SPRING FRAMEWORK

Coupling :→

The dependency between two applications or 2 classes or 2 objects is called as coupling.

⇒ In other words coupling can also be reffered as the amount of knowledge one class having about another class.

⇒ we have two types of coupling.

1. Tight coupling
2. loose coupling.

1 Tight coupling :→

If the change in application or class affects the dependent application or class then it is called as tight coupling.

⇒ If two classes or applications change simultaneously. it can be called Tight coupling.

Ex :—

ⓐ
```
public class engine
{

  public void start()
  {
    S.o.p("Engine started");
  }
}
```

ⓑ
```
public class car
{
  private Engine e = new Engine();
  public void starter(){

      e.start();
}
```

ⓒ
```
public class run
{ p.s.v.m()
  {
    car c = new car()

    c.starter();
  }
}
```

2 Loose Coupling : ⟶

If the change in application or class is not affecting the dependent application or class, then it is called as loose coupling.

Example to understand loose coupling :—

① public interface Vehicle
{
    void start();
}

② public class Bike implements Vehicle
{
    @Override
    public void start()
    {
        S.o.p ("Bike has been started");
    }
}

③ public class car implements Vehicle
{
    @Override
    public void start()
    {
        S.o.p ("car has been started");
    }
}

④ public class Ride {
    private vehicle v;

    // public getters & setters
}

```
public class TestRide
{ public static void main (String[] args)
  {
    Ride r = new Ride();
    r.setV(new car());  // dependency injection
    r.getV().start();
  }
}
```

Dependency Injection :→

⇒ The process of injecting the resources required for an object during the runtime to achieve the loose coupling is called as dependency Injection.

⇒ In Object oriented programming dependency injection is nothing but the process of injecting the reference of one object to another object during the runtime.

⇒ we can achieve the dependency injection with the help of Spring Containers present in spring Framework.

IOC (Inversion Of Control) :→

⇒ The process of transfering the control of creating the Objects (Instantiation), initialization of objects and injecting the resources required for object and managing the lifecycle of an Object from one part of the application to the another part of same application, so that the part which is transferring the control can focus on important task is called as IOC.

⇒ We can transfer the controls of instantiation, initialization, dependency Injection and managing lifecycle to Spring Containers present in Spring framework.

Spring : ⟶

⇒ Spring is open-source, non-invasive, lightweight application framework which work on a principle of ~~control~~ Inversion of Control (IOC) and dependency injection using which we can develop the Loosely coupled application.

⇒ By using Spring framework we can specify the development of enterprise applications.

⇒ Spring can also be called as framework of frameworks as it support various JEE specifications and framework such as JPA, Hibernate JDBC ... etc.

⇒ Spring can also be offered as the solution for the various technical problems that we used to face in JEE specifications

Modules of Spring framework : ⟶

1. Spring Core
2. Spring context
3. Spring MVC (Model view controller)
4. Spring ORM (Object Relational Mapping)
5. Spring AOP (Aspect Oriented programming)
6. Spring DAO (Data Access Object)

Spring

1.0
2.0
3.0
4.0
5.0
6.0

Advant

1. It
2. It
3. B
4. S

(i)

(ii)

Spring Framework Version History

1.0 → Was released in 2004
2.0 → in 2006
3.0 → in 2009
4.0 → in 2013
5.0 → in 2017
6.0 → in 2022

Advantages of Spring Framework

1. It is an open-source framework
2. It is a light-weight framework as it makes use of POJO ~~Imple me~~ Implementation.
3. By using Spring framework we can develop loosely coupled application as it supports dependency Injection.
4. Spring framework provide the pre-define templates using which we can avoid all the basic steps of execution that we were supposed to write in JDBC, Hibernate, JPA etc..
5. (i) If we use JdbcTemplate provided by Spring framework we can avoid the steps like loading and registering the driver class, establishing the connection, creating a statement and closing all the costly resources which are supposed to be written before and after the execution of query.

(ii) If we use HibernateTemplate we can avoid the creation of Configuration, SessionFactory, Session, Transaction and committing the transaction which are supposed to be written before and after saving the record.

