31/10/2019

1.Dependency Injection : Container dynamically Assigns dependent values to resource (class/object) is called Dependency Injection

1.Setter Injection

2.Constructor Injection

3.InterFace Injection

4.Lookup Method Injection

-------------------------------------------------------------------------------------------------------------

2.Container Operaions :

1. Creating Object To Spring Beans

2. providing data to variables

3. link with other Objects

4. finally , destroying the Object after application Executed.

--------------------------------------------------------------------------------------------------------------

3. we need 3 files for spring development

1.bean /pojo class

2. configuration file

3. test class

=================================================================

4.Dependency = variable(3 types)

1.primitive dependecy(8+1)

2.Collection Type(List,Set ,Map ,properties)

3.Reference Type Dependency(No count)

**5.Has-A Relation(Association)**

Using Child Class /Interface as a data type and creating variable inside parent class is Called Association Mapping.

6.we configure in 3 ways(Configure file)

1.xml Configuration

2.java Configuration

3.Annotations configuration

Xml configuration

<bean class="cls name" name="obj name">

<property name=" var name>

<value> data</value>

<property>

</bean>

bean class=*"com.app.bean.Student"* name=*"sobj"*>

<property name=*"id"*>

<value>33</value></property>

<property name=*"sname"*><value>narendra</value></property>

</bean>

7.Spring Container Provided Two Interfaces

1.Bean Factory(old)= supports xml only-implements Only one-XmlBeanFactoty class

2.Application Context= supports both xml and annotations

ApplicationContext b=new ClassPathXmlApplicationContext("config.xml");

BeanFactory b=**new** ~~XmlBeanFactory~~(**new** FileSystemResource("src/config.xml"));

**1/11/2019**

**Setter Injection :** Container calls setxxx()/setter methods to inject dependent values.

-- providing data using set mettod is called setter injection

**package** com.nare.bean;

**import** java.util.Date;

**public** **class** Student {

**private** **int** id;

**private** String name;

**private** Date d;

**public** Student() {

**super**();

System.***out***.println(" constructor");

}

@Override

**public** String toString() {

**return** "Student [id=" + id + ", name=" + name + ", d=" + d + "]";

}

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Date getD() {

**return** d;

}

**public** **void** setD(Date d) {

**this**.d = d;

}

}

**Config.xml**

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean class=*"java.util.Date"* name=*"dt"*></bean>

<bean class=*"com.nare.bean.Student"* name=*"sobj"*>

<property name=*"id"*>

<value>8</value>

</property>

<property name=*"name"*>

<value>narendra</value>

</property>

<property name=*"d"*>

<ref bean=*"dt"*/>

</property>

</bean>

</beans>

**Test.java**

**package** com.nare.test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** com.nare.bean.Student;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ApplicationContext ac=**new** ClassPathXmlApplicationContext("config.xml");

Student s=(Student) ac.getBean("sobj");

System.***out***.println(s);

}

}

**Value Tag**

**V**alue tag define in 3 ways

1. value as a tag

**<property name=*"id"*>**

**<value>8</value>**

**</property>**

**2. Value as Attribute**

<property name=*"id"* value=*"8"*>

3.P-Schema or name space

<bean class=*"java.util.Date"* name=*"dt" p:id=”8”*></bean>

Concept tag

Object creation <bean>

Primitive type <value>

Collection type <list>,<set>,<map>,<props>

Setter Injection <property>

Constructor Injection <constructor-arg>

Look up method <look-method>

WORKING WITH COLLECTION TYPES

Working with List

-Duplicate Values are Allowed

-Internally uses Array List

-no property tag injects null value

<list>

<value>naren</value>

<value>balu</value>

<value>saida</value>

<value>prince</value>

</list>

Working with Map

--Map is set of entries

--Spring Container Internally uses HashMap

<property name=*"addr"*>

<map>

<!-- key and value as tag -->

<entry>

<key><value>1</value></key>

<value>HYD</value>

</entry>

<!-- key and value as attribute -->

<entry key=*"2"* value=*"Vja"*></entry>

<!-- key attribte and value as tag -->

<entry key=*"3"*><value>chennai</value> </entry>

<!-- key tag and value as attribute -->

<entry value=*"Mumbai"*><key><value>4</value> </key> </entry>

</map>

</property>

Working with Properties

--Properties also key value format

<property name=*"fees"*>

<props>

<prop key=*"java"*>1000</prop>

<prop key=*"spring "*>2000</prop>

<prop key=*"boot"*>3000</prop>

</props>

</property>

EX:

Bean:

**public** **class** Course {

**private** **int** id;

**private** List name;

**private** Set course;

**private** Map addr;

**private** Properties fees;

//setter and getter methods

}

**Config.xml**

<bean class=*"com.nare.bean.Course"* name=*"cobj"* p:id=*"8"* >

<property name=*"name"*>

<list>

<value>naren</value>

<value>balu</value>

<value>saida</value>

<value>prince</value>

</list>

</property>

<property name=*"course"*>

<set>

<value>java</value>

<value>spring</value>

<value>boot</value>

</set>

</property>

<property name=*"addr"*>

<map>

<!-- key and value as tag -->

<entry>

<key><value>1</value></key>

<value>HYD</value>

</entry>

<!-- key and value as attribute -->

<entry key=*"2"* value=*"Vja"*></entry>

<!-- key attribte and value as tag -->

<entry key=*"3"*><value>chennai</value> </entry>

<!-- key tag and value as attribute -->

<entry value=*"Mumbai"*><key><value>4</value> </key> </entry>

</map>

</property>

<property name=*"fees"*>

<props>

<prop key=*"java"*>1000</prop>

<prop key=*"spring "*>2000</prop>

<prop key=*"boot"*>3000</prop>

</props>

</property>

</bean>

.

**5/11/2019**

**Working with reference type in Spring**

Has-A = reference type

-Using Child class as a data type and creating variable inside parent class is known as Association mapping

-Using Existed members in another class

-Ref tag we can use in 3 ways:

1. ref as a tag

2. ref as a attribute

3. ref as a p schema

4. Inner Bean

Ex:

**package** com.app.bean;

**public** **class** Invoice {

**private** **int** invId;

}

**package** com.app.bean;

**public** **class** Customer {

**private** Invoice inv;

}

Cfg.xml

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

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean class=*"com.app.bean.Invoice"* name=*"iob"*>

<property name=*"invId"*>

<value>555</value>

</property>

</bean>

<bean class=*"com.app.bean.Customer"* name=*"cob"*>

<!--1 ref as a tag -->

<property name=*"inv"*>

<ref bean=*"iob"*/> </property>

</bean>

<!--2 ref as a attribute -->

<bean class=*"com.app.bean.Customer"* name=*"cob1"*>

<property name=*"inv"* ref=*"iob"*> </property>

</bean>

<!--3 ref as p schema -->

<bean class=*"com.app.bean.Customer"* name=*"cob2"* p:inv-ref=*"iob"*>

</bean>

<!—4.InnerBean -->

<bean class=*"com.app.bean.Customer"* name=*"cob3"*>

<property name=*"inv"*>

<bean class=*"com.app.bean.Invoice"* name=*"iobinner"*>

<property name=*"invId"* value=*"454"*></property>

</bean>

</beans>

6/11/2019

Circular Dependency In Spring:

If two beans are dependent to each other then we say they in cyclic Dependency

-spring container can not perform cyclic dependency through constructor Injection but can perform using setter Injection.

