2. **@Scheduled** at method level which need to be repeated.

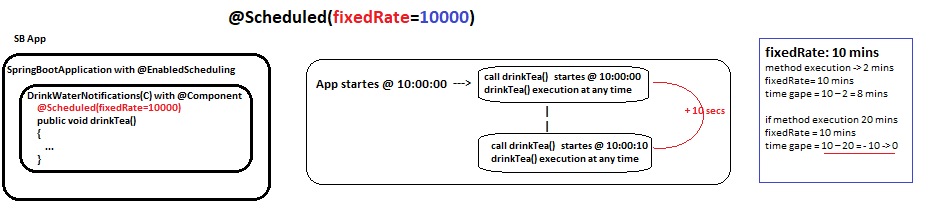
**Project creation steps:**

1. File🡪New🡪Spring Starter Project🡪 Name: **SbApp6\_Scheduling\_fixedDelay**, java11,🡪Net 🡪No dependency required. 🡪Finish
2. Add **@EnableScheduling** annotation at class level in SbApp6SchedulingFixedDelayApplication.java
3. Create DrinkNotifications class with **@Component.** Add public method with **@Scheduled(fixedDelay=2000)** ,

Another method with **@Scheduled(fixedRate=10000)** annotation.

1. Run the App.





**Cron Expression:** **cron** is an expression used to indicate DATE and TIME.

It is introduced at Linux/Unix based OS.

Later used at so many lanuages and APIs.

\*) **Spring Boot Scheduling supports using cron.**

**Syntax:** 6 POSITIONS (with one space gap)

SEC MIN HRS DAY MONTH WEEKDAY

0-59 0-59 0-23 1-31 1-12 or JAN-DEC MON-SUN or 0-7 [0 or 7 is Sunday]

**Symbols Allowed:**

\* = any /all /every

, = possible values. **Ex:**  5,10

* = Range **Ex:** 5-10

? = Used only for Day , WeekDay when month is provided.

/ = indicates period of time.

Examples:

1. 0 0 9 \* \* \* 🡪Execute task, Every Day 9:00:00 AM
2. 0 0 18 \* \* \* 🡪Execute task, Every Day 6:00:00 PM
3. 0 0 10,20 \* \* \* 🡪 Execute task, Every Day 10:00:00 AM and 08:00:00 PM
4. 59 59 23 \* \* \* 🡪 Execute task, Every Day 11:59:59 PM
5. 0 0 0,12 \* \* \* 🡪 Execute task, Every Day 00:00:00 AM and 12:00:00 PM
6. 0 0 \* \* \* \* 🡪 Execute task, Every Hour 0th min & 0th sec.
7. 0 30 \* \* \* \* 🡪 Execute task, Every Hour 30th min & 0th sec
8. 1 1 \* \* \* \* 🡪 Execute task, Every Hour 1st min & 1st sec.
9. 0 \* \* \* \* \* 🡪 Execute task, Every Minute 0th sec
10. 10 \* \* \* \* \* 🡪 Execute task, Every Minute 10th sec (Not 10 sec gap)
11. 0 0 20 \* \* SUN 🡪 Execute task, Every Sunday 08:00:00 PM

Examples:

1. cron = 10 \* \* \* \* \* 🡪 Execute every minute 10th sec [Point of time]
2. cron = \*/10 \* \* \* \* \* 🡪 For every 10 sec gap [Period of time]
3. cron = \* \*/10 \* \* \* \* 🡪 For every 10 mins gap [Period of Time]
4. cron = \* \* \* \* \* \* 🡪 Execute task, every sec
5. **cron** = **0 0 9 \* \* \* \*** 🡪**IllegalStateException… Cron expression must consist o 6 fields( found 7 )**
6. cron = 0 0 9 1 2 ? 🡪 Execute task, Every year 1st Feb 09:00:00 AM

**? is used for DAY and WEEK DAY POSITION, when Month is provided**

1. cron = 0 0 9 12 \* \* (Valid) 🡪 Execute task, Every Month 12th Day 09:00:00 PM
2. cron = 0 0 9 \* 2 \* 🡪 Feb every Day at 09:00:00 AM
3. cron = 0 0 9 ? 2 ? 🡪 Feb every Day at 09:00:00 AM

Special cases:

cron = @hourly

**cron = “@yearly”** or **cron = “@annually”** 🡪 once a year “0 0 0 1 1 \*”

**cron = “@monthly”** 🡪 once a month “0 0 0 1 \* \*”

**cron = “@weekly”**  🡪 one a week “0 0 0 \* \* 0”

**cron = ”@daily” or cron = ”@midnight” 🡪** once a day “0 0 0 \* \* \*”

**cron = “@hourly” 🡪** once an hour "0 0 \* \* \* \*”

**Last Day [L] :hnce an hour --> -->dExecute task, every sec. in .arly loading in spring.**

0 0 0 L \* \* 🡪 Last Day of Every Month 00:00:00: AM or midnight

0 0 0 L-3 \* \* 🡪 third-to-last day of the month midnight

0 0 0 \* \* 5L 🡪 Last Friday of the month midnight

0 0 0 \* \* THUL 🡪 last Thursday of the month at midnight

0 0 9 1 3 FRI,SUN

0 0 9 1 MAR FRI,SUN

0 0 9 1 MAR 5,7 or 0 0 9 1 MAR 5,0

Execute given task every year March 1st 9:00:00 if it is Friday or Sunday.

0/7 SUN

1 MON

2 TUE

3 WED

4 THU

5 FRI

6 SAT

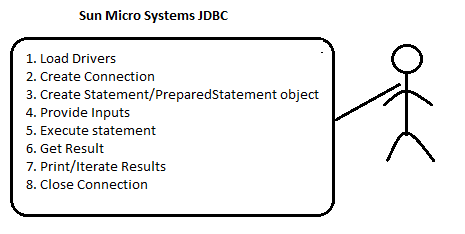
**Ref:**

<https://spring.io/blog/2020/11/10/new-in-spring-5-3-improved-cron-expressions>

**Spring Boot Data JDBC**

Database Operations 🡪 JDBC🡪 Spring JDBC 🡪 Spring Boot JDBC **[Simplified]**

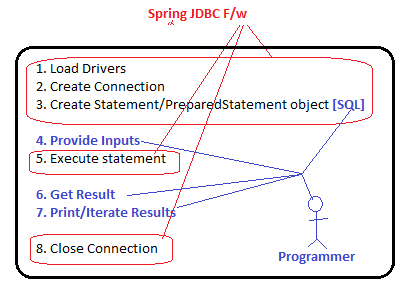
**JDBC:** all the code written by Programmer.

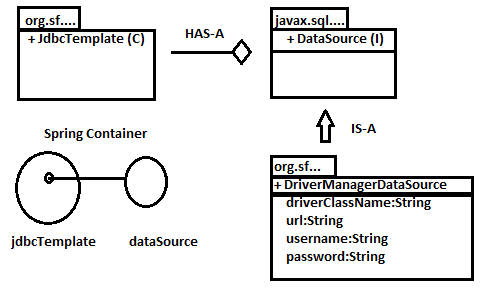


**Spring JDBC:** Spring JDBC API takes care of common work by using **JdbcTemplate** (Template Design Pattern).

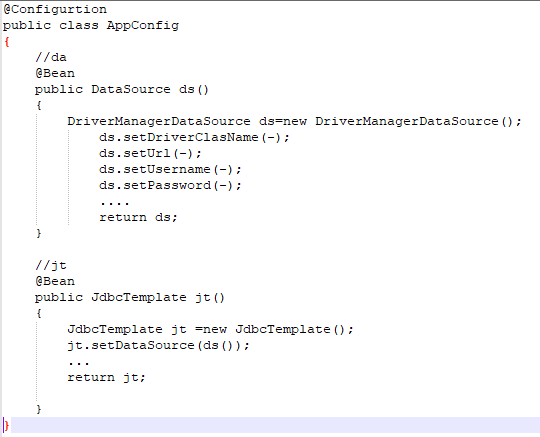
**Template:** It is design pattern to reduce the common lines of code.

**Jdbc API + Template DP= JdbcTemplate (C)**





**Spring Java Configuration Code for JDBC:** The below code required from programmer.



**Spring Boot JDBC (Auto-Configuration):**

**JdbcTemplate, DataSource …. etc.** objects are created and configured by Spring Boot automatically when user gives required inputs in application.properties or application.yml and required spring-boot-starter-data-jdbc dependencies in pom.xml.

Programmer has to provide required inputs through **application.properties** or **application.yml**

**Download and install Mysql:**

1. Download mysql installer from [**MySQL :: Download MySQL Installer**](https://dev.mysql.com/downloads/installer/)and **Install (Double Click) 🡪 Custom🡪Server,Workbench,Connector,Shell & document 🡪Next 🡪Config Type: Developer Computer,**

**Port:3306, X Protocal Port :** **33060 , Root Pwd: root/root, new user: user/user 🡪windows server name:** **MySQL80🡪 Execute 🡪 Finish🡪Next🡪 check connection by providing root pwd.**

1. **Workbench setup: connect workbench local instance by root pwd.**

**>create database testdb;**

>show databases;

**>use testdb;**

**>**

**CREATE TABLE `testdb`.`emp` (**

**`eid` INT NOT NULL,**

**`ename` VARCHAR(45) NOT NULL,**

**`esal` DOUBLE NULL,**

**PRIMARY KEY (`eid`))**

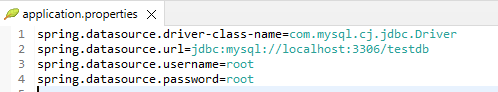
**COMMENT = 'sample emp table with eid, ename, esal columns';**

**>select \* from emp;**

**>insert into emp values(102,'ravi',35000.00);**

**Sample Spring Boot project creation:**

1. **MySql** should be installed and **testDB -** database should be created
2. **File** 🡪 **New** 🡪**Spring Starter Project** 🡪 **SbApp7\_JDBC\_Ds\_Jt\_Objects\_verify** 🡪 **java 11, Spring Data JDBC, Lombok, MySql Driver** 🡪 **Finish.**
3. **Update application.properties as below**



1. Create **JdbcConnectionCheckRunner** implementing CommandLineRunner with @Component: DataSource,JdbcTemplate should be @Autowired.

Apply **@PostConstruct** on method to inject static fields as @Autowired will not work on static fields.



1. Run the App. **Output: 🡪**

**Welcome to Spring Boot JDBC**

**HikariDataSource (HikariPool-1)**

[**org.springframework.jdbc.core.JdbcTemplate@5f96f6a2**](mailto:org.springframework.jdbc.core.JdbcTemplate@5f96f6a2)

**satic fields initialize with PostConstruct staticJt org.springframework.jdbc.core.JdbcTemplate@134ff8f8**

**satic fields initialize with PostConstruct staticDs HikariDataSource (HikariPool-1)**

**Q) How to check which variable related to which property in application.properties file.**

Ctrl+click on the property in .properties file.