5. Spring Applications are easy to test and develop as it supports dependency Injection.

6 Spring framework provide the strong Abstraction for various JEE specification such as JPA, JMA, JMS, etc.

ASSIGNMENT

1. What is container spring ?
2. What is Spring Bean ?
3. What is Java Bean ?
4. Different Between Java Bean and Spring Bean ?
5. Explain URL, URI and URN ?

Spring Container : ⟶

⇒ spring container is a factory of spring beans which is used to instantiate spring beans, initialize spring beans, it will inject the resources require for spring beans and it will manage the lifecycle of spring bean.

⇒

Spring Bean : ⟶

⇒ Spring bean is an object of POJO class created by Spring container initialize by spring container, resource of this object is injected by spring container and lifecycle of this object is manage by spring container.

⇒ If you want a spring Container to instantiate, initialize, inject the resources and manage the lifecycle of a Spring bean, you must provide the ~~act meta~~ - META - DATA about the POJO class to the spring container

⇒ The META-DATA can be provided to the Spring Container in the following ways

1. XML 2. Annotation 3. Java Code

Types of Spring Containers :—

1. IOC ~~Creation~~ Container
2. MVC Container

1. IOC Container

Core Container / Spring BeanFactoryContainer
(BeanFactory)

JEE Container / Spring ApplicationContext Container
(ApplicationContext)

Difference Between BeanFactory and ApplicationContext

BeanFactory	
1.	It is a root interface which has to be implemented by all Spring Containers
2.	This interface will provide all basic functionalities using which we can develop stand alone application
3.	It is lazy and creates the Spring beans only on the demand.
4.	It support only Singleton and prototype bean scope.
5.	Autowiring by BeanFactory can be done only By XML

BeanFactory :⟶

⇒ it is an interface present in spring framework.
⇒ This interface is the root interface for all the spring containers.
⇒ This interface will provide the basic functionality which are suitable for the development of stand-alone applications.
⇒ This interface is lazy and creates the spring bean only after the call of getBean().
⇒ This interface represents the core container.
⇒ It will support only SingTon and prototype bean Scopes.
⇒ Autowiring by this container can be achieve only by XML.
⇒ .

Application Context : ⟶

⇒ It is an interface belongs to spring framework.
⇒ It is the sub-interface of listable BeanFactory, which is the child of BeanFactory interface.
⇒ This interface represents the JEE container in spring framework
⇒ This container will have all the functionalities of BeanFactory and also few additional functionalities which are suitable for the development of Enterprise applications.
⇒ This container is eager and creates the spring bean on startup.
⇒ This container will support BeanScopes.
⇒ autowiring by this container can be achieve by XML as well as annotation.

getBean() :⟶

⇒ It is a method declared in BeanFactory Interface.

⇒ This method will return the spring Beans created by the Spring Container.

⇒ This method is Overloaded and following are the overloaded methods of getBean().

Method Signature	returnType
getBean(String)	java.lang.Object
getBean(String, class<T>)	T
getBean(class<T>)	T

What is BeanScope ?

Ans :⟶

⇒ Bean Scope is used to specify the numbers of times a spring has to be created or to specify the scope or visibility of the Spring Bean.

⇒ Following are the spring bean scopes available.

~~1. Sinfleton~~

1. Singleton
2. prototype.
3. request.
4. session
5. global-session

1. Singleton :⟶

⇒ If the bean scope is Singleton, we will get same object everyTime we will call getBean().

⇒ The spring bean will be created only once per a Spring container.

2. Prototype : ⟶

⇒ if a spring bean is created every time we call getBean(), then the bean scope is prototype.

3. request : ⟶

⇒ A new spring bean will be created for every HttpRequest if the bean scope is request and it is mostly used in web applications.

4. Session : ⟶

⇒ A new spring bean will created for every HttpSession if the bean scope is session and this one also used in web applications.

5. global-session : ⟶

⇒ This bean scope is mostly used in web ~~sockets~~ Sockets.

Steps to Configure Spring bean

Step 1 : Create a simple maven projects by specifying the groupId and Artifact Id.