**package** com.nare.bean;

**public** **class** A {

**private** B b;

**public** A() {

**super**();

}

@Override

**public** String toString() {

**return** "A [b=" + b + "]";

}

**public** B getB() {

**return** b;

}

**public** **void** setB(B b) {

**this**.b = b;

}

}

**package** com.nare.bean;

**public** **class** B {

**private** A a;

**public** B() {

**super**();

}

**public** A getA() {

**return** a;

}

**public** **void** setA(A a) {

**this**.a = a;

}

}

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

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

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

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean class=*"com.nare.bean.A"* name=*"aob"*>

<property name=*"b"* ref=*"bob"*></property>

</bean>

<bean class=*"com.nare.bean.B"* name=*"bob"*>

<property name=*"a"* ref=*"aob"*></property>

</bean>

</beans>

**Spring Bean Life Cycle**

Spring IOC container manages Spring bean life cycle

init():calls after creating object to the bean.

-By using init method we can check all the dependent values are injected or not.

destroy():calls before destroying the object

-By using destroy method we can Close all the resources

We can call explicitly Spring life cycle methods in three ways

1) Declarative approach

2) Programmatic approach

3) Annotation approach

***1) Declarative approach***

In this we take method with no param and configure in xml file like

**init-method=*"setinit"***

destroy-method=*"setdestroy*

**package** com.app.bean;

**public** **class** Ex {

**private** **int** i;

**public** Ex() {

**super**();

System.***out***.println("object creation");

}

**public** **void** setinit() {

System.***out***.println("init method");

}

**public** **void** setdestroy() {

System.***out***.println("setting values");

System.***out***.println("init method");

}

}

<bean class=*"com.app.bean.Ex"* name=*"eob"* init-method=*"setinit"* destroy-method=*"setdestroy"*>

<property name=*"i"* value=*"55"*></property>

</bean>

Op:

object creation

setting values

init method

Ex [i=55]

***2) Programetic Approach***

This Approach is used to implement two interfaces InitialisingBean and DisposableBean

IntialisingBean interface gives a abstract method –aferPropertiesSet() like init method

DisposableBean interface gives a abstract method –destroy() like Destroy method

- we need to implement these two Interfaces for custom init and custom destroy.

- By using init method we can check all the dependent values are injected or not.

- By using destroy method we can Close all the resources

**public** **class** Ex **implements** InitializingBean,DisposableBean {

@Override

**public** **void** destroy() **throws** Exception {

System.***out***.println("destroy");

}

**public** **void** afterPropertiesSet() **throws** Exception {

System.***out***.println("after properties set");

}

}

***3) Annotations Approach***

We can use two annoatations to implement custom init and destroy

1) @PostConstruct- init method

2) @PreDestroy-destroy method

@PostConstruct

**public** **void** setinit() {

System.***out***.println("init method");

}

@PreDestroy

**public** **void** setdestroy() {

System.***out***.println("init method");

}

We need to add context schema in configuration xml file

<context:annotation-config/>

<bean class=*"com.app.bean.Ex"* name=*"eob"*></bean>

**Bean Externalization( working with properties file in xml configuration )**

We can get data from properties file is called Bean Externalization

In real scenario we use jdbc, orm, security we can use several classes we need not to mention every where so we can create properties file access in anywhere

-in xml we access by using like this ${drivername}

Ex:

Application.properties

#dummy values for database

dc=oracle

url=DbUrl

un=father

pwd=amma

**public** **class** Ex {

**private** String driver;

**private** String url;

**private** String un;

**private** String pwd;

}

1) configure properties file by using class

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" name="pob">

<property name="location" value="application.properties"></property>

</bean>

2) configure properties file by using context schema

<context:property-placeholder location=*"application.properties"*/>

<bean class=*"com.app.bean.Ex"* name=*"eob"*>

<property name=*"driver"* value=*"${dc}"*></property>

<property name=*"url"*>

<value>${url}</value>

</property>

<property name=*"un"* value=*"${un}"*></property>

<property name=*"pwd"* value=*"${pwd}"*></property>

</bean>

</beans>

**Stand Alone collection(SAC):**

A collection is created as object in spring container , out side <bean > Object it behave as a independent stand alone collection

SAC are reusable objects .

To define SACs are in util- schema

Ex:

<util:list list-class=*"java.util.ArrayList"* id=*"lob"*>

<value>8</value>

<value>876</value>

<value>56</value>

</util:list>

----------------------------------------------------------------

<util:map map-class=*"java.util.HashMap"* id=*"mob"* >

<entry>

<key><value>1</value></key>

<value>HYD</value>

</entry>

<entry>

<key><value>5</value></key>

<value>fhfh</value>

</entry>

</util:map>

----------------------------------------------------------------

<util:properties id=*"pob"*>

<prop key=*"kfkf"*>hdhd</prop>

<prop key=*"s"*>hdhd</prop>

</util:properties>

----------------------------------------------------------------

**Annotations Configuration in Spring:**

It has mainly two types of annotations...

**1)Stereo type annotations:**

[1.@Component](mailto:1.@Component)

2.@Repository

[3. @Service](mailto:3.@Service)

4.@ Controller

5.@ RestController

**2)Basic/Data Annotations:**

[1.@Value](mailto:1.@Value)

[2.@Autowired](mailto:2.@Autowired)

[3.@Qualifier](mailto:3.@Qualifier)

**Annotation | Meaning                                             |**

+------------+-----------------------------------------------------+

| [**@Component**](https://www.dineshonjava.com/spring-component-annotation/) | **generic stereotype for any Spring-managed component** |

| **@Repository**| **stereotype for persistence layer**                    |

| **@Service** | **stereotype for service layer**                        |

| **@Controller|** **stereotype for presentation layer (spring-mvc)**

**@Controller** annotation is for a class as a Spring Web MVC controller. It is a meta annotation of @Component, so beans annotated with it are automatically imported into the Spring container. If you add the @Controller annotation to a class then you can use handler mappling annotation i.e. @RequestMapping; to map URLs to instance methods of a class.

**@Service** annotation is for a class as a Service of application.

**@Repository** annotation is more suitable annotation that provides additional benefits specifically for DAOs. The @Repository annotation is a meta annotation of the @Component annotation with similar use and functionality. In addition to importing the DAOs into the DI container, it also makes the unchecked exceptions eligible for translation into Spring DataAccessException.

**@Component** should be used when your class does not fall into either of three categories i.e. ***Controllers***, ***Services*** and ***DAOs***.

Those all are detect the class and creating the a object.

@Component("eob")

**public** **class** Employee {

@Value("10")

**private** **int** id;

}

@Configuration

@ComponentScan("com.app")

**public** **class** AppConfig {

}

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

//System.out.println(ac.getBean("employee")); if ur not giving componenet sacn classname stating letter with create object\

System.***out***.println(ac.getBean("eob"))

**Bean Externalisation with Annotations:**

Providing direct value to the variable is called Hard Coding

To Remove Hard Coding we can give data in runtime in different sources

1.command line arguments

2.scanner/ BufferReader

3.properties File

4.html forms

Properties file is created values with key value pairs

-both key and value ar strings

-Environment Object is used to call properties file values in to our class

-To identify the properties file in our class we have a Annotation @PropertySource(“app.properties)

Spring Bean Scope: In general Scope means Life Time of in memory

1)SingleTon scope:

Makes creating Single object to every Request. This is **default scope** given for every bean by spring container.