**Q) Q) How to identify the Driver class for mySql, oracle.... etc?**

Ctrl+shit+t ---> search Driver

Or

Go to Maven dependencies🡪 mysql-connectr….jar🡪 META-INF🡪Services🡪open java.sql.Driver🡪

com.mysql.cj.jdbc.Driver

\*) Auto-Loading or Auto-Detecting Driver class is available in Sun JDBC 4.0 API based on spring.datasource.url value

**Q) Is DriverClassName key=value optional in properties?**

Yes. JDBC Auto-Loading Driver class [Sun JDBC 4.0]

Based on URL and Driver Jar details (mysql--8.0.jar)

**JDBC Operation can be performed by using JdbcTemplate object:**

1) Non-Select Operations:Insert, Delete, Update

2) Select Operations: select

**Non-Select operations :Insert, Delete, Update**

update(SQL:String, args:Object...):int

This method is defined in JdbcTemplate(C) that support non select operation(insert,update,delete).

Ref: EmpJdbcOperationsRunner.java

**@Component:** used to create spring bean object by container.

**@Repository:** Is subtype of Component with DB operations,Exception Handling & Transaction support.

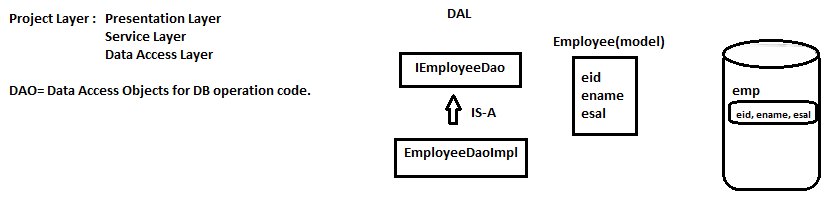
**POJI-POJO Design Pattern:**

**POJI**- Plain Old Java Interface

**POJO-** Plain Old Java Object (Implementation class Object)

Interface reference = Implementation class Object

**i.e** IEmployeeDao dao= new EmployeeDaoImpl();



To be continued…..

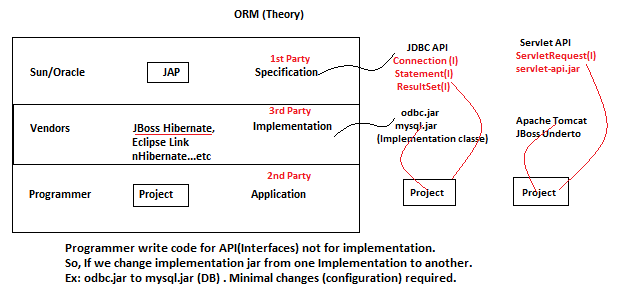
**ORM with Hibernate ( Database Operations)**

**Theory 🡪ORM**

**API(Specification) 🡪JPA(Java Persistency API)**

**Vendor 🡪 Hibernate**

**Pre-Defined code : Spring Boot Data JPA**

****

**ORM:** Object Relational Mapping (Theory): Perform all db operations using OOPs concepts.

**Everything should be in object format. Programmer must follow mapping Rules.**

**Mapping Rules:**

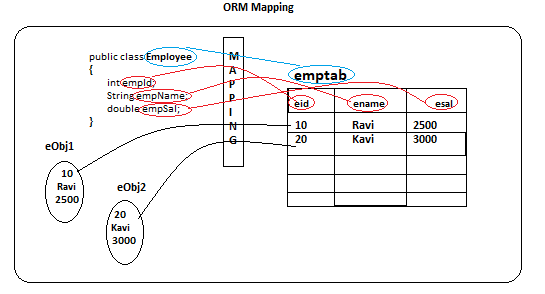
1 class --------🡪 Mapped --🡪 1 Table

1 variable ----🡪 Mapped --🡪 1 column

Then [ORM]

1 Object 🡨 ---------Mapped -------🡪 1 Row

**>> No sql query need to be defined by programmer.**



**Sun Microsystem/Oracle has given one API-JAP**

It is a specification. Given a few interfaces, enums, Annotations… etc.

**Persistency: Something that is connected with DB to perform fetch, save, update….etc operations.**

**Persist()🡪 INSERT**

**>> Spring Data JAP: Internally uses Hibernate.**

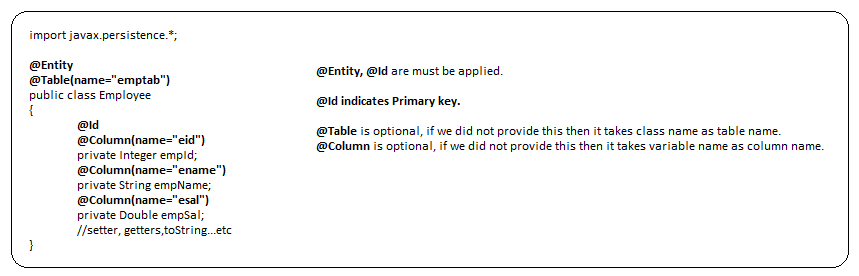
**Coding Files:**

1. **Entity:** class that is mapped with database table.
2. **Persistence XML File** ( Database Connection details, ORM keys…. Etc)
3. **Test class**

**JAP Annotations:- are used in Entity class.**

Ref: <https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html>

1. **Entity class**

****

1. **Persistence XML File** ( Database Connection details, ORM keys…. Etc)

**JPA.Hibernate Keys**

1. **Database Connection Keys**
2. Driver-class
3. url
4. username
5. password
6. **ORM Keys**
7. **dialect** : Is a class that generate all SQLs when we perform DB operations

SQL queries are DB dependent. i.e SQL queries are different for Oracle, MySql, Sybase, Postgress …etc

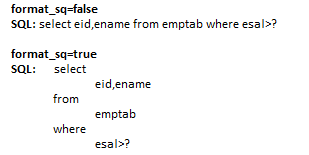
Ref: org.hibbernage.dialect.\*

<https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/dialect/package-summary.html>

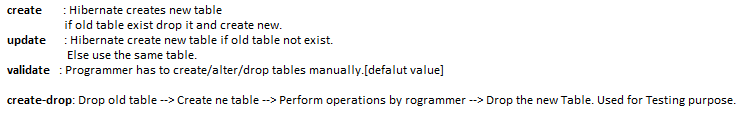
2**. show-sql:** (boolean false)

Use show\_sql=true to view generated SQL at console.

3**. format\_sql:**(boolean - false)



**4. hbn2ddl.auto** --- it has 4 possible values

****

**Hibernate with JPA Application**

**Coding Steps**

Open Mysql workbenach 🡪Connect Local instance root/loclhost:3306

create database scdb;

show databases;

use scdb;

show tables;

select \* from emptab;

1. Create maven project File🡪New🡪Other🡪Maven Project🡪ScApp9\_HibernateWithJPA 🡪Finish
2. Update java 8 compiler plugin, add (hibernate-core, mysql-connector, Lombok) dependencies
3. Create **Employee** class with **@Entity**, @Table at class level, **@Id**, @Column at variables.

package in.rk.sc.bean;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import lombok.Data;

@Data

@Entity

@Table(name = "emptab")

public class Employee

{

    @Id

    @Column(name = "eid")

    private Integer empId;

    @Column(name = "ename")

    private String empName;

    @Column(name = "esal")

    private Double empSal;

}

1. Create folder **META-INF** inside resources. Create persistence.xml in META-INFO folder. Copy &Paste the connect in persistence.xml file

<? xml version = "1.0" encoding = "UTF-8" ?>

    <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence

        http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">

        <persistence-unit name="AppDB">

            <properties>

<!-- datasource/db properties -->

                <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />

                <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/scdb" />

                <property name="javax.persistence.jdbc.user" value="root" />

                <property name="javax.persistence.jdbc.password" value="root" />

<!-- ORM properties -->

                <property name="hibernate.show\_sql" value="true" />

                <property name="hibernate.hbm2ddl.auto" value="create" />

                <property name="hibernate.dialect" value="org.hibernate.dialect.MySQL8Dialect" />

            </properties>

        </persistence-unit>

    </persistence>

1. Check scdb is available in mysql database, and root/root is valid.
2. Create AppTest.java class to do
   1. Create EntityManagerFctory by passing persistent unit nameLoad Driver, Create Connection, Prepare statement
   2. Create EntityManager object from Factory(To do DB operation)
   3. Get Transaction object form EntityManager to Begin Transaction
   4. Start Transaction
   5. Prepare Entity Object: emp object
   6. Perform DB operation:
   7. Commit Transaction
   8. Close EntityManagerFactory
   9. Rollback Transaction if any Exception

package in.rk.sc.bean;

import javax.persistence.EntityManager;

import javax.persistence.EntityManagerFactory;

import javax.persistence.EntityTransaction;

import javax.persistence.Persistence;

public class AppTest {

    public static void main(String[] args) {

        EntityTransaction tx = null;

        try {

            // 1. Create EntityManagerFctory by passing persistent unit name: Load Drivers,

            // Create DB connection, Prepare statement

            EntityManagerFactory emf = Persistence.createEntityManagerFactory("AppDB");

            // 2. Create EntityManager object from Factory(To do DB operation)

            EntityManager et = emf.createEntityManager();

            // 3. Get Transaction object form EntityManager to Begin Transaction

            // 4. start Transaction

            tx = et.getTransaction();

            tx.begin();

            // 5. Prepare Entity Object: emp object

            Employee emp = new Employee();

            emp.setEmpId(100);

            emp.setEmpName("Ravi");

            emp.setEmpSal(5000.00);

            // 6. Perform DB operation:

            et.persist(emp);

            //7. Commit Transaction

            tx.commit();

            //8. Close EntityManagerFactory

            emf.close();

        } catch (Exception e) {

            //9. Rollback Transaction if any Exception

            tx.rollback();

            e.printStackTrace();

        }

    }

}

1. Run AppTest

* The above step all are not required in Spring Boot Data JPA. Minimal Coding in SB Data JPA.

**Ref: ScApp9\_HibernateWithJPA project**

**JAP with Hibernate:**

All annotations are given by JPA(Java Persistency AP) from package: **javax.persistence.\***

**@Entity:**  It maps java class with DB table and variables with columns.

**@Id:** It indicates Primary Key (unique + not null) variables /columns.

**>> In JDBC Programming PK is optional but not in JPA.**

**@Table:** It is optional. If we did not specify Table name class name is taken as Table name.

**@Column:** It is optional. If we did not specify column name then variables name is taken as column name.

**@Data:**  It is Lombok annotation. It generates set/get methods, toString, constructor…etc.

**persistence.xml :** It is mandatory file must be located in src/main/resources/META-INF folder.

Else Could not find any META-INFO/persistence.xml Exception will be thrown

Contains the below properties

[Persistence-unit] – any name

*javax.persistence.jdbc.driver =*

*javax.persistence.jdbc.url =*

*javax.persistence.jdbc.user =*

*javax.persistence.jdbc.password =*

*hibernate.show\_sql =*

*hibernate.format\_sql =*

*hibernate.hbm2ddl.auto =*

*hibernate.dialect =*

>> Old Hibernate Versions (3.x) are not following JPA standards. Uses **hibernate.cfg.xml** file

//Load driver class, Create DB connection, Statement, select dialect , Table creation

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("AppDB");

//Operation INSERT, UPDATE

EntityManager et = emf.createEntityManager();

//Either commit od success or rollback on exception

EntityTransaction tx;

//INSERT operation

et.persist(emp);

**>> Once we go to Spring Boot, we don’t write the above code and xml file.**

**Spring Boot Data JPA**

**Spring Boot Data JPA Dependency in pom.xml**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>**spring-boot-starter-data-jpa**</artifactId>

</dependency>

**application.properties**

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

>> **spring-boot-starter-data-jpa (jar):** that provides an Interface **JpaRespository (I)**

It provides basic methods for DB operations such as insert, update, delete, select all, select one, count… etc

>> We need not to write code here. Code exist in **SimpleJpaRespository(C)**

>> We just define one interface that extends **JpaRepository** and provide two generic inputs Entity Name(T) and PK Data Type (ID)

>> Custom methods can also be defined.