Step 2 : Adding Spring-context (5.3.20) in pom.xml

Step 3 : Create the POJO class in src/main/~~resource~~ java folder

```
package org.jsp.springpractice;

public class Car {
    public void start(){
        System.out.println("car has been started");
    }
}
```

Step 4: Create an XML file in src/main/resources to configure Spring bean.

```
<beans...>
    <bean id="mycar" class="org.jsp.springpractice.Car">
    </bean>
</beans>
```

Step 5: Call getBean() of BeanFactory to use the Spring Bean.

Resource :⟶

⇒ It is an interface belongs to spring framework.

⇒ This interface is used to represents a resource in which the Spring Beans have been configured.

⇒ Following are the important implementation classes of Resource

1. ClassPathResource
2. FileSystemResource

Note:— XmlBeanFactory is an implementation class of BeanFactory.

Note:— Following are the important implementation classes of ApplicationContext interface.

(i) ClassPathXmlApplicationContext

(ii) FileSystemXmlApplicationContext.

(iii) AnnotationConfigApplicationContext.

Test.java

```
public class Test
{
  public static void main(String[] args)
  {
    Resource r = new ClassPathResource("car.xml");
    BeanFactory factory = new XmlBeanFactory(r);
  // Car c1 = (Car) factory.getBean("myCar");
  // Car c1 = factory.getBean("myCar", Car.class);
    Car c1 = factory.getBean(Car.class);
      c1.start();
  }
}
```

Spring Bean Initialization By XML

⇒ The Spring Bean By XML can be initialization in 2 ways

1. By using Setter.
2. By using Constructor.

1. By using Setter :⟶

⇒ If you want a Spring Bean to be initialized By Setter we need to use <proterty> element which is a child of <bean> element.

⇒ The Spring Bean class must have a public getter and setter for each field which has to be initialized else we will get BeanCreationException with the root cause NotWritablePropertyExcept

⇒ If you want to initialize a field
id → It must be public setter method ❤ setId(),
name → must be setName(), phone must have setPhone() - etc.

Example program to understand initialize spring Bean by setter :→

User.java

```
public class User
{ private String UserName;
  private String password;
   public void display()
   {
     System.out.println("UserName : " + UserName);
   }
  // getter & setter
}
```

XML code :—

```
<beans>
   <bean id = "user" class = "org.JSP.SpringPractice" >
    < Property name "UserName" value = "ABC" />
    < property name = "password" value = "Aka123" />
    </bean>
</beans>
```

TestUser.java

```
public class TestUser
{
  public static void main (String[] args)
  {
    Resource r = new ClassPathResource ("car.xml");
    BeanFactory factory = new XmlBeanFactory(r);
    User user = factory.getBean("user", User.class);
    user.display();
  }
}
```

Spring Bean initialization by using constructor ⟹ using Spring XML

⇒ In Spring XML we can configure the spring container to initialize a spring bean by constructor by using element <constructor-arg /> which is the child element of <bean>.

⇒ If we using <constructor-arg /> to initialize the spring bean the spring bean class must have a parameterized constructor, else we will get exception.

⇒ we can make use of the following attribute in <constructor-arg /> to specify the value for parameter

1. Index
2. Type
3. Name.

Example to understand Spring bean initialization by constructor using Spring XML :—

PanCard.java

```
package org.jsp.springpractice;
public class PanCard
{
 private String number;
 private int pinCode;

 public PanCard(){

 }
 public PanCard (String number , int pincode)
 {
    this.number = number;
    this.pinCode = pincod;
 }

 // override toString() method
 // getters & setters
}
```

pancard.xml

```
<beans>
  <bean id = "pancard" , class = "org.jsp.springpractic.PanCard" >

    <Constructor-arg name = "number" , value = "ABCD123" />
    <Constructor-arg name = "pincode" , value = "821105" />

  </bean>
</beans>
```

TestCard.java

```
public class TestCard
{
  public static void main (String[] args)
  {
    Resource r = new ClassPathResource ("pancard.xml");
    BeanFactory factory = new XmlBeanFactory (r);
    PanCard card = factory.getBean ("pancard", panCard.class);
    System.out.println (card);
  }
}
```

Dependency Injection By Spring XML : ⟶

⇒ The dependency injection By spring XML can be achieve in 2 way

1. Setter Injection
2. Constructor Injection.

1. Setter Injection : ⟶

⇒ If a Spring container injects the resources required for a Spring Bean by calling public setter methods, then it is called as setter injection.