2) prototype:

Makes creating new object to every Request. On every access it creates new Object with data by container

3) request:

Bean class Object is to every request

4)session scope

Bean class Object is visible through out of session

5)Global Session scope(removed from 3.X)

In case of spring xml configuration

<bean class=*"com.nare.Employee"* id=*"emp"* scope=*"singleton"*></bean>

<bean class=*"com.nare.Employee"* id=*"emp1"* scope=*"prototype"*></bean>

Test.java

//singleton scope

Employee e=(Employee) ac.getBean("emp");

Employee e1=(Employee) ac.getBean("emp");

System.***out***.println(e==e1);

System.***out***.println(e.hashCode()+" "+e1.hashCode());

//prototype

Employee e2=(Employee) ac.getBean("emp1");

Employee e3=(Employee) ac.getBean("emp1");

System.***out***.println(e2==e3);

System.*out*.println(e2.hashCode()+" "+e3.hashCode());

Op:

true

346861221 346861221

false

1188392295 226710952

1)target class singleton and dependent class Singleton

**public** **class** A {

}

**public** **class** B {

**private** A a;

}

c.xml

<!--target class -->

<bean class=*"com.app.bean.B"* name=*"bob"* scope=*"singleton"*

p:a-ref=*"aob"*/>

<!-- dependent class -->

<bean class=*"com.app.bean.A"* name=*"aob"* scope=*"singleton"*>

test.java

B b=(B) ac.getBean("bob");

B b1=(B) ac.getBean("bob");

B b2=(B) ac.getBean("bob");

System.***out***.println(" dependent hash codes");

System.***out***.println(b.getA().hashCode());

System.***out***.println(b1.getA().hashCode());

System.***out***.println(b2.getA().hashCode());

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

System.***out***.println(" target hash codes");

System.***out***.println(b.hashCode());

System.***out***.println(b1.hashCode());

System.***out***.println(b2.hashCode());

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

We expected values dependent class hash codes same and target class hash codes same

Op: we got expected results

dependent hash codes

1121647253

1121647253

1121647253

-------------

target hash codes

1694556038

1694556038

1694556038

-------------

2)target class prototype and dependent class prototype

<!--target class -->

<bean class=*"com.app.bean.B"* name=*"bob"* scope=*"prototype"* p:a-ref=*"aob"*/>

<!-- dependent class -->

<bean class=*"com.app.bean.A"* name=*"aob"* scope=*"prototype"*> </bean>

We expected output values target class hash codes different and dependent class hash codes different

Op: we got expected results

dependent hash codes

1476394199

837764579

1501587365

-------------

target hash codes

1007603019

348100441

1597249648

-------------

3)target class prototype and dependent class singleton

<!--target class -->

<bean class=*"com.app.bean.B"* name=*"bob"* scope=*"prototype"* p:a-ref=*"aob"*/>

<!-- dependent class -->

<bean class=*"com.app.bean.A"* name=*"aob"* scope=*"singleton"*> </bean>

We expected output values target class hash codes different and dependent class hash codes same

Op: we got expected results

dependent hash codes

1121647253

1121647253

1121647253

-------------

target hash codes

1694556038

1076496284

1508646930

-------------

4)target class singleton and dependent class prototype

<!--target class -->

<bean class=*"com.app.bean.B"* name=*"bob"* scope=*"singleton"* p:a-ref=*"aob"*/>

<!-- dependent class -->

<bean class=*"com.app.bean.A"* name=*"aob"* scope=*"prototype"*> </bean>

We expected output values target class hash codes same and dependent class hash codes different

Op: we got unexpected results prototype also comes same hashcodes

dependent hash codes

1121647253

1121647253

1121647253

-------------

target hash codes

1694556038

1694556038

1694556038

-------------

But we got Unexpected Results here we Use Look-Up-Method Injection

**Look-up-method Injection :**

Target class singleton and dependent class prototype

We expected output values target class hash codes same and dependent class hash codes different

But we get unexpected results shown above that’s why spring container provide look-up- method injection

Solution:1

1) Make target class as a abstract .

2) Delete setter method of dependent in target class( actually we are injecting values by setter injection here we change to lookup method)

3)write a abstract method with dependent (child class ) return type and any method name and call this method in getter dependent getter method and return this value.

**public** **abstract** **class** B {

**private** A a;

**public** A getA() {

a=setLookUpmethod();

**return** a;

}

// delete setter method

**public** **abstract** A setLookUpmethod();

}

<bean class=*"com.app.bean.B"* name=*"bob"* scope=*"singleton"*>

<lookup-method name=*"setLookUpmethod"* bean=*"aob"*/>

</bean>

Then we get Write values

dependent hash codes

1150538133

662822946

92150540

-------------

target hash codes

1110623531

1110623531

1110623531

-------------

Solution:2

@lookUp annotation

**Factory method:** the method that’s capable creating either same class or different class is called factory method

1)Instance Facory Method

Ex: String s=new String(“hello”);

String s2=s.cancat(“ how are you ?”);// returns same class object

StringBuffer s=new StringBuffer(“hello”);

String s2=s.subString(0,4);// returns different class object class object

2)static Facory Method

Class .forName(“java.util.Date”); // returns same class object

Console c= System.console();//returns different class object class object

WE Can make IOC container to create Spring bean class in four ways

1) Using Zero param constructor

2) Using Parameterized constructor

3)Using Instance Factory method

4)Using static factory method

<!-- USing static factory methods -->

<bean class=*"java.lang.Class"* factory-method=*"forName"* id =*"c1"*>

<constructor-arg value=*"java.util.Date"*/>

</bean>

<bean id=*"c2"* class=*"java.util.Calendar"* factory-method=*"getInstance"*> </bean>

<!-- using instance factory methods -->

<bean class=*"java.lang.String"* id=*"s1"*>

<constructor-arg value=*"hello"*></constructor-arg>

</bean>

<bean id=*"s2"* factory-bean=*"s1"* factory-method=*"concat"*>

<constructor-arg>

<value>how are you</value>

</constructor-arg>

</bean>

</beans>

**Construtor Injection**

Providing data to variable by using constructor is called Constructor injection

We can configure constructor injection by using Xml

Connot possible circular dependency injection with constructor

At the bean creation it will load bit fast then setter Injection

<bean class=*"com.nare.bean.Employee"* id=*"eob"*>

<constructor-arg><value>26</value></constructor-arg>

<constructor-arg><value>narendra</value></constructor-arg>

</bean>

Dependency Check

Whether we perform in the setter injection we cannot give values in real may or may not .

If we perform those values are mandatory then we check with @Required

@required is deprecated

**\*\*\*Auto wiring**

***Explicit Wiring Vs Auto wiring :***

Programmer made has a relation by using ref tag is Bean wiring whereas IOC makes relation between parent and child is called Auto wiring

-Here IOC automatically detect the values to the dependents and inject to the target objects

Auto wiring can perform in four ways:

1). byname

2).byType

3).constructor

4).autodetect

Autowiring using xml

*1)None* :

Which is no wiring.programmer has to do wirig by using ref tag.

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address a;

}

**public** **class** Address {

**private** **int** ad;

}

<bean class=*"com.nare.Address"* id=*"aob"* p:ad=*"7"*></bean>

<bean class=*"com.nare.Employee"* id=*"eob"* p:id=*"8"* p:name=*"narendra"* p:a-ref=*"aob"* autowire=*"no"*> </bean>

</beans>

*2)bytype:*  
spring container will compare has relation class name (datatype ) with <bean class=” ”> in xml congfiguration

No of matchings Result

=0 null

=1 inject

>1 NoUniqueBeanDefException

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address a;

}

**public** **class** Address {

**private** **int** ad;

}

<bean class=*"com.nare.Address"* id=*"aob"* p:ad=*"7"*></bean>

<bean class=*"com.nare.Employee"* id=*"eob"* p:id=*"8"* p:name=*"narendra"* autowire=*"byType"* > </bean>

</beans>

3)byname:

Spring container will compare has relation class variable with <bean name=” ”> in xml congfiguration

No of matchings Result

=0 null

=1 inject

>1 NoUniqueBeanDefException

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address a;

}

**public** **class** Address {

**private** **int** ad;

}

<bean class=*"com.nare.Address"* id=*"a"* p:ad=*"7"*></bean>

<bean class=*"com.nare.Employee"* id=*"eob"* p:id=*"8"* autowire=*"byName"*> </bean>

</beans>

4)constructor Autowiring

Spring container will search parameterized constructor with child param

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address a;

**public** Employee(Address a) {

**super**();

**this**.a = a;

}

}

**public** **class** Address {

**private** **int** ad;

}

<bean class=*"com.nare.Address"* id=*"a"* p:a=*"55"*></bean>

<bean class=*"com.nare.Employee"* id=*"eob"* p:id=*"8"* p:name=*"narendra"* autowire=*"constructor"*> </bean>