Spring Boot Data JPA Example code

1. File🡪New🡪Spring starter project🡪 SbApp8a\_DataJPA , java 11,**Data JPA, MySql Connector, Lombok** 🡪Finish
2. Open application.properties file and add database connection properties and jpa properties

#Database properties

#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.show-sql=true

#spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=create

# driver-class-name and database-platform are optional based on url those will be detected

>> Both **driver-class-name** and **databse-platform** are optional

1. Create Entity class : Product with JPA annotations.

package com.example.demo.entity;

import javax.persistence.\*;

import lombok.Data;

@Data

@Entity

@Table(name = "prodtab")

public class Product {

    @Id

    @Column(name = "pid")

    private Integer prodId;

    @Column(name = "pname")

    private String prodName;

    @Column(name = "pcost")

    private Double prodCost;

}

1. Create Repository Interface which extends JpaRepository interface.

package com.example.demo.repository;

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

import com.example.demo.entity.Product;

public interface ProductRepository extends JpaRepository<Product, Integer>

{

}

1. To test application create a CMD Runner ProductTestRunner by implementing CommandLineRunner

**save(-)**: behave like INSERT or UPDATE.

1st it takes @Id (PK) calls select by Id if object available UPDATEs else INSERTs.

**saveAll(-):** will inserts or updates multiple object in a single network calls(Batch insert or update).

**findAll():** like SELECT \* FROM <TableName>. return List<T>

**findById(-):** return optional<T>

**existsById(id):** return Boolean

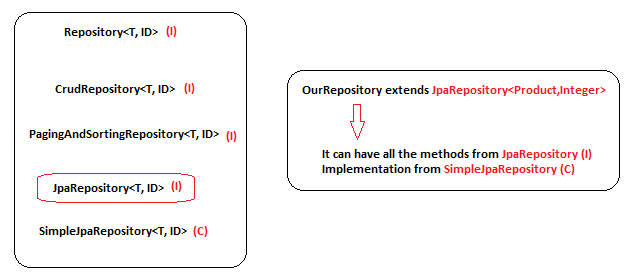
**count():** return long

**deleteById(id)**:void

**deleteAll():** To delete all rows one by one

**deleteAllInBatch():** To delete all rows at a time

**saveAndFlush(-):** saves and returns saved object



1. Database commands

show databases;

use testdb;

show tables;

select \* from prodtab;

**Pagination**

**Pagination:** Fetching data from DB table by making it into multiple equal parts.(i.e called as pages)

**>> Pagination logic is in-**built (Similar to Hibernate Logic) But must provide inputs.

**Page<T> findAll(Pageable)**

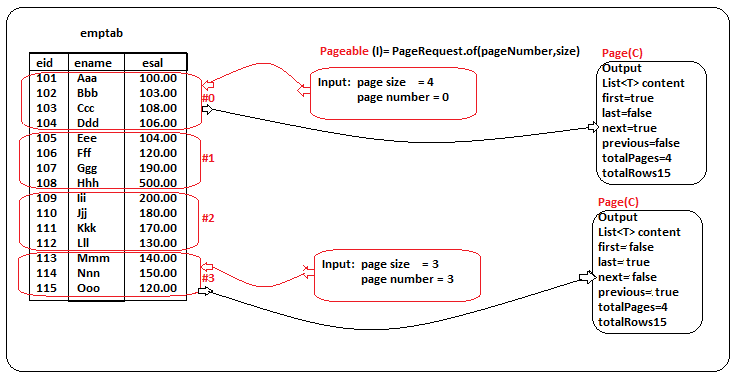
Input: **Pageable(I)** [Page Number, Page Size]

[ **PageRequest (C) ]**

Output: **Page** ( List<T>, meta-data[totalPages,pageSize, next, previous, first,last])

**List<T> findAll():** Returns all rows from DB

**Page findAll(Pageable):** Returns specific Page based on given inputs Pageable(pageNumber, size).



Spring Boot Data JPA Example code

1. File🡪New🡪Spring starter project🡪 SbApp8b\_DataJPA\_Pagination, java 11,**Data JPA, MySql Connector, Lombok** 🡪Finish
2. Open application.properties file and add database connection properties and jpa properties

#Database properties

#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/testdb

spring.datasource.username=root

spring.datasource.password=root

spring.jpa.show-sql=true

#spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto=create

# driver-class-name and database-platform are optional based on url those will be detected

>> Both **driver-class-name** and **databse-platform** are optional

1. Create Entity, Repo and Runner classes:

//Total rows 16, try with page 0,3,4

Pageable pageable=PageRequest.of(0, 5);

Page<Employee> page0=repo.findAll(pageable);

System.out.println(page0);

System.out.println("Page number:"+page0.getNumber());

System.out.println("Page size:"+page0.getSize());

System.out.println("No of elements in page:"+page0.getNumberOfElements());

System.out.println("Total no of elements(all rows):"+page0.getTotalElements());

System.out.println("Total no of pages:"+page0.getTotalPages());

System.out.println("Has Content:"+page0.hasContent());

System.out.println("Has Next:"+page0.hasNext());

System.out.println("Has Preavious:"+page0.hasPrevious());

System.out.println("Is Empty:"+page0.isEmpty());

System.out.println("Is First:"+page0.isFirst());

System.out.println("Is last:"+page0.isLast());

List<Employee>pageList=page0.getContent();

pageList.forEach(System.out::println);

**Q) DB Question: Find the nst highest salary details?**

Mysql> select \* from emptab order by esal desc limit **n-1,1**;

**Q) DB Question: Find the 2nd highest salary details?**

Mysql> select \* from emptab order by esal desc limit **1,1**;

**Q) DB Question: Find the 5th highest salary details?**

Mysql> select \* from emptab order by esal desc limit **4,1**;

**Hibernate Annotations**

**Date and Time:**

**@Temporal** with **TemporalType.DATE or TIME or TIMESTAMP** property.

**@Temporal** can be applied on java.util.Date variable.

If we did not specify any TemporalType.by default TIMESTAP is taken.

**Complex Types:** (Lobs - Large Objects):

**BLOB= Binary Large object🡪** Image, Audio, Video, PDF, Document…etc

**byte[]** + **@Lob**

**CLOB= Character Large Object🡪**Large Test

**char[] + @Lob**

@Temporal(TemporalType.DATE)

@Column(name = "doj")

private Date doj;

@Temporal(TemporalType.TIME)

@Column(name = "shift\_time")

private Date shiftTime;

@Temporal(TemporalType.TIMESTAMP)

@Column(name = "login\_time")

private Date loginTime;

@Column(name = "empsal")

private Double empSal;

@Lob //BLOB

@Column(name = "emp\_img")

private byte[] empImg;

@Lob //CLOB

@Column(name = "emp\_desc")

private char[] empDesc;

FileInputStream fis= new FileInputStream("C:\\Users\\LENOVO\\Downloads\\ravi.jpg");

byte[] imgBArray= fis.readAllBytes();

char[] descCArray= "Hello World. How are you, What are you doing?".toCharArray();

Employee emp = new Employee(100, "ravi", new Date(), new Date(), new Date(), 20000.00, imgBArray, descCArray);

repo.save(emp);

fis.close();

**>> SQL Yog : a UI tool for MySql (GUI): Install from** [**https://sqlyog.en.softonic.com/download**](https://sqlyog.en.softonic.com/download)

**>> Now a days, Files or Images are not stored in DB. It is recommended to store in File System.**

**Attribute Converter**

Java 8 brought lots of great features and one of the most important and most anticipated ones was the new Date and Time API.

Attribute Converter is required to convert one type to another type vice versa while inserting and retrieving data.

It can be applied automatically.

java.time.LocalDateTime 🡨 LocalDateTimeAttributeConverter 🡪 java.sql.Date

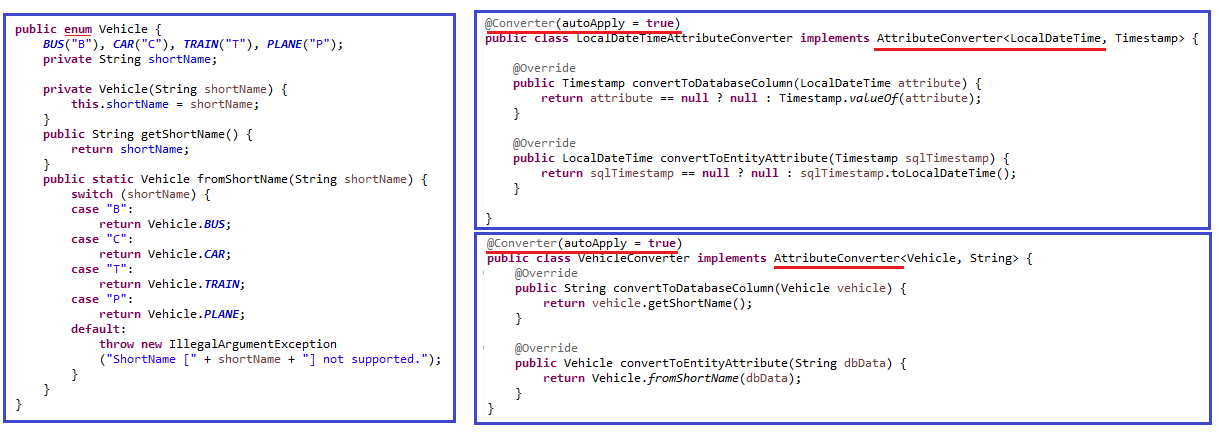
java.time.LocalDate 🡨 LocalDateAttributeConverter 🡪 java.sql.Timestamp

Enum Type (Vehicle) 🡨 VehicleConverter 🡪String Type

**Steps:**

1. Create a converter class by implementing **AttributeConverter**(Interface)
2. Override the unimplemented methods(abstract methods)
3. Apply **@Converter(autoApply=true)** at class level.
4. Create Entity class by defining Java types that will supposed to store in DB as DB types.

****

****

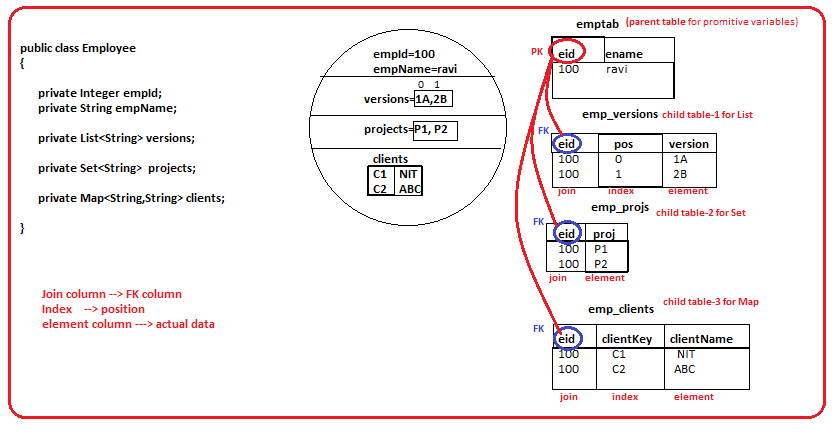
**Ref: SbApp8c\_DataJPA\_HibernateAnnotations Project**

**Collection Mapping & Association Mapping**

**Collection Mapping (Collections with Primitives):** List<String> , Set<Integer>, Map<String,String>

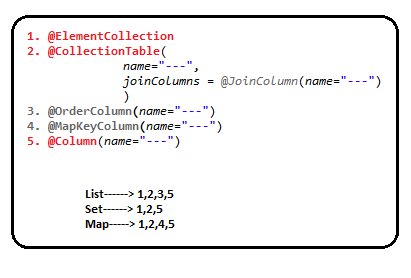
For one collection variable 🡪 1 child table is created with 2 or 3 columns.