⇒ The Setter injection by XML can be achieved by using the element <property> which is child of <bean> element.

⇒ The setter injection can be achieved in 2 ways

1. Nested Bean.
2. ref attribute.

Example to understand Setter Injection by using ref attribute.

Engine.java

```
public class Engine
{  public void startU
   {
     System.out.println("Engine is started");
   }
}
```

Car.java

```
public class car
{
  private Engine e;

  //public getters & setters
}
```

Car-engine.xml

```
<beans>
   <bean id="myCar" class="org.Spring.Car">
     <property name="e" ref="myEngine" />
   </bean>
   <bean id="myEngine" class="org.Spring.Engine">
   </bean>
</beans>
```

TestCar.java

```
public class TestCar
{
  public static void main(String[] args)
  {
    ApplicationContext context =
            new ClassPathXmlApplicationContext("car-engine.xml");
    Car c = context.getBean("myCar", Car.class);
    c.getE().start();
  }
}
```

XML code to achieve setter injection by using nested bean :-

```
<bean id="yourCar" class="org.spring.car">
    <property name="e">
        <bean class="org.spring.Engine" </bean>
    </property>
</bean>
```

TestCar.java

```
public class TestCar
{
  public static void main(String[] args)
  {
    ApplicationContext context =
            new ClassPathXmlApplicationContext("car-Engine
                                                .xml");
    Car c = context.getBean("yourCar", Car.class);
    c.getE().start();
  }
}
```

~~Construction~~

Constructor Injection : ⟶

⇒ The process of spring container injecting the resources required for an object by using a constructor is called as constructor injection.

⇒ The constructor injection by xml can be achieved by using the element <constructor-arg/> which is child of <bean>

⇒ Again the constructor injection achieve in 2 ways.

1. nested bean
2. ref attribute

Example program to understand constructor injection by ref attribute and nested bean :—

PanCard.java

```
public class PanCard
{
  private String number;
  private int age;

  // public getters & setters

  // toString
}
```

Person.java

```
public class Person
{
  private PanCard panCard;

  public Person() {
  }
  public Person (Pancard panCard) {
    this.panCard = panCard;
```

```
    // getters & setters
}
```

person_card.xml

```
<beans>
    <bean id = "card" class = "org.spring.panCard" >
        <property name = "number" value = "ABC123" />
        <property name = "age" value = "25" />
    </bean>
    <bean id = "person1" class = "org.spring.Person" >
        <constructor-arg type = "org.spring.PanCard" index = "0" ref = "card" />
    </bean>
    <!-- Constructor Injection by using nested bean -->
    <bean id = "person2" class = "org.spring.Person" >
        <constructor-arg name = "panCard" >
            <bean class = "org.spring.PanCard" >
                <property name = "number" value = "ACBC123A" />
                <property name = "age" value = "65" />
            </bean>
        </constructor-arg>
    </bean>
</beans>
```

TestPanCard.java

```
public class TestPanCard
{public static void main(String[] args)
 {
   ApplicationContext context =
      new ClassPathXmlApplicationContext("person-card.xml");
   Person P1 = context.getBean("Person1", Person.class);
   Person P2 = context.getBean("Person2", Person.class);
   System.out.println(P1.getPanCard());
   System.out.println(P2.getPanCard());
 }
}
```

Autowiring : ⟶

⇒ The process of implicit dependancy injection done by Spring container either by using setter injection or by using constructor injection is called autowiring.

⇒ The Autowiring can be achieve in 2-ways:

1. By xml
2. Annotation

⇒ ~~There~~ Three ~~modes~~ modes of Autowiring

1. ByName
2. ByType
3. Constructor

Autowiring By XML : ⟶

⇒ The Autowiring by XML can be achieved by using the attribute autowire which is present in the element <bean>

⇒ Following are the Autowiring modes supported by Spring Container.