</beans>

5).autodetect: it is removed from 3.x

Auto wiring Using Annotations:

@Autowired is used to link objects

@Autowired internally follows @Required is true

No of matchings Result

=0 -NoSuchBeanDefException

@Autowired 🡪required is true

-Null

@Autowired 🡪required is false

=1 inject(Bytype)

>1 a) Compare variable name and bean name

Matching injects else NoUniqueBeanDefException

b)use @qualifier if matching found inject

else NoSuchBeanDefException

If we perporm annotation by using xml configuration we can configure

<context:annotation-config/>

1) one matching found(bytype)

**public** **class** Address {

**private** String a;

}

**public** **class** Employee {

**private** **int** id;

**private** String name;

**private** Address a;

}

<bean>

context:annotation-config/>

<bean class=*"com.app.bean.Address"* name=*"aob"*>

<property name=*"a"* value=*"vja"*></property>

</bean>

<bean class=*"com.app.bean.Employee"* name=*"eob"*>

<property name=*"id"* value=*"4"*></property>

</bean>

2) zero matching found(NoSuchBeanDefinitionException)

Then we need to @Autowired(required=**false**)

**public** **class** Employee {

**private** **int** id;

**private** String empname;

@Autowired(required=**false**)

**private** Address a;

}

<bean>

<context:annotation-config/>

<bean class=*"com.app.bean.Employee"* name=*"eob"*>

<property name=*"id"* value=*"4"*></property>

<property name=*"empname"* value=*"naren"*></property>

</bean>

3) more than one matching found(NoUniqueBeanDefinitionException)

**public** **class** Employee {

**private** **int** id;

**private** String empname;

@Autowired

**private** Address a;

}

<context:annotation-config/>

<bean class=*"com.app.bean.Address"* name=*"aob"*></bean>

<bean class=*"com.app.bean.Address"* name=*"aoob"*></bean>

<bean class=*"com.app.bean.Employee"* name=*"eob"*>

If we give same name of dependent variable bean name it injects by bynametype like below

<bean class=*"com.app.bean.Address"* name=*"a"*></bean>

<bean class=*"com.app.bean.Address"* name=*"aoob"*></bean>

<bean class=*"com.app.bean.Employee"* name=*"eob"*>

4) more than one matching found use Qualifier of given name like below

**public** **class** Employee {

**private** **int** id;

**private** String empname;

@Autowired

@Qualifier("aob")

**private** Address a;

}

<bean class=*"com.app.bean.Address"* name=*"aob"*></bean>

<bean class=*"com.app.bean.Address"* name=*"aoob"*></bean>

This annotation we can use at below 3 levels

1) field

@Autowired

**private** Address a;

2) setter method

@Autowired

**public** **void** setA(Address a) {

**this**.a = a;

}

3) constructor (if we have only one parameterized constructor with beans as parameters @Autowired is optional)

/\* @Autowired\*/ it is optional

**public** Employee(Address a) {

**super**();

**this**.a = a;

}

Auto wiring Using Annotations Configuration:

By using configuration

-Container find to find child object and inject into parent object

@Component

**public** **class** Employee {

@Value("8")

**private** **int** id;

@Value("naren")

**private** String empname;

@Autowired

**private** Address a;}

@Component

**public** **class** Address {

@Value("vja")

**private** String a;

}

@Configuration

@ComponentScan("com")

**public** **class** AppConfig {

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

System.***out***.println(ac.getBean("employee"));

}

}

If depedent is interface then container search child obj

@Component

**public** **class** Employee {

@Value("8")

**private** **int** id;

@Value("naren")

**private** String empname;

@Autowired

**private** Address a;

}

@Component

**public** **interface** Address {

}

@Component

@Primary

**public** **class** Address1 **implements** Address{

@Value("hyd")

**private** String a;

}

@Component

**public** **class** Address2 **implements** Address {

@Value("vja")

**private** String a;

}

@Configuration

@ComponentScan("com")

**public** **class** AppConfig {

}

**public** **class** Test {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

System.***out***.println(ac.getBean("employee"));

}

}

Op: Employee [id=8, empname=naren, a=Address1 [a=hyd]]

If two depedent child class a`vailable then we get NoUniqueBeanDefinitionException

Solution we need to give @Primary or @Qualifier

@Qualifier is used in the target with proper name like

@Component

**public** **class** Employee {

@Value("8")

**private** **int** id;

@Value("naren")

**private** String empname;

@Autowired

@Qualifier("address1")

**private** Address a;

}

@Primary is used in the child like

@Component

@Primary

**public** **class** Address1 **implements** Address{

@Value("hyd")

**private** String a;

}

If we both give both @Qualifier and @Primary then container will take higher priority to @Qualifier

SpEL: (Spring expression language)

It is combination of variables , methods , collections, operations and special symbols

Syntax: @value(“#{....}”)

private data type varname;

Ex:

@Value("#{'dhh'.length()}")

**private** **int** a;

**Spring JDBC:**

*While jdbc we need do:*

Loading Driver

Creating Connection

Create statement

Create SQL query

Providing data sql query

Execute sql query

Print result

Close connection

*In Sprig jdbc we no need to write all those*

Just configuring JdbcTemplate and

Create query

Prepare data to query

Print result

-So avoid boiler plate code by using spring jdbc

-JdbcTemplate has –a relation with DriverManagerDataSource.

- by using JdbcTemplate we can perform jdbc operation

- In DriverManagerDataSource we can set the

* Driver class
* url
* user name
* password

Var-args :

Var-args concept is introduced in 1.5

* in array we can not change the size
* in var args concept to we can send no of values.
* It is also internally array without worring size.

**package** com;

**public** **class** VarargsConcept {

**static** **void** m( **int**... a){

**for**(**int** i=0;i<a.length;i++) {

System.***out***.println(a[i]);

}

}

**public** **static** **void** main(String[] args) {

**int**[] a= **new** **int**[] {4,5,4,5,47,7};

VarargsConcept.*m*(a);

*m*(4,5,4,5,47,7,4);

}

}

*Note :*

1) one var-arg argument is allowed in the method else ce

2) var-arg argument must be defined in last . it can’t be first or middle

**static** **void** m( **int** j, **int**... a){// allowed

**for**(**int** i=0;i<a.length;i++) {

System.***out***.println(a[i]);

}

}

In java 8 introduced static and default methods in interface.

-Actually static methods executed from object reference variable

-if we have one static method we better to choose interface in place of class so memory can saved and performance will be increased.

**public** **interface** Concept {

**static** **void** m() {

System.***out***.println("interface static method");

}

**public** **static** **void** main(String[] args) {

Concept.*m*();

}

}

* Default methods in interface
* Is also called instance methods.
* This is used to write to avoid writing abstract class for common logic for all methods.
* Where as default methods can be overridden but abstract must be overridden.
* **package** com;
* **public** **interface** VarargsConcept {
* **default** **void** m() {
* System.***out***.println("interface defaultmethod");
* }
* }
* **package** com;
* **public** **class** Ex **implements** VarargsConcept{
* Ex name = **new** Ex();
* **public** **static** **void** main(String[] args) {
* VarargsConcept name = **new** Ex();
* name.m();
* }
* }

update()

* update method is used insert,update,delete the data .
* Non select queries we can perform by using update().
* It is in JdbcTemplate
* It returns how many rows will be effected

**int** update(String sql, @Nullable Object... args) **throws** DataAccessException

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.2.1.RELEASE</version>

</dependency>

package com.app.config;

import javax.sql.DataSource;

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import org.springframework.context.annotation.PropertySource;

import org.springframework.core.env.Environment;

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration

@PropertySource("app.properties")

@ComponentScan("com")

public class AppConfig {

@Autowired

private Environment e;