1. **List, Map** 🡪 Child Table with 3 columns (**Join column**(Foreign key), **Index/order** column, **Element** column)
2. **Set** 🡪Child table is created with 2 columns (**Join column(FK)**, **Element column**).



>> **@ElementCollection** is JPA annotation must be applied on collections variables.

It is provided by JPA. It creates child tables for every collection variable and links FK column with parent table PK column.



>> Data JPA internally follows normalization rules and gives standard table design.

<https://www.studytonight.com/dbms/database-normalization.php>

SHOW DATABASES;

USE testdb;

SHOW TABLES;

SELECT \* FROM emptab;

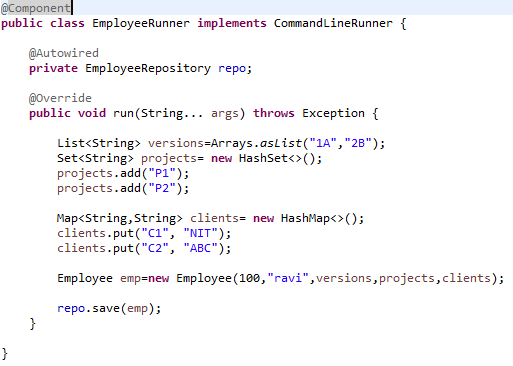
SELECT \* FROM emp\_versions;

SELECT \* FROM emp\_projects;

SELECT \* FROM emp\_clients;

**Entity class with Collections with Primitives [Employee.java] & EmployeeRunner.java**





**Ref: SbApp8d\_DataJPA\_HibernateElementCollection Project**

**Association Mapping:** (1…1/1…\*/\*….1/\*….\*) **and joins** (Multiple tables).

>> Implementing **HAS-A** relation between two or more entities is called Association Mapping.

1….1 , 1….\*, \*….1, \*…..\*

**One-to-One: 1… 1** Ex: Car Has-a Engine  **@OneToOne**

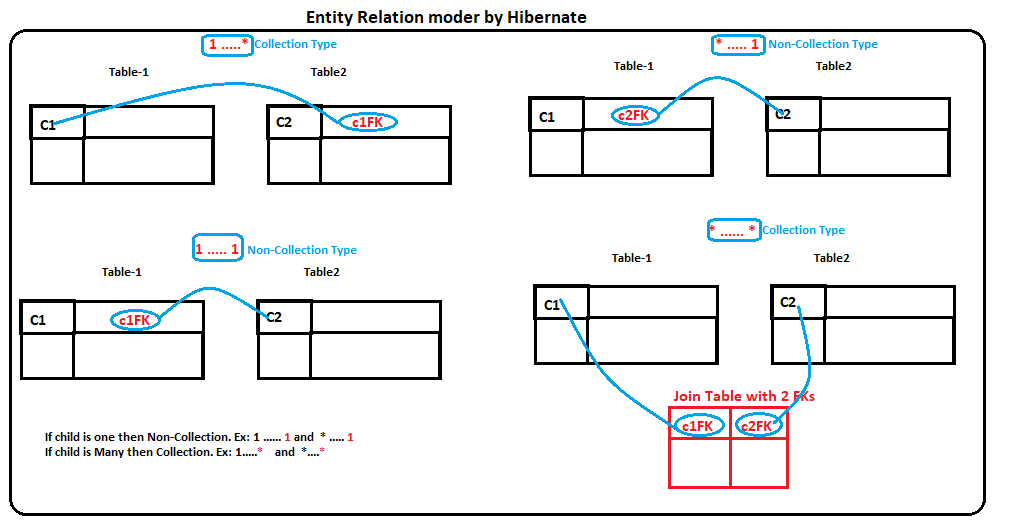
**One-to\_Many: 1…\*** Ex: Department Has-A Employees **@OneToMany**

**Many-to-One:** **\*…. 1** Ex: IdCards Has-A Employee **@ManyToOne**

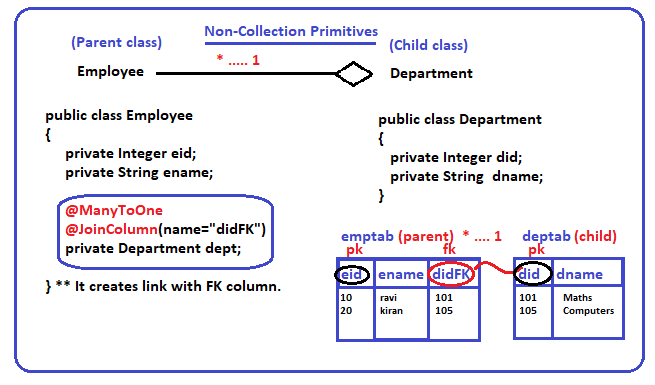
**Many-to-Many: \*….\*** Ex: Student Has-a Course (1…\* or \*…1 or \*…\*) **@ManyToMany**

>> Two tables are created and connected. One table PK is connected with another tables FK.

>> Based on relation FK is created at selected table.



1. >> **Many-to-one (\*…… 1) :** Keep association annotations in Parent class.



**Coding Steps:**

1. **Define two entity classes**
2. **Create Has-a variable in Parent class**
3. **Check for Collection or Non-Collection**
4. **Apply Multiplicity Annotation at Has-a variable.**

**@OneToOne**

**@ManyToOne**

**@OneToMany**

**@ManyToOne**

1. **Add JoinColumn/ JoinTable[\*…\*] at Has-avariable**
2. **Check Tables in DB**

**Ref:** SbApp8e\_DataJPA\_ManyToOne Project

drop databse testdb;

create database testdb;

use testdb;

show tables;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="depttab")

public class Department {

    @Id

    @Column(name="did")

    private Integer deptId;

    @Column(name="dname")

    private String deptName;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="emptab")

public class Employee {

    @Id

    @Column(name="eid")

    private Integer empId;

    @Column(name="ename")

    private String empName;

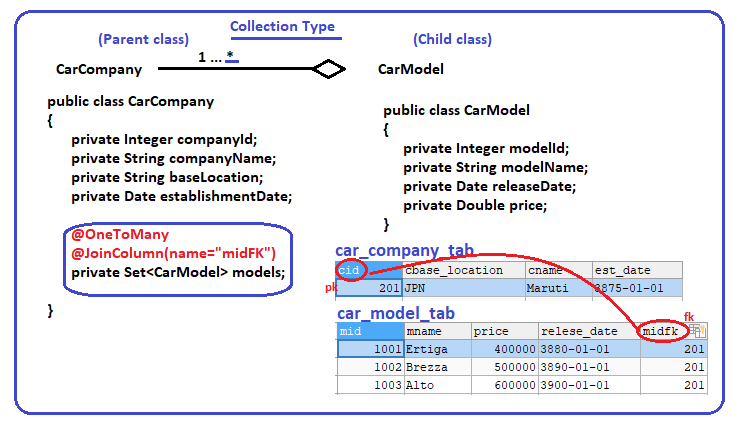
    @ManyToOne

    @JoinColumn(name="didFK")

    private Department dop;//Has-a Relation ManyToOne

}

1. **>> One-To-Many ( 1 .... \*):**

****

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="car\_model\_tab")

public class CarModel {

    @Id

    @Column(name="mid")

    private Integer modelId;;

    @Column(name="mname")

    private String modelName;

    @Temporal(TemporalType.DATE)

    @Column(name="relese\_date")

    private Date releaseDate;

    @Column(name="price")

    private Double price;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="car\_company\_tab")

public class CarCompany {

    @Id

    @Column(name="cid")

    private Integer companyId;

    @Column(name="cname")

    private String companyName;

    @Column(name="cbase\_location")

    private String baseLocation;

    @Temporal(TemporalType.DATE)

    @Column(name="est\_date")

    private Date establishmentDate;

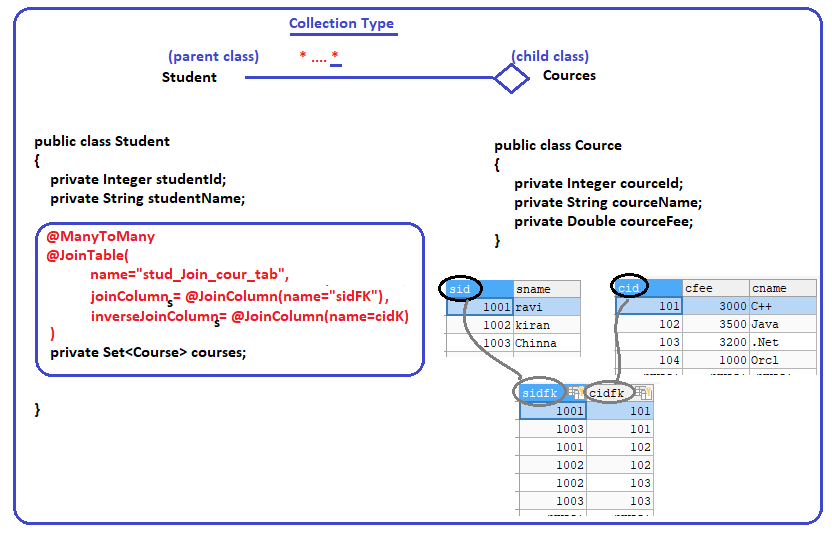
    @OneToMany

    @JoinColumn(name="midFK")

    private Set<CarModel> models;

}

1. **>> Many-To-Many (\* … \*):**

****

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name = "studenttab")

public class Student {

    @Id

    @Column(name = "sid")

    private Integer studentId;

    @Column(name = "sname")

    private String studentName;

    @ManyToMany

    @JoinTable(

            name = "stud\_join\_cour\_tab",

            joinColumns = @JoinColumn(name = "sidFK"),

            inverseJoinColumns = @JoinColumn(name = "cidFK")

    )

    private Set<Cource> cources;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name = "courcetab")

public class Cource {

    @Id

    @Column(name = "cid")

    private Integer courcetId;

    @Column(name = "cname")

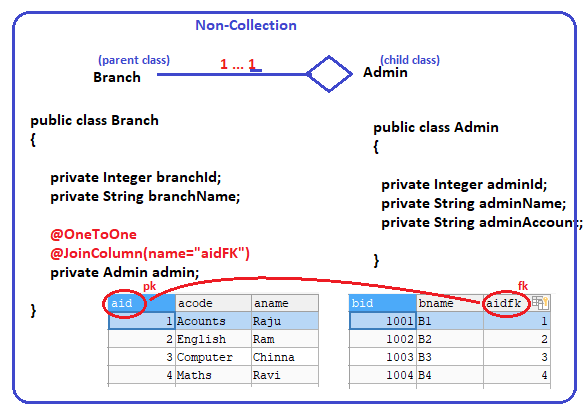
    private String courceName;

    @Column(name = "cfee")

    private Double courceFee;

}

1. **>> One-To-One ( 1….1)**

****

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="admintab")

public class Admin {

    @Id

    @Column(name="aid")

    private Integer adminId;

    @Column(name="aname")

    private String adminName;

    @Column(name="acode")

    private String adminCode;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="branchtab")

public class Branch {

    @Id

    @Column(name="bid")

    private Integer branchId;

    @Column(name="bname")

    private String branchName;

    @OneToOne

    @JoinColumn(name="aidFK")

    private Admin admin;

}

**Spring Boot Data JPA with Custom Query Programming**

Writing programmer defined query and get executed by Data JPA is called **Custom Query**.