(i) byName
(ii) byType
(iii) Constructor
(iv) default/no

(i) byName :—

⇒ In this mode of Autowiring Spring Container will try to inject the dependency by finding a bean whose id is same as the reference name.

⇒ Autowiring byName internally uses setter injection.

⇒ If there is no qualifying bean whose id is same as reference name, Spring container will not inject the dependency.

⇒

(ii) ByType :—

⇒ In this mode of autowiring Spring Container will try to find a qualifying bean whose type is same as reference type.

⇒ If there are no qualifying beans present, it will not inject the dependency.

⇒ If there are more than one qualifying bean present for Single ~~whose type is~~ reference type, then we will get UnsatisfiedDependency Exception with the root cause NoUniqueBeanDefinitionException.

⇒ If there are more than one qualifying bean present for a single reference type, we can use the attribute 'primary = "true"' inside <bean> to specify the primary bean which has to be injected.

⇒ This mode of autowiring will use setter injection internally.

(iii) Constructor : ⟶

⇒ In this mode of autowiring Spring Container will inject the dependency by calling a constructor which is having most number of parameters.

(iv) default / no : ⟶

⇒ This will disable autowiring.

Example to understand Autowiring by xml:

Car.java

```
public class car
{
  private Engine engine;

  public car()
  {
  }

  public car(Engine engine)
  {
    this.engine = engine;
  }

  //getters and setters
}
```

Engine.java

```
public class Engine
{
  public void start()
  {
    System.out.println("Engine start");
  }
}
```

```
<beans>
    <bean id = "eng" class = "org.jsp.springdi.autowiring.Engine">
    </bean>
    <bean id = "myCar" class = "org.jsp.springdi.autowiring.car"
                autowiring = "byName">
    </bean>
</beans>
```

TestAutowiring.java

```
public class TestAutowiring
{
public static void main(String[] args)
{
ApplicationContext context =
        new ClassPathXmlApplicationContext("autowiring.xml");
Car c = context.getBean("myCar", Car.class);
c.getEngine().start();
}
}
```

Example to understand Primary = "true" attribute :—

BankAccount.java

```
public interface BankAccount
{
    double getBalance();
}
```

Phonepe.java

```
public class Phonepe
{
  private BankAccount account;
  // getters & setters
}
```

SBI.java

```
public class SBI implements BankAccount
{
    public double getBalance()
    {
        return 25000;
    }
}
```

ICICI.java

```
public class ICICI implements BankAccount
{
    public double getBalance()
    {
        return 6500;
    }
}
```

TestPhonepe.java

```
public class TestPhonepe
{
  public static void main(String[] args)
  {
   ApplicationContext context =
  new ClassPathXmlApplicationContext("Phonepe.xml");
    Phonepe pe = context.getBean("phonepe", Phonepe.class);
     System.out.println(pe.getAccount().getBalance());
  }
}
```

Phonepe.xml

```
<beans>
 <bean id="phonepe" class="org.jsp.springdi.autowiring.Phonepe"
        autowire="byType"></bean>
 <bean id="Sbi" class="org.jsp.Springdi.autowiring.SBI"
        primary="true"></bean>
 <bean id="ICICI" class="org.jsp.springdi.autowiring.ICICI"
   </bean>
</beans>
```

Collection Injection :—

⇒ The process of Spring container injecting the collection to the spring bean by using either either setter or constructor is called as collection injection.

⇒ The collection injection by Spring XML can be achieved by using the elements like <List>, <set>, <map>, <props> etc.

⇒ Inside <property> element which are use to inject List, Set, Map, and Properties respectively.

<property> </property>

Example to understand Collection injection by using Spring XML.