@Bean

public DriverManagerDataSource datasourc() {

DriverManagerDataSource d=new DriverManagerDataSource();

d.setDriverClassName(e.getProperty("dc"));

d.setUrl(e.getProperty("url"));

d.setUsername(e.getProperty("un"));

d.setPassword(e.getProperty("pwd"));

return d; }

@Bean

public JdbcTemplate jd() {

JdbcTemplate j=new JdbcTemplate();

j.setDataSource(datasourc());

return j;

}

}

**package** com.app.test;

**import** org.springframework.context.ApplicationContext;

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

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** com.app.config.AppConfig;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

JdbcTemplate t=(JdbcTemplate) ac.getBean("jd");

String sql="insert into emptab values(?,?,?)";

**int** i=t.update(sql,18,"aa",6);

System.***out***.println(i);

}

}

Lamda Expression:

* In java 8 it is introduced
* It is short term of writing implementation class and returning object of class.
* It is only possible with only functional interface
* Functional interface have only one non static method
* Saves lot of code
* This is implementation logic + object
* If only one parameter () symbol is optional
* Only one statement { } is optional.
* Syntax
* Interfacename s=(param)🡪 {body}
* VarargsConcept v=()->{
* System.***out***.println("djdjdj");
* };
* **package** com;
* @FunctionalInterface
* **public** **interface** VarargsConcept {
* **public** **void** m();
* }

Select operations using JdbcTemplate

-Spring provides special methods

queryForObject() : it is used to fetch one row in to one objects returns class type

**public** <T> T queryForObject(String sql,

RowMapper<T> rowMapper) **throws** DataAccessException

query: it is used to fetch multiple row in to list of objects returns list of class type

**public** <T> List<T> query(String sql,

RowMapper<T> rowMapper) **throws** DataAccessException

RowMapper : Rowmapper is functional interface. it will convert one row to one object.

***queryForObject():***

**package** com.app.test;

**import** java.sql.ResultSet;

**import** org.springframework.context.ApplicationContext;

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

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.jdbc.core.RowMapper;

**import** com.app.bean.Product;

**import** com.app.config.AppConfig;

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

JdbcTemplate t=(JdbcTemplate) ac.getBean("jd");

String sql1="select \* from prodtab where pid=?";

RowMapper<Product> rm=(ResultSet rs, **int** rowNum)->{

**return** **new** Product(

rs.getInt("pid"),

rs.getString("pcode"),

rs.getString("pcost")

);

};

Product p=t.queryForObject(sql1,rm,14);

System.***out***.println(p);

}

}

***query:***

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ac=**new** AnnotationConfigApplicationContext(AppConfig.**class**);

JdbcTemplate t=(JdbcTemplate) ac.getBean("jd");

String sql1="select \* from prodtab";

RowMapper<Product> rm=(ResultSet rs, **int** rowNum)->{

**return** **new** Product(

rs.getInt("pid"),

rs.getString("pcode"),

rs.getString("pcost")

);

};

List<Product> p=t.query(sql1,rm);

System.***out***.println(p);

}

}

ORM:

-Object Relational Mapping

-Jdbc works on primitive data.

-Hibernate(ORM) works on object .

-programmer follow mapping rule.

-class link with table

-Variable link with column

-No sql by programmer .

- then one row converted to one object and vice versa.

Mapping with Annotations javax.persistence package only

@Entity

@Table(name="prodtab")

**public** **class** Example {

@Column(name="product id")

@GeneratedValue

**private** **int** prodid;

@Transient

**private** String prodCost;

@ElementCollection

**private** List names;

@Temporal(TemporalType.***DATE***)

Date d;

}

Spring orm reduce common lines code in application by using template design pattern

Internally uses HibernateTemplate

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.2.17.Final</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>5.0.6.RELEASE</version>

</dependency>

**import** java.util.Properties;

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

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

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

**import** org.springframework.core.env.Environment;

**import** org.springframework.jdbc.datasource.DriverManagerDataSource;

**import** org.springframework.orm.hibernate5.HibernateTemplate;

**import** org.springframework.orm.hibernate5.HibernateTransactionManager;

**import** org.springframework.orm.hibernate5.LocalSessionFactoryBean;

@Configuration

**public** **class** AppConfig {

@Autowired

**private** Environment e;

@Bean

**public** DriverManagerDataSource ds() {

DriverManagerDataSource d=**new** DriverManagerDataSource();

d.setDriverClassName(e.getProperty("dc"));

d.setUrl(e.getProperty("url"));

d.setUsername(e.getProperty("un"));

d.setPassword(e.getProperty("pwd"));

**return** d;

}

@Bean

**public** LocalSessionFactoryBean sf() {

LocalSessionFactoryBean s=**new** LocalSessionFactoryBean();

s.setAnnotatedClasses(Example.**class**);

s.setHibernateProperties(p());

s.setDataSource(ds());

**return** s;

}

@Bean

**public** Properties p() {

Properties pr=**new** Properties();

pr.put("hibernate.dialect", e.getProperty("dialect"));

pr.put("hibernate.show-sql",e.getProperty("show"));

pr.put("hibernate.format-sql",e.getProperty("format"));

pr.put("hibernate.hbmddl.auto", e.getProperty("ddl"));

**return** pr;

}

@Bean

**public** HibernateTemplate ht() {

HibernateTemplate t=**new** HibernateTemplate();

t.setSessionFactory(sf().getObject());

**return** t;

}

**public** HibernateTransactionManager hr() {

HibernateTransactionManager htm=**new** HibernateTransactionManager();

htm.setSessionFactory(sf().getObject());

**return** **null**;

}

}

**We can perform below operation in hibernate template:**

**Ex:**

package com.app.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

@Entity

@Table(name="uomtab")

public class Uom {

@Id

@GeneratedValue(generator="uompk")

@GenericGenerator(name="uompk",strategy="increment")

@Column(name="uid")

private Integer uomId;

@Column(name="utype")

private String uomType;

@Column(name="umodel")

private String uomModel;

@Column(name="udsc")

private String uomDesc;

public Uom() {

super();

}

public Uom(Integer uomId) {

super();

this.uomId = uomId;

}

public Integer getUomId() {

return uomId;

}

public void setUomId(Integer uomId) {

this.uomId = uomId;

}

public String getUomType() {

return uomType;

}

public void setUomType(String uomType) {

this.uomType = uomType;

}

public String getUomModel() {

return uomModel;

}

public void setUomModel(String uomModel) {

this.uomModel = uomModel;

}

public String getUomDesc() {

return uomDesc;

}

public void setUomDesc(String uomDesc) {

this.uomDesc = uomDesc;

}

@Override

public String toString() {

return "Uom [uomId=" + uomId + ", uomType=" + uomType + ", uomModel=" + uomModel + ", uomDesc=" + uomDesc + "]";

}

}

package com.app.dao;

import java.util.List;

import com.app.model.Uom;

public interface IUomDao {

public Integer saveUom(Uom u);

public void updateUom(Uom u);

public void deleteUom(Integer id);

public Uom getUomById(Integer id);

public List<Uom> getAllUoms();

public boolean isUomModelExist(String uomModel);

}

package com.app.dao.impl;

import java.util.List;

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

import org.springframework.orm.hibernate5.HibernateTemplate;

import org.springframework.stereotype.Repository;

import com.app.dao.IUomDao;

import com.app.model.ShipmentType;

import com.app.model.Uom;

@Repository

public class UomDaoImpl implements IUomDao {

@Autowired

private HibernateTemplate ht;

public Integer saveUom(Uom u) {

return (Integer) ht.save(u);

}

public void updateUom(Uom u) {

ht.update(u);

}

public void deleteUom(Integer id) {

ht.delete(new Uom(id));

}

public Uom getUomById(Integer id) {

return ht.get(Uom.class, id);

}

public List<Uom> getAllUoms() {

return ht.loadAll(Uom.class);

}

@Override

public boolean isUomModelExist(String uomModel) {

long count=0;

String hql=" select count(uomModel) "