We can pass inputs and fetch required data.

There are two concepts for Custom Query:

1. **@Query(“JPQL/HQL”)**  🡪 Select and non-select operations (support SQL queries too)
2. **findBy 🡪** Select operations

**SQL:** SQL queries are data base dependent. Defined SQL query may work for one DB and may not work for another DB.

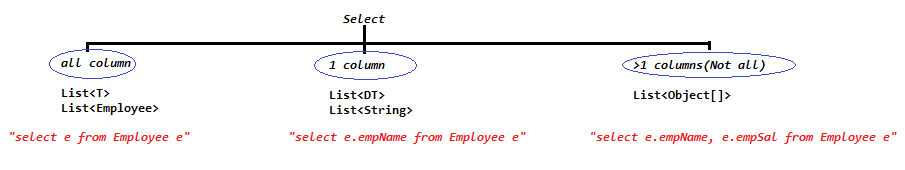
SQL is created by using table name, column names…(case insensitive)

* **select ename from emptab where eid>10;**

**JPQL/HQL:** Java Persistency Query Language/Hibernate Query Language is data base independent because dialect.

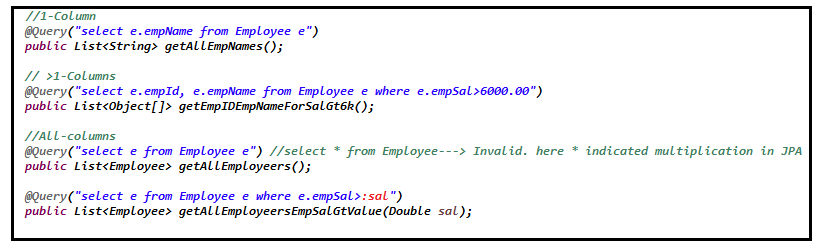
JPQL/HQL is created by using class and variable names…(case sensitive)

* **select empName from Emloyee where empId>10;**



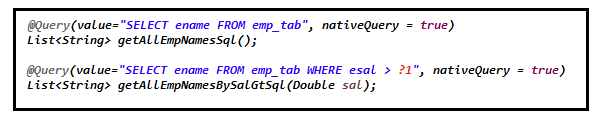
* We just define abstract methods with @Query(--) in Repository interface and implementation provided by Data JPA.

We can specify the parameters by using **:name** and pass using method param at Abstract method.



* We can specify pure SQL queries too.(Not recommended)

**@Query( value=”--sql query--“ nativeQuery=true)** and can pass parama for ?1 ?2 ?3.. etc

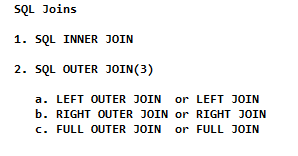
****

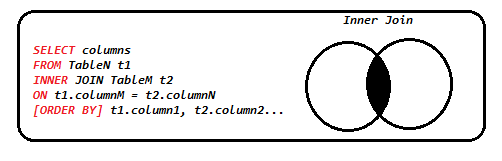
**Spring Data JPA - SQL JOINS**

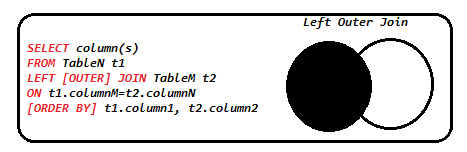
**Ref: Joins**

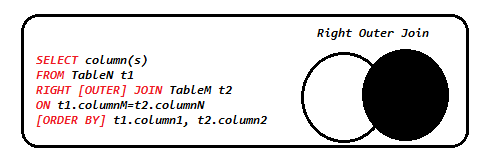
[**https://www.techonthenet.com/sql/joins.php**](https://www.techonthenet.com/sql/joins.php)

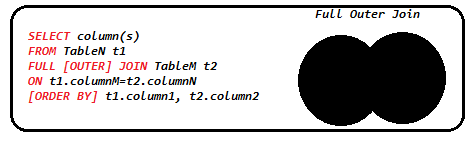
[**https://www.youtube.com/c/NareshIT/search?query=SQL%20joins**](https://www.youtube.com/c/NareshIT/search?query=SQL%20joins)

****

****

****

****

****

**Project Creation Steps:**

1. STS 🡪 File🡪New🡪Starter Spring Project🡪 **SbApp9a\_DataJPA\_CustomQuery** 🡪 Data JPA, MySql Deiver & lombok🡪Next 🡪Finish
2. Add required DB and JPA properties in application.properties file.
3. Create Employee, Department, Product,ProdDetai entity class, EmployeeRepo,DepartmentRepo,ProductRepository,ProdDetailRepository interface by implementing JpaRespositoty.
4. Create TestRunner by implementing CommandLineRunner.

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="emp\_tab")

public class Employee {

    @Id

    @Column(name="eid")

    private Integer empId;

    @Column(name="ename")

    private String empName;

    @Column(name="esal")

    private Double empSal;

    @ManyToOne

    @JoinColumn(name = "didFK")

    private Department dop;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="dep\_tab")

public class Department {

    @Id

    @Column(name="did")

    private Integer depId;

    @Column(name="dname")

    private String depName;

}

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

    //1-Column

    //JPQL/HQP:  select e.empName from Employee e

    @Query("select e.empName from Employee e")

    public List<String> getAllEmpNames();

    // >1-Columns

    //JPQL/HQL: select e.empId, e.empName from Employee e where e.empSal>6000.00

    @Query("select e.empId, e.empName from Employee e where e.empSal>6000.00")

    public List<Object[]> getEmpIDEmpNameForSalGt6k();

    //All-columns

    //JPQL/HQL: select e from Employee e

    @Query("select e from Employee e") //select \* from Employee---> Invalid. here \* indicated multiplication in JPA

    public List<Employee> getAllEmployeers();

    //Dynamic Inputs through :param

    //JPQL/HQL: select e from Employee e where e.empSal>:sal

    @Query("select e from Employee e where e.empSal>:sal")

    public List<Employee> getAllEmployeersEmpSalGtValue(Double sal);

    //==========SQL(Not recommended, params can be passes ?1, ?2....etc===========

    @Query(value="SELECT ename FROM emp\_tab", nativeQuery = true)

    List<String> getAllEmpNamesSql();

    @Query(value="SELECT ename FROM emp\_tab WHERE esal > ?1", nativeQuery = true)

    List<String> getAllEmpNamesBySalGtSql(Double sal);

    // ========JOINS when HAS-A relation in Tables =========

    //Fetch Employees by Department iD.

    @Query("select e.empName,d.depName from Employee e INNER JOIN e.dop as d ON d.depId =:did")

    public List<Object[]> getEmpIdNameDepNameByDid(Integer did);

}

public interface DeparmentRepo extends JpaRepository<Department, Integer> {

}

@Component

public class TestDataRunner implements CommandLineRunner {

    @Autowired

    private EmployeeRepository eRepo;

    @Autowired

    private DeparmentRepo dRepo;

    @Override

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

        Department d1= new Department(10, "QA");

        Department d2= new Department(20, "UAT");

        Department d3= new Department(30, "ADMIN");

        Department d4= new Department(40, "ACCT");

        dRepo.save(d1);

        dRepo.save(d2);

        dRepo.save(d3);

        dRepo.save(d4);

        //save a few records

        eRepo.save(new Employee(101,"ravi",5000.00,d1));

        eRepo.save(new Employee(102,"kavi",6500.00,d2));

        eRepo.save(new Employee(103,"Raju",7500.00,d3));

        eRepo.save(new Employee(104,"Raju",8800.00,d4));

        eRepo.save(new Employee(105,"Raju",8500.00,d1));

        //Query: select empName from Employee

        eRepo.getAllEmpNames().forEach(System.out::println);

        //Query:  select e.empId, e.empName from Employee e where e.empSal>6000.00

        eRepo.getEmpIDEmpNameForSalGt6k().stream().map(each->each[0]+"  "+each[1]).forEach(System.out::println);

        //Query: select e from Employee e

        eRepo.getAllEmployeers().forEach(System.out::println);

        //Query: select e from Employee e where e.empSal>:sal

        eRepo.getAllEmployeersEmpSalGtValue(5999.00).forEach(System.out::println);

        //SQl: SELECT ename FROM emp\_tab

        eRepo.getAllEmpNamesSql().forEach(System.out::println);

        //SQl: SELECT ename FROM emp\_tab WHERE esal > ?1

        eRepo.getAllEmpNamesBySalGtSql(6000.00).forEach(System.out::println);

        //======JOINS======

        eRepo.getEmpIdNameDepNameByDid(10).stream().map(obj->obj[0]+"-"+obj[1]).forEach(System.out::println);

    }

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="prod\_tab")

public class Product {

    @Id

    @Column(name="pid")

    private Integer prodId;

    @Column(name="pcode")

    private String prodCode;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

@Entity

@Table(name="prod\_detail\_tab")

public class ProdDetail {

    @Id

    @Column(name="pcode")

    private String prodCode;

    @Column(name="pname")

    private String prodName;

    @Column(name="pprice")

    private Double prodPrice;

}

public interface ProductRepository extends JpaRepository<Product, Integer> {

    //====inner Join====

    @Query("select p.prodId, pd.prodName, pd.prodPrice from Product p inner join ProdDetail pd on p.prodCode = pd.prodCode")

    public List<Object[]> getDataInnerJoin();

    @Query("select p.prodId, pd.prodName, pd.prodPrice from Product p left outer join ProdDetail pd on p.prodCode = pd.prodCode")

    public List<Object[]> getDataLeftOuterJoin();

    @Query("select p.prodId, pd.prodName, pd.prodPrice from Product p right outer join ProdDetail pd on p.prodCode = pd.prodCode")

    public List<Object[]> getDataRightOuterJoin();

    //@Query("select p.prodId, pd.prodName, pd.prodPrice from Product p full outer join ProdDetail pd on p.prodCode = pd.prodCode")

    //public List<Object[]> getDataFullOuterJoin();

}

public interface ProdDetailRepository extends JpaRepository<ProdDetail, String>{

}

@Component

public class TestProdJoinDataRunner implements CommandLineRunner {

    @Autowired

    private ProductRepository prodRepo;

    @Autowired

    private ProdDetailRepository prodDetailRepo;

    @Override

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

        // store data

        prodRepo.save(new Product(101, "P001"));

        prodRepo.save(new Product(102, "P003"));

        prodRepo.save(new Product(103, "P003"));

        prodRepo.save(new Product(104, "P001"));

        prodRepo.save(new Product(105, "P005"));

        prodDetailRepo.save(new ProdDetail("P001", "Apple", 20.00));

        prodDetailRepo.save(new ProdDetail("P003", "Banana", 10.00));

        prodDetailRepo.save(new ProdDetail("P004", "Anar", 30.00));

        prodRepo.getDataInnerJoin().stream().map(obj -> obj[0] + "-" + obj[1] + "-" + obj[2])

                .forEach(System.out::println);

        prodRepo.getDataLeftOuterJoin().stream().map(obj -> obj[0] + "-" + obj[1] + "-" + obj[2])

                .forEach(System.out::println);

        prodRepo.getDataRightOuterJoin().stream().map(obj -> obj[0] + "-" + obj[1] + "-" + obj[2])

                .forEach(System.out::println);

        //prodRepo.getDataFullOuterJoin().stream().map(obj -> obj[0] + "-" + obj[1] + "-" + obj[2])

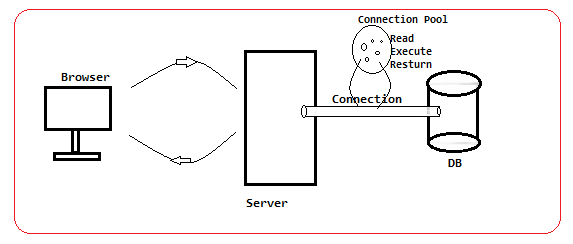
        //      .forEach(System.out::println);

    }

}

**Connection Pool:**  A group of same type of objects.

Connection Pool(List of connection objects) that makes execution faster between server and database.