Department.java

```
public class Department
{
  private Properties dept_details;
  private Set<Integer> emp_ids;
  private List<String> emp_names;
  private Map<Integer, String> emp_details;

  // getters and setters
}
```

dept.xml

```
<beans>
  <bean id = "dept" class = "org.jsp.springdi.autowiring.Department" >
    <!-- Injecting Set -->
    <property name = "emp_ids" >
      <Set>
        <value> 101 </value>
        <value> 201 </value>
        <value> 301 </value>
      </Set>
```

```
</property>
<!-- Injecting List -->
<property name = "emp_names">
    <List>
        <value> ABC </value>
        <value> PQR </value>
        <value> xyz </value>
    </List>
</property>

<!-- Injecting Map -->
<property name = "emp_details">
    <map>
        <entry key = "101" value = "ABC" />
        <entry key = "201" value = "PQR" />
        <entry key = "301" value = "xyz" />
    </map>
</property>

<!-- Injecting Properties -->
<property name = "dept_details">
  <props>
      <prop key = "department_name"> Development </prop>
      <prop key = "department_loc"> BTM Layout </prop>
      <prop key = "Numbers_of_employee"> Three </prop>
  </props>
</property>
</bean>
</beans>
```

Test.java

```
public class Test
{ public static void main (String[] args)
  {
    ApplicationContext context =
        new ClassPathXmlApplicationContext ("dept.xml");
    Department d = context.getBean("dept", Department.class);
    System.out.println (d.getDept_details());
    System.out.println (d.getEmp_details());
    System.out.println (d.getEmp_ids());
    System.out.println (d.getEmp_names());
  }
}
```

Spring Bean Life Cycle :⟶

⇒ The process of depicting the different stages of a Spring bean from instantiation to destruction is called as Spring bean life cycle.

⇒ Following are important stages of Spring bean life cycle

1. Spring bean class loading
2. Spring bean Instantiation
3. Spring bean Initialization
4. Spring bean destruction

⇒ We can track the Lyfe cycle of Spring bean by implementing ~~Initializati~~ InitializingBean and DisposableBean Interfaces

⇒ The InitializingBean have an ~~abstraction~~ method afterPropertiesSet() which can be implemented in Spring Bean class, it will be

execute after initialization of Spring bean.

⇒ The ~~Disposable bean~~ DisposableBean have an abstract method destroy() which we can be implemented in Spring Bean class, it will execute after the destruction of Spring Bean.

Example to understand spring bean life cycle by implementing InitializingBean and disposableBean.

Demo.java

```
public class Demo implements InitializingBean, DisposableBean
{
 Static {
      System.out.println ("Demo class has been loaded");
  }
public void afterPropertiesSet() throws Exception
{
  System.out.println ("Demo has been initialized");
}
public void destroy() throws Exception
{
  System.out.println ("Demo has been distroyed");
}
}
```

lifecycle.xml

```
<beans>
   <bean id = "demo" class = "org.JSP.autowiring.Demo" >
        Scop = "prototype" > </bean>
</beans>
```

TestLifeCycleByApplicationContext.java

```
public class TestLyfeCycleByApplicationContext
{ public static void main(String[] args)
  {
    ApplicationContext context =
          new ClassPathXmlApplicationContext("lifecycle.xml");
    System.out.println("context.getBean("demo", Demo.class));
    System.out.println("context.getBean("demo", Demo.class));
    System.out.println(context.getBean("demo", Demo.class));
    ((ClassPathXmlApplicationContext)context).close();
  }
}
```

LifeCycleByBeanFactory.java

```
public class LifeCycleByBeanFactory
{
public static void main(String[] args)
  {
    Resource r = new ~~xml~~ ClassPathResource("lifecycle.xml");
    BeanFactory factory = new XmlBeanFactory(r);
    System.out.println(factory.getBean("demo", ~~D~~Demo.class));
    System.out.println(factory.getBean("demo", Demo.class));
    System.out.println(factory.getBean("demo", Demo.class));
  }
}
```

Note :→

We can also track the Spring bean Lifecycle by using the attributes ~~init~~ init-method and destroy-method present in <bean> element or by using attributes defaut-init-method and defaut-destroy-method present in <beans> element.