+ " from " + Uom.class.getName()

+ " where uomModel=? ";

List<Long> list=(List<Long>) ht.find(hql, uomModel);

if(list!=null && !list.isEmpty()) {

count=list.get(0); }

return count>0?true:false;

}

}

package com.app.service;

import java.util.List;

import com.app.model.Uom;

public interface IUomService {

public Integer saveUom(Uom u);

public void updateUom(Uom u);

public void deleteUom(Integer id);

public Uom getUomById(Integer id);

public List<Uom> getAllUoms();

public boolean isUomModelExist(String uomModel);

}

package com.app.service.impl;

import java.util.List;

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

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.app.dao.IUomDao;

import com.app.model.Uom;

import com.app.service.IUomService;

@Service

public class UomServiceImpl implements IUomService {

@Autowired

private IUomDao dao;

@Transactional

public Integer saveUom(Uom u) {

return dao.saveUom(u);

}

@Transactional

public void updateUom(Uom u) {

dao.updateUom(u);

}

@Transactional

public void deleteUom(Integer id) {

dao.deleteUom(id);

}

@Transactional(readOnly=true)

public Uom getUomById(Integer id) {

return dao.getUomById(id);

}

@Transactional(readOnly=true)

public List<Uom> getAllUoms() {

return dao.getAllUoms();

}

@Transactional(readOnly=true)

public boolean isUomModelExist(String uomModel) {

return dao.isUomModelExist(uomModel);

}

}

package com.app.controller;

import java.util.Arrays;

import java.util.List;

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

import org.springframework.stereotype.Controller;

import org.springframework.ui.ModelMap;

import org.springframework.validation.Errors;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

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

import org.springframework.web.servlet.ModelAndView;

import com.app.model.Uom;

import com.app.service.IUomService;

import com.app.validator.UomValidator;

import com.app.view.UomExportViewController;

import com.app.view.UomPdfView;

@Controller

@RequestMapping("/uom")

public class UomController {

@Autowired

private IUomService service;

@RequestMapping("/register")

public String uomReg(ModelMap map) {

//creating model class and add to ModelMap

map.addAttribute("uom", new Uom());

return "UomRegister";

}

//insert data in db

@RequestMapping(value="/insert",method=RequestMethod.POST)

public String save(@ModelAttribute Uom uom,Errors errors,ModelMap map) {

//1.call validator

//2.check errors

if(errors.hasErrors()) {

map.addAttribute("message","please check all errors");

}else {

//call service method in Service Layer

int id=service.saveUom(uom);

String msg="succesfully registered with id"+id;

map.addAttribute("message",msg);

map.addAttribute("uom",new Uom());

}

return "UomRegister";

}

//fetch data from db to ui

@RequestMapping("/all")

public String viewAll(ModelMap map) {

List<Uom> ls=service.getAllUoms();

map.addAttribute("list",ls);

return "UomData";

}

//delete row based on id

@RequestMapping("/delete")

public String delete(@RequestParam Integer uid,ModelMap map) {

service.deleteUom(uid);

//read new data

List<Uom> ls=service.getAllUoms();

map.addAttribute("list",ls);

//add message to display

map.addAttribute("message","Record deleted"+uid);

return "UomData";

}

//5.view One Row or Object

@RequestMapping("/viewOne")

public String viewOne(@RequestParam Integer sid,ModelMap map) {

Uom ob=service.getUomById(sid);

map.addAttribute("st",ob);

return "UomView";

}

//6.show Edit Page

@RequestMapping("/editone")

public String showEdit(@RequestParam Integer sid,ModelMap map) {

//load db rows as Object

Uom st=service.getUomById(sid);

//send to ui

map.addAttribute("uom",st);

return "UomEdit";

}

//do update

@RequestMapping(value="/update",method=RequestMethod.POST)

public String doUpadte(@ModelAttribute Uom uom,ModelMap map) {

//call service Update

service.updateUom(uom);

//success msg

map.addAttribute("message","Uom Updated");

//get new Data

map.addAttribute("list",service.getAllUoms());

return "UomData";

}

//8.excelExport

@RequestMapping("/excelExport")

public ModelAndView doExcelExport() {

//call service

List<Uom> uom=service.getAllUoms();

//key,value and view

return new ModelAndView(new UomExportViewController(),"all",uom);

}

//8.excelExport

@RequestMapping("/excelExportOne")

public ModelAndView doOneExcelExport(@RequestParam Integer id) {

Uom uom=service.getUomById(id);

return new ModelAndView(new UomExportViewController(),"all",Arrays.asList(uom));

}

//9.export pdf all

@RequestMapping("/pdfAll")

public ModelAndView exportPdf() {

List<Uom> uom=service.getAllUoms();

return new ModelAndView(new UomPdfView(),"list",uom);

}

//10.export pdf one

@RequestMapping("/pdfone")

public ModelAndView exportPdfOne(@RequestParam Integer id) {

Uom uom=service.getUomById(id);

return new ModelAndView(new UomPdfView(),"list",Arrays.asList(uom));

}

}

Refer above code in : <https://github.com/narendradasara99/WhareHouse-Project-Final-Code>

*Why should we prefer id as Integer not int in model class:*

**If we take int as id**

In ClientProgram.java, i have been written setters for productId, proName, but i have not written setter for price

But once you execute this program in the database it will saves the price as 0(zero), so misunderstanding of data will happen like watch price is zero [ free of cost hah ]

**If we take Integer as id**

See in this case if we forget to write the setter for the price, in the database its not inserting any thing [ actually it has to insert NULL value, as of now leave it ],  no way of data misunderstanding.

But in the first case (primitive types) it inserted zero, see the above screen short

Referal link: <https://www.java4s.com/hibernate/difference-between-wrapper-and-primitive-types-in-hibernate/>

Spring MVC

Spring g Mvc Folder Structure:

AppInit,java

This file is used to configure front controller url and provide spring config file

**package** com.app.config;

**import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

**public** **class** AppInit **extends** AbstractAnnotationConfigDispatcherServletInitializer {

@Override

**protected** Class<?>[] getRootConfigClasses() {

// **TODO** Auto-generated method stub

**return** **new** Class[] {AppConfig.**class**};

}

@Override

**protected** Class<?>[] getServletConfigClasses() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**protected** String[] getServletMappings() {

// **TODO** Auto-generated method stub

**return** **new** String[] {"/"};

}

}

2) AppConfig.java : this file is used to configure the beans .. like HibernateTemplate, InternalResorceviewResolver ..etc.

**package** com.app.config;

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

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

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

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

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

**import** org.springframework.core.env.Environment;

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

@PropertySource("classpath:app.properties")

@ComponentScan(basePackages="com.app")

@EnableWebMvc

**public** **class** AppConfig {

@Autowired

**private** Environment e;

@Bean

**public** InternalResourceViewResolver ivr() {

InternalResourceViewResolver i=**new** InternalResourceViewResolver();

i.setPrefix(e.getProperty("prefix"));

i.setSuffix(e.getProperty("suffix"));

**return** i;

}

}

3) Controller.java: this class is used to request the data and response the data.

Every method binding to the http methods.

**package** com.app.config;

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

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

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

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

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

**import** org.springframework.core.env.Environment;

**import** org.springframework.web.servlet.config.annotation.EnableWebMvc;

**import** org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration

@PropertySource("classpath:app.properties")

@ComponentScan(basePackages="com.app")

@EnableWebMvc

**public** **class** AppConfig {

@Autowired

**private** Environment e;

@Bean

**public** InternalResourceViewResolver ivr() {

InternalResourceViewResolver i=**new** InternalResourceViewResolver();

i.setPrefix(e.getProperty("prefix"));

i.setSuffix(e.getProperty("suffix"));

**return** i;

}

}

App.properties

prefix=/WEB-INF/views/

suffix=.jsp

Welcome.jsp:

<html>

<body>

welcome ${msg}

</body>

</html>

Pom.xml:

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

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

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>naren</groupId>

<artifactId>Mvc1Ex</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>Mvc1Ex Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

</dependencies>

<build>

<finalName>Mvc1Ex</finalName>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

</plugins>

</build>

</project>

Spring Form Tag library:

These are custom tags given by sping frame work to design ui page using

a) Bi directional form:

* form data can be converted to Object and object data can be converted to form
* every form created using spring tags is connected with model class object. It means every form contain a back one object is also called Form Backing Object .
* every column name and form tag path name must be same .