****

**>> Spring Boot provides auto-configuration for connection pool provider HikariCP (faster as of now)**

Ref: <https://github.com/brettwooldridge/HikariCP>

<dependency>

<groupId>com.zaxxer</groupId>

<artifactId>HikariCP</artifactId>

</dependency>

By Default Spring boot provides HikariCP as connection pool for JDBC and Data JPA.

Ref: HikariConfig class.

#default is 30 sec

spring.datasource.hikari.connection-timeout=10000

# Default is -1 no limit

spring.datasource.hikari.maximum-pool-size=20

# Default is -1 no limit

spring.datasource.hikari.minimum-idle=15

# Pool name

spring.datasource.hikari.pool-name=Hikari-NIT-CP

Q) How can we configure multiple database using Spring (or) Spring Boot?

A) <https://www.youtube.com/watch?v=nzszxQbQ5WU&list=PLVlQHNRLflP9XSWeY4x4FLwnL3UOIxnTr>

Full Code: <https://github.com/javabyraghu/SpringBoot2MultipleDatabases>

**Spting Boot with Multiple DB**

Spring Boot (Data JPA) support one DB connection by default.

DB Connection Configuration (AutoCoinfiguration) is given by SB only but programmers has to provide inputs (properties) in application.properties or application.yml files.

**Q) How to connect multiple DBs with one SB applications?**

> SB doesn’t support multiple DB connections.

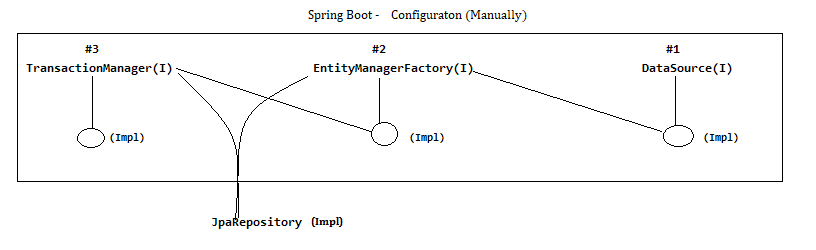
Programmer has to create Spring style (Java Config) +SB application connections to multiple DBs.

We need to define below objects(Beans) Spring Java Configuration Style.

1. DataSource(I)

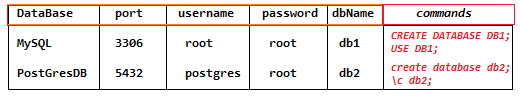
2. EntityMnagerFactory(I)

3. TransactionManger(I)



**Steps:**

1. **Download and install Two databases.**

****

1. **Open STS🡪File 🡪 New 🡪SbApp10a\_DataJPA\_MultiDBs🡪java11,SpringDataJPA,MySQL Driver, PostGreSql Driver,Lombok**

**Dev Tools, Spring Web🡪Next🡪Finish**

1. **Update application.properties files with DB1(MySql)and DB2(PostgreSQL)**

# DB1 ---> MySQL

#DB properies

db1.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

db1.datasource.jdbc-url=jdbc:mysql://localhost:3306/testdb

db1.datasource.username=root

db1.datasource.password=root

# DB2 ---> PostgreSQL

#DB properies

db2.datasource.driver-class-name=org.postgresql.Driver

db2.datasource.jdbc-url=jdbc:postgresql://localhost:5432/testdb

db2.datasource.username=postgres

db2.datasource.password=root

1. **Create Entity classes, Respository interfaces, config classes, Runner, Controller classes.**

Customer.java

**package com.example.demo.db1.entity;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**import lombok.AllArgsConstructor;**

**import lombok.Data;**

**import lombok.NoArgsConstructor;**

**@Data**

**@NoArgsConstructor**

**@AllArgsConstructor**

**@Entity**

**public class Customer {**

**@Id**

**private Integer id;**

**private String name;**

**private String email;**

**}**

Product.java

**package com.example.demo.db2.entity;**

**import javax.persistence.Entity;**

**import javax.persistence.Id;**

**import lombok.AllArgsConstructor;**

**import lombok.Data;**

**import lombok.NoArgsConstructor;**

**@Data**

**@NoArgsConstructor**

**@AllArgsConstructor**

**@Entity**

**public class Product {**

**@Id**

**private Integer id;**

**private String code;**

**private String name;**

**}**

CustomerRepository.java

**package com.example.demo.db1.repo;**

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

**import com.example.demo.db1.entity.Customer;**

**public interface CustomerRepository extends JpaRepository<Customer, Integer> {**

**}**

ProductRepository.java

**package com.example.demo.db2.repo;**

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

**import com.example.demo.db2.entity.Product;**

**public interface ProductRepository extends JpaRepository<Product, Integer>{**

**}**

DbOneConfig.java

**package com.example.demo.db1.config;**

**import java.util.HashMap;**

**import javax.persistence.EntityManagerFactory;**

**import javax.sql.DataSource;**

**import org.springframework.beans.factory.annotation.Qualifier;**

**import org.springframework.boot.context.properties.ConfigurationProperties;**

**import org.springframework.boot.jdbc.DataSourceBuilder;**

**import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;**

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

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

**import org.springframework.context.annotation.Primary;**

**import org.springframework.data.jpa.repository.config.EnableJpaRepositories;**

**import org.springframework.orm.jpa.JpaTransactionManager;**

**import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;**

**import org.springframework.transaction.PlatformTransactionManager;**

**import org.springframework.transaction.annotation.EnableTransactionManagement;**

**@Configuration**

**@EnableTransactionManagement**

**@EnableJpaRepositories(**

**entityManagerFactoryRef = "db1EntityManagerFactory",**

**transactionManagerRef = "db1TransactionManager",**

**basePackages = "com.example.demo.db1.repo"**

**)**

**public class DbOneConfig {**

**// DataSource**

**@Bean**

**@Primary**

**@ConfigurationProperties(prefix = "db1.datasource")**

**public DataSource db1DataSource() {**

**return DataSourceBuilder.create().build();**

**}**

**// EntityManagerFactory**

**@Bean**

**@Primary**

**public LocalContainerEntityManagerFactoryBean db1EntityManagerFactory(EntityManagerFactoryBuilder emfb) {**

**HashMap<String, Object> props = new HashMap<>();**

**props.put("hibernate.hbm2ddl.auto", "create");**

**props.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");**

**return emfb.dataSource(db1DataSource())**

**.packages("com.example.demo.db1.entity")**

**.properties(props)**

**.build();**

**}**

**// TransactionManager**

**@Bean**

**@Primary**

**public PlatformTransactionManager db1TransactionManager(**

**@Qualifier("db1EntityManagerFactory") EntityManagerFactory entityManagerFactory) {**

**return new JpaTransactionManager(entityManagerFactory);**

**}**

**}**

DbTwoConfig.java

**package com.example.demo.db2.config;**

**import java.util.HashMap;**

**import javax.persistence.EntityManagerFactory;**

**import javax.sql.DataSource;**

**import org.springframework.beans.factory.annotation.Qualifier;**

**import org.springframework.boot.context.properties.ConfigurationProperties;**

**import org.springframework.boot.jdbc.DataSourceBuilder;**

**import org.springframework.boot.orm.jpa.EntityManagerFactoryBuilder;**

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

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

**import org.springframework.data.jpa.repository.config.EnableJpaRepositories;**

**import org.springframework.orm.jpa.JpaTransactionManager;**

**import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;**

**import org.springframework.transaction.PlatformTransactionManager;**

**import org.springframework.transaction.annotation.EnableTransactionManagement;**

**@Configuration**

**@EnableTransactionManagement**

**@EnableJpaRepositories(**

**entityManagerFactoryRef = "db2EntityManagerFactory",**

**transactionManagerRef = "db2TransactionManager",**

**basePackages = "com.example.demo.db2.repo"**

**)**

**public class DbTwoConfig {**

**// DataSource**

**@Bean**

**@ConfigurationProperties(prefix = "db2.datasource")**

**public DataSource db2DataSource() {**

**return DataSourceBuilder.create().build();**

**}**

**// EntityManagerFactory**

**@Bean**

**public LocalContainerEntityManagerFactoryBean db2EntityManagerFactory(EntityManagerFactoryBuilder emfb) {**

**HashMap<String, Object> props = new HashMap<>();**

**props.put("hibernate.hbm2ddl.auto", "create");**

**props.put("hibernate.dialect", "org.hibernate.dialect.PostgreSQL10Dialect");**

**return emfb.dataSource(db2DataSource())**

**.packages("com.example.demo.db2.entity")**

**.properties(props)**

**.build();**

**}**

**// TransactionManager**

**@Bean**

**public PlatformTransactionManager db2TransactionManager(**

**@Qualifier("db2EntityManagerFactory") EntityManagerFactory entityManagerFactory) {**

**return new JpaTransactionManager(entityManagerFactory);**

**}**

**}**

DBRunnerTest.java

**package com.example.demo.runner;**

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

**import org.springframework.boot.CommandLineRunner;**

**import org.springframework.stereotype.Component;**

**import com.example.demo.db1.entity.Customer;**

**import com.example.demo.db1.repo.CustomerRepository;**

**import com.example.demo.db2.entity.Product;**

**import com.example.demo.db2.repo.ProductRepository;**

**@Component**

**public class DBRunnerTest implements CommandLineRunner {**

**@Autowired**

**private ProductRepository pRepo;**

**@Autowired**

**private CustomerRepository cRepo;**

**@Override**

**public void run(String... args) throws Exception {**

**//insert DB1**

**pRepo.save(new Product(101,"M\_001","iPhone14"));**

**pRepo.save(new Product(102,"M\_002","OnePlus1"));**

**//insert DB2**

**cRepo.save(new Customer(21,"Ravi","ravi@gmail.com"));**

**cRepo.save(new Customer(32,"kiran","kiran@gmail.com"));**

**//check in DB data inserted or not.**

**}**

**}**

TestMultDbController.java

**package com.example.demo.controller;**

**import java.util.List;**

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

**import org.springframework.web.bind.annotation.GetMapping;**

**import org.springframework.web.bind.annotation.RestController;**

**import com.example.demo.db1.entity.Customer;**

**import com.example.demo.db1.repo.CustomerRepository;**

**import com.example.demo.db2.entity.Product;**

**import com.example.demo.db2.repo.ProductRepository;**

**@RestController**

**public class TestMultDbController {**

**@Autowired**

**private ProductRepository pRepo;**

**@Autowired**

**private CustomerRepository cRepo;**

**@GetMapping("/products")**

**private List<Product> getAllProducts()**

**{**

**return pRepo.findAll();**

**}**

**@GetMapping("/customers")**

**private List<Customer> getAllCustomers()**

**{**

**return cRepo.findAll();**

**}**

**}**

1. **Test Application: Run Application**

[**http://localhost:8080/customers**](http://localhost:8080/customers)

[**http://localhost:8080/products**](http://localhost:8080/products)

**Check DB Tables in Both Databases.**

**Spring Boot Web MVC**

Web application: collection of web pages(resources, images, audio, video) and web pages are two types.