Spring Form Tag Library:

1. pom.xml

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>5.1.8.RELEASE</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

<plugin>

<artifactId>maven-war-plugin</artifactId>

<version>2.4</version>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

</plugins>

</build>

2. Model class

**package** com.app.model;

**import** java.util.List;

**public** **class** Student {

**private** Integer sid;

**private** String sname;

**private** String gen;

**private** String course;

**private** String addr;

**private** List<String> langs;

**public** Student() {

**super**();

}

**public** Integer getSid() {

**return** sid;

}

**public** **void** setSid(Integer sid) {

**this**.sid = sid;

}

**public** String getSname() {

**return** sname;

}

**public** **void** setSname(String sname) {

**this**.sname = sname;

}

**public** String getGen() {

**return** gen;

}

**public** **void** setGen(String gen) {

**this**.gen = gen;

}

**public** String getCourse() {

**return** course;

}

**public** **void** setCourse(String course) {

**this**.course = course;

}

**public** String getAddr() {

**return** addr;

}

**public** **void** setAddr(String addr) {

**this**.addr = addr;

}

**public** List<String> getLangs() {

**return** langs;

}

**public** **void** setLangs(List<String> langs) {

**this**.langs = langs;

}

@Override

**public** String toString() {

**return** "Student [sid=" + sid + ", sname=" + sname + ", gen=" + gen + ", course=" + course + ", addr=" + addr

+ ", langs=" + langs + "]";

}

}

3. Controller

**package** com.app.controller;

**import** **static** org.springframework.web.bind.annotation.RequestMethod.***POST***;

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

**import** org.springframework.ui.ModelMap;

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

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

**import** com.app.model.Student;

@Controller

**public** **class** StudentController {

// /register -> StudentReg.jsp

@RequestMapping("/register")

**public** String showPage(ModelMap map) {

//form backing object

Student s=**new** Student();

/\*

s.setSid(111);

s.setSname("AJAY");

s.setGen("Male");

\*/

map.addAttribute("student", s);

**return** "StudentReg";

}

@RequestMapping(value="/save",method=***POST***)

**public** String readForm(

@ModelAttribute Student student,

ModelMap map) {

map.addAttribute("ob",student);

**return** "Data";

}

}

4. UI (JSP)

Student.jsp

<%@taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

<html>

<body>

<h3>Welcome to Student Register Page</h3>

<form:form action=*"save"* method=*"POST"* modelAttribute=*"student"*>

<pre>

ID : <form:input path=*"sid"*/>

NAME : <form:input path=*"sname"*/>

GENDER:

<form:radiobutton path=*"gen"* value=*"Male"*/> Male

<form:radiobutton path=*"gen"* value=*"Female"*/> Female

COURSE:

<form:select path=*"course"*>

<form:option value=*""*>-SELECT-</form:option>

<form:option value=*"Core"*>Core</form:option>

<form:option value=*"Adv"*>Adv</form:option>

<form:option value=*"Hibernate"*>Hibernate</form:option>

<form:option value=*"Spring"*>Spring</form:option>

</form:select>

ADDRESS: <form:textarea path=*"addr"*/>

LANGUAGES:

<form:checkbox path=*"langs"* value=*"English"*/> English

<form:checkbox path=*"langs"* value=*"Hindi"*/> Hindi

<form:checkbox path=*"langs"* value=*"Telugu"*/> Telugu

<input type=*"submit"* value=*"Register"*/>

</pre>

</form:form>

</body>

</html>

Data.jsp

<html>

<body>Data is:${ob}

</body>

</html>

b) form validation api:

validation is used to avoid the invalid data using inputs of data .

validations can done in two ways :

1) client side validation:done at browser generally using javascript, jquery..etc.

2) server side validation: once request is received by server then server data is violated before processing it.

Generally pattern validations are done at client side and data existence validation check at server side.

But spring supports both at server side regular expressions.

Spring Form Validator API Example:-

**Model class:**

**package** com.app.model;

**import** java.util.List;

**public class** Employee {

**private** Integer empId;

**private** String empName;

**private** String empGen;

**private** String empProj;

**private** String empAddr;

**private** List<String> empLangs;

**public** Employee() {

**super**();

}

**public** Integer getEmpId() {

**return** empId;

}

**public void** setEmpId(Integer empId) {

**this**.empId = empId;

}

**public** String getEmpName() {

**return** empName;

}

**public void** setEmpName(String empName) {

**this**.empName = empName;

}

**public** String getEmpGen() {

**return** empGen;

}

**public void** setEmpGen(String empGen) {

**this**.empGen = empGen;

}

**public** String getEmpProj() {

**return** empProj;

}

**public void** setEmpProj(String empProj) {

**this**.empProj = empProj;

}

**public** String getEmpAddr() {

**return** empAddr;

}

**public void** setEmpAddr(String empAddr) {

**this**.empAddr = empAddr;

}

**public** List<String> getEmpLangs() {

**return** empLangs;

}

**public void** setEmpLangs(List<String> empLangs) {

**this**.empLangs = empLangs;

}

@Override

**public** String toString() {

**return** "Employee [empId=" + empId + ", empName=" + empName + ", empGen=" + empGen + ", empProj=" + empProj

+ ", empAddr=" + empAddr + ", empLangs=" + empLangs + "]";

}

}

**Controller:-**

@Controller

**public class** EmployeeController {

@Autowired

**private** EmployeeValidator validator;

//1. show Form

@RequestMapping("/register")

**public** String showPage(ModelMap map) {

//Form Backing Object

map.addAttribute("employee",**new** Employee());

**return** "Register";

}

//2. on click submit

@RequestMapping(value="/save",method=RequestMethod.***POST***)

**public** String processData(

@ModelAttribute Employee employee,

Errors errors,

ModelMap map)

{

String page=**null**;

//1. call vlidator

validator.validate(employee,errors);

//2. check for errors exist

**if**(errors.hasErrors()) {

page="Register";

}else {

page="Data";

map.addAttribute("emp",employee); }

**return** page;

}

}

**Validator:-**

**package** com.app.validator;

**import** java.util.regex.Pattern;

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

**import** org.springframework.util.StringUtils;

**import** org.springframework.validation.Errors;

**import** org.springframework.validation.Validator;

**import** com.app.model.Employee;

@Component

**public class** EmployeeValidator

**implements** Validator {

@Override

**public boolean** supports(Class<?> clz) {

**return** Employee.**class**.equals(clz);

}

@Override

**public void** validate(Object target, Errors errors) {

//Down cast to Model class

Employee e=(Employee)target;

//Text Input Validation

**if**(!Pattern.*matches*("[A-Z]{2,6}",e.getEmpName())){

errors.rejectValue("empName", **null**, "Please enter valid name");

}

//Radio Button

//if(e.getEmpGen()==null || "".equals(e.getEmpGen())) {

**if**(!StringUtils.*hasText*(e.getEmpGen())) {

errors.rejectValue("empGen", **null**, "Please choose one Gender");

}

//Text Area

**if**(!Pattern.*matches*("[A-Za-z0-9\\s]{2,50}", e.getEmpAddr())) {

errors.rejectValue("empAddr", **null**,"Please Enter Address");

}

//DropDown

**if**(!StringUtils.*hasText*(e.getEmpProj())) {

errors.rejectValue("empProj", **null**, "Please choose one Project");

}

//Checkbox : List

**if**(e.getEmpLangs()==**null** || e.getEmpLangs().isEmpty()) {

errors.rejectValue("empLangs", **null**,"Please choose one Language");

}

}

}

Register.jsp

<%@taglib prefix=*"form"* uri=*"http://www.springframework.org/tags/form"* %>

<html>

<head>

<style type=*"text/css"*>

*.err{*

color: *red*;

}

</style>

</head>

<body>

<h3>Welcome to Employee Register page</h3>

<form:form action=*"save"* method=*"POST"* modelAttribute=*"employee"*>

<pre>

Employee Id : <form:input path=*"empId"*/>

Employee Name : <form:input path=*"empName"*/>

<form:errors path=*"empName"* cssClass=*"err"*/>

Employee Gender: <form:radiobutton path=*"empGen"* value=*"Male"*/>Male <form:radiobutton path=*"empGen"* value=*"Female"*/>Female

<form:errors path=*"empGen"* cssClass=*"err"*/>

Employee Address: <form:textarea path=*"empAddr"*/>

<form:errors path=*"empAddr"* cssClass=*"err"*/>

Employee Project:

<form:select path=*"empProj"*>

<form:option value=*""*>-SELECT-</form:option>

<form:option value=*"P1"*>P1</form:option>

<form:option value=*"P2"*>P2</form:option>

</form:select>

<form:errors path=*"empProj"* cssClass=*"err"*/>

Employee Language:

<form:checkbox path=*"empLangs"* value=*"English"*/> English

<form:checkbox path=*"empLangs"* value=*"Hindi"*/> Hindi

<form:checkbox path=*"empLangs"* value=*"Telugu"*/> Telugu

<form:errors path=*"empLangs"* cssClass=*"err"*/>

<input type=*"submit"* value=*"Register"*/>

</pre>

</form:form>

</body>

</html>

**Data.jsp**

Data is: ${emp}

**Handler Mapper:**

HandlerMapper is interface Spring IOC called their implementation class in runtime.

Which holds data in key and value format.

In simple words for what url which method is should be called such details are stored

Here key is url and value is classname- methodname

Ex: /show EmpConltroller-showemp()

/delete EmpController-deleteemp()

Front Controller operation:

Front Controller (AbstractAnnotationConfigDispatcherServletInitializer) is a class is a predefind servlet. Operations are

1. Get the request
2. Get controller details from HandlerMapper
3. Get controller object from IOC
4. Call ( request) the method based on matching url
5. Get view name back
6. Goto ViewResolver add suffix and prefix to the given view name
7. Get ui page
8. Finally ui page is response

* One method connect with mutiple urls

@RequestMapping(value= {"/get","/show","/view"})

**public** ModelAndView show() {}

<http://localhost:8088/Mvc1Ex/view>

<http://localhost:8088/Mvc1Ex/show>

<http://localhost:8088/Mvc1Ex/get>

* One method connect with mutiple urls binding with multiple methods as ashown below

@RequestMapping(value= {"/get","/show","/view"},method= {RequestMethod.***GET***,RequestMethod.***POST***})

**public** ModelAndView show() {}

* By using static import we can write above code like this

**import** **static** org.springframework.web.bind.annotation.RequestMethod.***POST***;

**import** **static** org.springframework.web.bind.annotation.RequestMethod.***GET***;

@RequestMapping(value= {"/get","/show","/view"},method= {***GET***,***POST***})

**public** ModelAndView show() {**}**

* Reading servlet container objects is possible in Controller by using @Autowired or method parameter

@Controller

**public** **class** ExController {

@Autowired

**private** ServletContext s;

@Autowired

**private** HttpServletRequest req;

@GetMapping("/get")

**public** ModelAndView show(ServletConfig c) {}

}

* Project setup:

1. Add jdk to Workspace🡪window🡪preferrence🡪installed jre🡪click on standard vm🡪select jdk file location🡪finish
2. Server setup(tomcat)🡪window🡪show view🡪server🡪clink on server🡪 environment select jdk 🡪finish
3. Create a project🡪 create new 🡪maven🡪search with web app🡪enter details🡪finish
4. Add server to the project🡪configure build path🡪add library🡪server runtime🡪tomcat🡪apply🡪close
5. Remove web.xml and index.jsp
6. In pom add required dependencies
7. Set facet problem🡪 show view 🡪navigator🡪choose expand project

🡪open .settings🡪 open file .xml🡪click on source🡪modify jst.web=3.1 and java =1.8🡪back to package explorer🡪update project(alt+f5)

1. If src/java missing 🡪click on project🡪configure buildpath🡪select jre and edit🡪change system library to work space default jre
2. If you have any dependency problem we need to check folder C:\Users\naren\.m2\repository\org\springframework🡪and proper dependency means same version

* Coding order:

1. App.properties
2. AppConfig.java
3. AppInit.java
4. HomeController.java
5. Welcome jsp

* Browser supports 3 requests, those are

1. Enter url in address bar(Get)

<http://localhost:8088/Mvc1Ex/view>

1. html form submit (Get/Post)

<form action=*"/show"* method=*"get"*></form>

1. Clink on link( hyperlink)(Get)

<a href=*"logout"*>logout</a>

1. But every request is processed by Controller class only. Based on url+ method type(Get/Post) controller executed by front controller.

* Controller class in spring MVC:

Controller is a class which is annotated with @Controller( Stereo type annotation) will support

1. Detect the class (by container)
2. Crete object
3. Provide data to variables
4. Process the request

Controller will have methods those are called request methods. every method will be linked with url and method type. That method returns Model and View. Controller class object is call by spring container. Method also call by spring container when request made by Browser

Sending Data From controller to ui

* By using ModelAndView we can send data from controller to ui.
* In ModelAndView , model means data which holds in map format.
* Here key is string type and value is java.lang.Object type.

@GetMapping("/get")

**public** ModelAndView show() {

ModelAndView v=**new** ModelAndView();

String name=" narendra ";

v.addObject("msg", name);

v.setViewName("Welcome");

**return** v;

}

In ui page:

We can get data from controller to ui

<html>

<body>

welcome ${msg}

<!--above and below are the same below we can get data from implicit object -->

<% Object ob= request.getAttribute("msg");

out.println(ob);

%>

</body>

</html>

In ModeLAndView view name is mandatory and data is optional.

ModelAndView allocates memory for model if there is not exist also. So this problem spring framework provides model map and string in case of model and view.

ModelMap

So spring framework provides ModelMap(on demand memory allocation) which allocates memory by container. Now method return type is String and data is ModelMap

@GetMapping("/get")

**public** String show(Model v) {

String name=" narendra ";

v.addAttribute("msg",name);

**return** "Welcome";

}

Sending Data From ui to controller

We can send primitive data or object data from ui.

* Parameters : these are used to send primitive data a by using url , if follows key value format.

Like: url?key=value& key=val......

Servlet code look like:

String name = req.getParameter(“key”);

In spring we can write like

@RequestParam(“key”) String val;

Request variable takes from anchr tag and queryparam will takes from query like below

http://localhost:8088/Mvc1Ex/get/narendra

@GetMapping("/get/{param}")

**public** String show(Model v, @PathVariable("param") String n) {

v.addAttribute("msg",n);

**return** "Welcome";

}

Op: welcome narendra

1) by default every request id required .if key is not present in url...then FC throw error.400

http://localhost:8088/Mvc1Ex/get

2) to make key is optional and property required is true.

@GetMapping("/get")

**public** String show(Model v, @RequestParam(value="param" ,required=**false**) String n) {

v.addAttribute("msg",n);

**return** "Welcome";

}

Op: welcome null

3) in above case value is null if key is not present on url. In this situation we can handle default value

@GetMapping("/get")

**public** String show(Model v, @RequestParam(value="param" ,required=**false**,defaultValue="no data found send data") String n) {

v.addAttribute("msg",n);

**return** "Welcome";

}