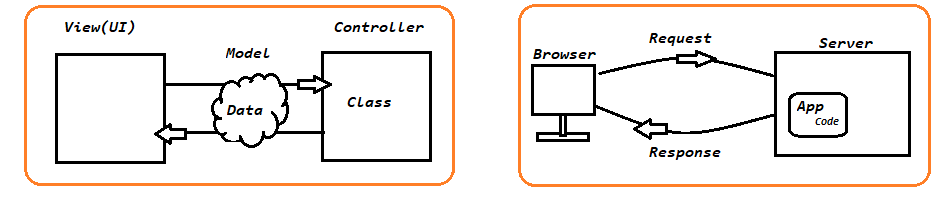
* 1. Static Pages: Data of page always remains same. (HTML)
  2. Dynamic Pages: Data of page get modified / changed based on user, location, time. Java: Servlet

**MVC** is a design pattern used to develop web application.

**M = Model**  ( Data - Primitive, Object, Collections )

**V = View** (Present it to Enduser , UI)

**C = Controller** ( class / request processing logic)



Q) How can we select Controllers?

A)

Controller 🡪 Class#methods

🡪 It will process the request.

🡪 Class per Module

Ex: Amazon (project) 🡪 Search, Cart, Payment, …ect are Modules

SearchController

CartController

PaymentController

Q) What is **ViewResoler** in Spring MVC?

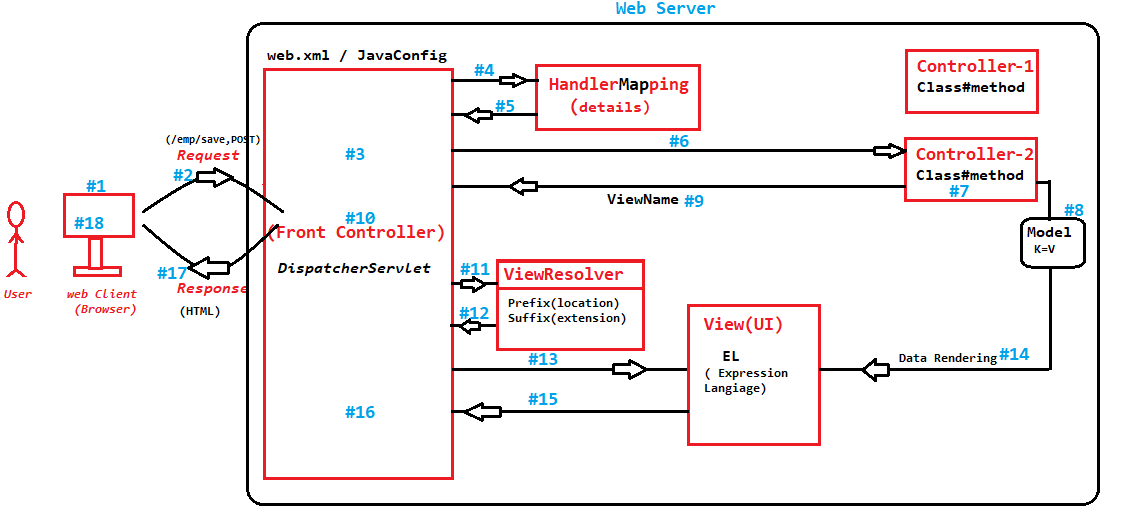
A) View Resolver will provide full UI details(prefix,suffic) to FrontController by adding prefix and suffix to viewName.

View Page = Prefix + ViewName + Suffix

=/mypages + Home + .jsp

=/mypages/Home.jsp

**Spring Web MVC**



\* FrontController need not be to configured in web.xml (it is removed in latest)

\* Need not configure HandlerMapping (you can not even configure this)

\* Need not to configure ViewResolver (it is already configured, we can change values)

#1 🡪 Open Browser/Client Application to make request

#2 🡪 Enter URL details and press Enter that makes the request to FrontController

#3 🡪 FrontController reads the request PATH, HttpMethod…

#4 🡪 Use the help of HandlerMapping with inputs (Path & HttpMethod)

#5 🡪 HandlerMapping, returns Class and method details(Controller details).

#6 🡪 FrontController, calls/executes the Controller#method

#7 🡪 Controller will process the request

#8 🡪 Controller adds data to Model memory in key=val that can shared with UI

#9 🡪 Controller returns ViewName(UI Name)

#10🡪 FrontController reads viewName

#11🡪 FrontController taks the help of ViewResolver that adds prefix(location) and suffix(extension)

#12🡪 Full view details return to FrontController

#13🡪 FrontController will call View/UI Page

#14🡪 View(UI) will read the data from Model using EL(Data rendering)

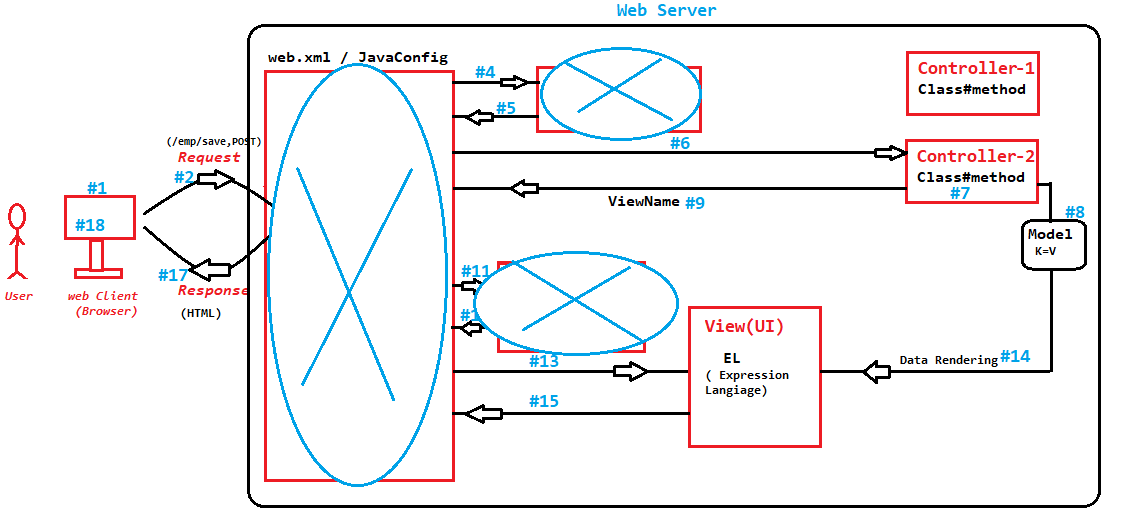
#15🡪 View(UI) page given back to to FrontController

#16🡪 View is converted to HTML format.

#17🡪 Sent back to Browser as Response

#18🡪 Display the output.

**Spring Boot MVC**



Q) What is the use of HandlerMapping?

A) It is like a Register(Map) that holds all Controller details. i.e All the request details and its Controller#method.

Q) How Spring Boot MVC is different from Spring Web MVC?

A) Both are same. In Spring Boot no need configuration for FrontController in web.xml, ViewResolver, Handler Mapping, (comes with auto-configuration in spring Boot).

Q) What should we provide with Controller method?

A) ViewName – String as ReturnType

Path – URL to be accessed by browser

HttpMethod –GET, POST…

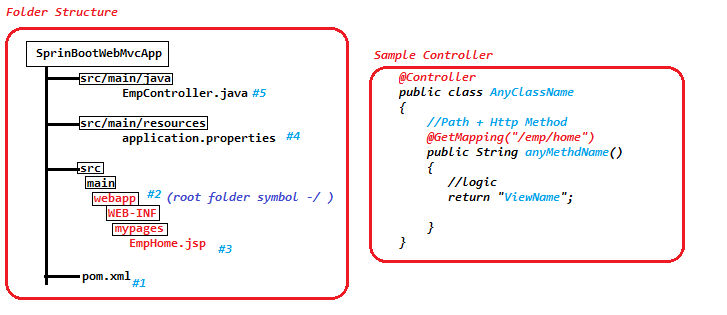
GET🡪**@GetMapping(“/path”)** or 🡪**@RequstMapping(value=”/path”, method=RequestMethod.GET)** old style

POST🡪**@PostMapping(“/path”)** or 🡪**@RequstMapping(value=”/path”, method=RequestMethod.POST)** old style

**Spring Boot Application Development**

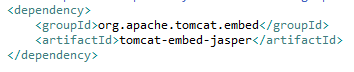
1. FC, HanderMapping, ViewResolver comes with Auto-Configuration.
2. Write the code for Controller, Model, View
3. Spring Boot : 3 Embedded Servers(Apache Tomcat, JBoss Undertow, Eclipse Jetty)
4. Properties files: port 🡪8080(default), prefix and suffix
5. Apache Tomcat(Embedded) comes with only CATALINA support. We should add JASPER(JSP Runtime) to work with JSP Pages.

* Writing Controller Code 🡪 By Programmer
* Objects Creation 🡪 By Spring Container
* Calling Controller#methods 🡪 By FrontController
* Spring Container exist in Spring Boot
* Spring Web 🡪 Tomcat(Catalina-Servlets)



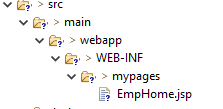
**Coding Steps:**

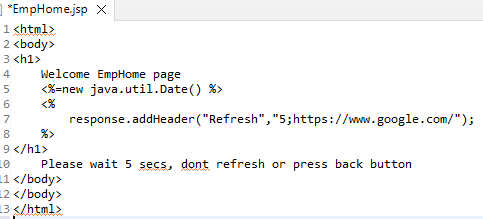
1. Create Spring Boot application: File 🡪new 🡪Spring Starter Project🡪**SbApp11a\_WebMvc🡪 java11, spring Web, Lombok, DevTool🡪next🡪Finish**
2. Update pom.xml file: Add **tomcat-embed-jasper** from web. [To work with **jsp** pages]



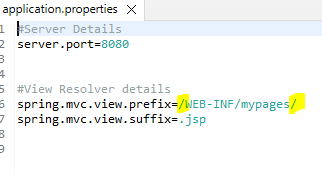
1. Create Folder structure

main🡪webapp🡪WEB-INF🡪mypages🡪EmpHome.jsp

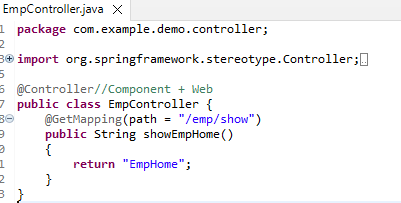




1. Update application.properties file



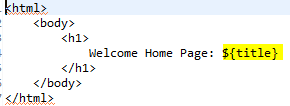
1. CreateEmpController class

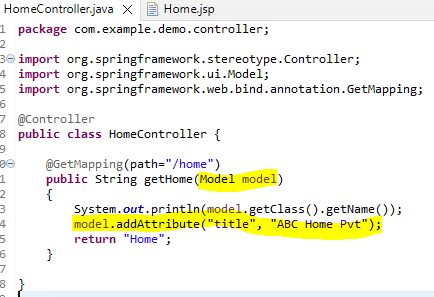


1. Run the mail class and access URL: <http://localhost:8080/emp/show>

**Model (I)**

* Used to send Data from Controller to View. **Controller 🡪View(UI)**
* Keep **“model”** as method param in Controller
* View(UI), use EL(Expression Language) to read data. **${key}**
* **Implementation class for Modle (I): org.springframework.validation.support.BindingAwareModelMap**





[**http://localhost:8080/home**](http://localhost:8080/home)

**HTML Form:**

* Used to send data from View(UI) to Controller. **UI 🡪Controller**
* Ex: Login form , Register from,…etc
* Spring Container created form data as object and stores at container only.
* Programmer, Defines a class with variable(count of variables = count of fields/inputs in from)
* Bind Form input names with variables

<input **name**=”username”…<textarea **name**=”username”…

* Form Action must match with Controller#method URL and TYPE.
* Read object from controller using Syntax

**@ModleAttribute ClassName objectName**

\* Spring Boot DevTools is used in development environment only. It avoids restarting app for every change during development. Otherwise Need to restart the application.

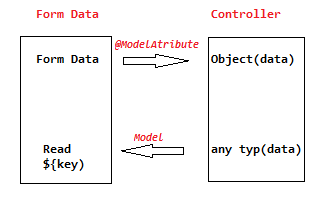
1 Form 🡪 1 Class

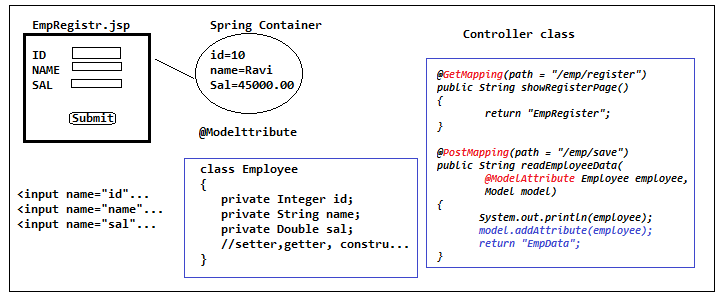
1 Input 🡪 1 variable

Input name == variable name

1 Form(filled data) submit 🡪 1 object as Request.

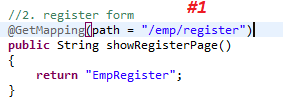
Read using **@ModelAttribute**



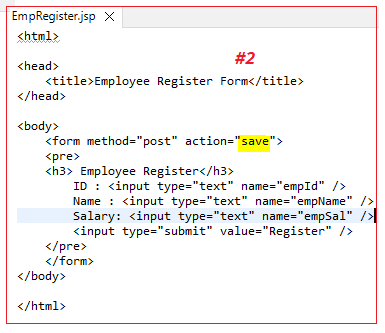


**Coding steps :**

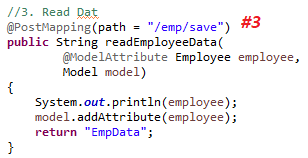
* Create a method in Controller which call EmpRegister.jsp page.



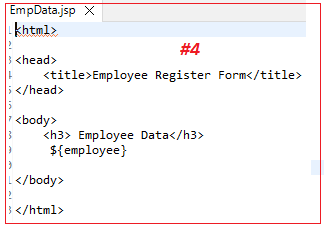
* Create **EmpRigister.jsp** page which takes form data and calls( **emp/save** URI +Post) on sumbit



* Create a method to read form data and calls EmpDat.jsp files



* Create EmpData.jsp which takes employee data from Controller.



* Test in Browser

<http://localhost:8080/home>

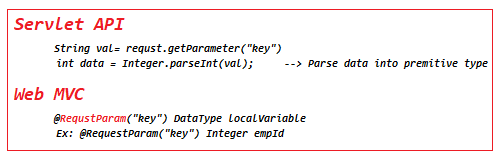
<http://localhost:8080/emp/show>

<http://localhost:8080/emp/register> --> will redirect to <http://localhost:8080/emp/save>

**Request Parameters**

**Request Parameters** (In servlets): **URL?key1=val1&key2=val2….**

Ex: [https://www.google.com/search**?q=ravikiran**&**time=5mins**](https://www.google.com/search?q=ravikiran&time=5mins)



* Key=value can be sent in any order: /emp?empId=101&empName=”Ravi” or /emp?empName=”Ravi”&empId=101
* Key=value must be present in URL. Else: WhitelabelError Page.

**DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MissingServletRequestParameterException: Required request parameter 'empName' for method parameter type String is not present]**

* We can set Request parameter as optional. Ex: **RequestParam(value=”empSal” required=false) Double eSal**

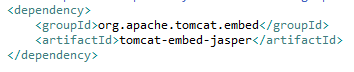
**Redirect Concept**

**Redirect:** When request comes to one resource execute and move to another resource instead of giving a direct response, 2nd resource /last resource provides response.

* Moving from one Controller method to another Controller method for same request return “**redirect**:URL”.

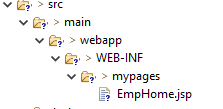
**Coding Steps:**

1. Create Spring Boot application: File 🡪new 🡪Spring Starter Project🡪**SbApp11b\_WebMvc🡪 java11, spring Web, Lombok, DevTool🡪next🡪Finish**
2. Update pom.xml file: Add **tomcat-embed-jasper** from web. To work with jsp pages(jsp🡪java🡪.class).

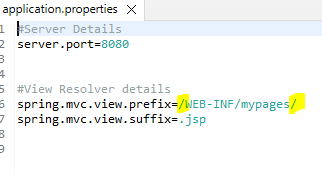


1. Create Folder structure

main🡪webapp🡪WEB-INF🡪mypages🡪EmpHome.jsp



1. Update application.properties file



1. **EmpController.java**

**package** com.example.demo.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.ui.Model;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

@Controller

**public** **class** EmpController

{

// URL: http://localhost:8080/emp?empId=101&empName="Ravi"&empSal=6000

// URL: http://localhost:8080/emp?empId=101&empName="Ravi"

@GetMapping(path = "/emp")

**public** String getEmpData(

@RequestParam("empId") Integer eId,

@RequestParam(value="empName") String eName,

@RequestParam(value="empSal" , required = **false**, defaultValue = "1000.00") Double eSal,

Model model //send data to UI

)

{

System.***out***.print("Employee details"+eId+"-"+eName+"-"+eSal);

model.addAttribute("empId", eId);

model.addAttribute("empName", eName);

model.addAttribute("empSal", eSal);

**return** "EmpData";//view Name

}

//http://localhost:8080/delete?eId=101

//will redirect to http://localhost:8080/emp?empId=101&empName=Raju&empSal=50000.0

@GetMapping(path="/delete")

**public** String deleteEmpData(

@RequestParam(value="eId") Integer eId,

Model model //send data to UI

)

{

System.***out***.println("Employee Delete called....");

String eName="Raju";

Double eSal=50000.00;

model.addAttribute("empId", eId);

model.addAttribute("empName",eName);

model.addAttribute("empSal", eSal);

**return** "redirect:emp?empId="+eId+"&empName="+eName+"&empSal="+eSal;

}

}

1. **EmpData.jsp**

<%@page isELIgnored="false"%>

<html>

<head>

<title>Employee Data</title>

</head>

<body>

<pre>

${empId} - ${empName} - ${empSal}

</pre>

</body>

</html>

1. **Run and access the below URLs.**

<http://localhost:8080/emp?empId=105&empName=Raju&empSal=50000.0>

<http://localhost:8080/emp?empId=105&empName=Raju>

<http://localhost:8080/delete?eId=110>

**Note:**

* Controller class must be in base or sub package.
* Do not give spaces in properties after values.
* Must start prefix with / and must end prefix with /.
* Folder system must be case-sensitive.
* Controller#method return type and page name must match(case-sensitive).
* Check did you add tomcat-embedd-jasper manually.
* If you do code modifications then stop app and restart it.
* Controller class must have @Controller annotation and method must have GetMapping or PostMapping…

**SB Web MVC with ThymeLeaf**

**JSP Benifits:**

* JSP UI [Java Server Pages]
* Compared to servlet easy to write and handle errors.
* Dynamic UI can be designed.
* JSTL Tags and Custom Tags are supported.

**JSP Limitation:**

* **JSP** = Static Tags(HTML) + Dynamic Tags(JSTL/Java)🡪 all converted to Servlet(Java code) internally.
* **JSP** 🡪Servlet (.java) 🡪 Compiled(.class) 🡪Executed. ( By **JASPER**)
* **Even Static Tags are converted into Java format. (Performance is Slow)**

**ThymeLeaf UI: Used to create dynamic web pages using Java programming.**

* It is light weight (uses less memory) and faster compare to JSP. Here, static Tags are not converted , Only Dynamic tags/java code is converted and embedded back to static content.
* This is converted to HTML not Java format.

**ThymeLeaf Symbols:**

**@{…} 🡪** Location /Path

HyperLink, Form Action, Script(JS) and CSS links

Ex:

*HTML 🡪 <a href=”/employee/show”> View </a>*

*🡪 <form action=”/student/save” >…..</form>*

*ThymeLeaf 🡪 <a th:href=”@{/employee/show}”> View </href>*

*🡪 <form th:action=”@{/student/save}”> …..</form>*

**${…} 🡪** Reading data from Model(I) (Given by controller)

Ex: Controller 🡪 Model(I) sid=10 🡪Read at UI : ${sid}

**\*{…} 🡪** Form Inputs for Bi-Directional (for Edit/Update)

Ex:

*HTML 🡪 <input type=”text” name=”sname” id=”sname”/>*

*ThymeLeaf🡪<input type=”text”* ***th:field=”\*{sname}”*** */>*

**[[….]] 🡪**Execute some expression and print

Ex:

*[[ ‘Name is ‘ +* ***${sname}*** *]] 🡺 Out put : Name is Ravi*

* *Spring Boot support ThymeLeaf by Default. No need to add dependency*
* *No need to provide view resolver.* ***prefix=templates*** *folder and* ***suffix= .html***
* *Need not to provide key=values in properties file.*

**Coding Steps:**

1. Create Spring Boot application: File 🡪new 🡪Spring Starter Project🡪**SbApp11c\_WebMvcThymeLeaf🡪 java11, spring Web,ThymeLeaf , DevTool🡪next🡪Finish**
2. Create Controller class EmpController which is same for any UI (JSP or ThymeLeaf…etc).