~~Ex:~~

Example to understand spring bean Lyfecycle by using attribute of <bean> and <beans> method :—

Demo.java

```
public class Demo
{
  Static {
      System.out.println ("Demo class has been loaded");
    }
   public Demo() {
       System.out.println ("Demo is ~~initialized~~ instantiated.");
   }
  public void hii(){
      System.out.println ("Demo has been initialized");
  }
  public void byee(){
      System.out.println ("Demo has been destroyed");
    }
}
```

xml code for using default-init-method and default-destroy-method

```
<beans default-init-method="hii" default-destroy-method="byee">
  <bean id="demo" class="org.jsp.springdi.autowiring.Demo"
      Scop="Singleton">
  </bean>
</beans>
```

xml code for using init-method and destroy-method

```
<beans>
  <bean id="demo" class="org.jsp.springdi.autowiring.Demo"
      Scope="singleton" init-method="hii" destroy-method="bye"
    >
  </bean>
</beans>
```

@Component : ⟶

⇒ It is a class level annotation belongs to Spring framework.

⇒ This annotation is use to make a class as a component which has to be managed by the Spring Container.

⇒ The class which is annotated with @Component is eligible for class path scanning and that class is managed by Spring container.

⇒ Following are the alternatives which can be used in place of @Component.

(i) @Service (ii) @Repository (iii) @Configuration (iv) @ Controller

@ Service

⇒ It is a class level annotation belongs to Spring framework.
⇒ This annotation is indicates that the annotated class is a Service class which will have the methods which contains the business logic.
⇒ It is a specialization of @component, so the class annotated with @Service are also eligible for class path Scanning.

@ Repository :—

⇒ It is a class level annotation belongs to Spring framework.
⇒ This annotation is indicates that the annotated class have the methods to persist the data (Store the data) or access the data.
⇒ We can annotate the DAO classes with @Repository as those classes will have the persistence logic.
⇒ The classes which are annotated with @Repository are eligible for class-path Scanning as it is a specialization of @ component.

@ Controller :—

⇒ It is a class level annotation belongs to Spring framework.
⇒ This annotation indicates that the class have the handler methods which are used to handle the requests.
⇒ A class ~~with~~ which carries @ Controller is eligible for class-path Scanning as it is a specialization of @ component.

@configuration :—

⇒ It is a class level annotation belongs to Spring framework.

⇒ This annotation indicates that the class have the one or more @Bean method.

⇒ The class which is annotated with @configuration will be eligible for class-path scanning as it is a specialized annotation of @component

Example to understand @component annotation

Paytm.java

```
package org.spring.jsp.Springannotationpractice.Paytm;
@component
public class Paytm
{ public void open()
  {
    System.out.println("welcometo paytm");
  }
}
```

ApplicationContext.xml

```
<beans>
    <context: component-scan
              base-package="org.jsp.Springannotationpractice"/>
</beans>
```

OpenPaytm.java

```
public class OpenPaytm
{
  public static void main (String[] args)
  {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("ApplicationContext.xml");
    Paytm paytm = context.getBean("paytm", Paytm.class);
    paytm.open();
  }
}
```

AnnotationConfigApplicationContext :—

⇒ It is an implementation class of ApplicationContext.

⇒ It is stand alone container which will ~~makes~~ take @component as input.

⇒ We can register the Spring beans manually by using register(class<T>...) or by using scan(String...).

⇒ After calling scan() or register() we must refress the contains by calling refresh().

⇒ Following are important constructor present in AnnotationConfigApplicationContext.

(i) AnnotationConfigApplicationContext()

(ii) AnnotationConfigApplicationContext(class<T>... configClasses)

(ii) AnnotationConfigApplicationContext(String... basePackages)

Example to understand Spring bean Configuration using AnnotationConfigApplicationContext() constructor :-

Phonepe.java

```
@component
public class Phonepe
{
  public void open()
  {
    System.out.println("welcome to phonepe");
  }
}
```

OpenPhonepe.java

```
public class OpenPhonepe
{
public static void main(String[] args)
 {
 AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext();

// context.register(Phonepe.class);

 context.scan("org.jsp.springAnnotationPractice");
 context.refresh();
 Phonepe phonePe = context.getBean("phonePe", Phonepe.class);
 phonePe.open();
}
}
```

Example to understand Spring bean configuration using AnnotationConfigApplicationContext(String... basePackages) constructor :-

TestACAC.java

```
public class TestACAC
{
 public static void main (String[] args)
 {
  ApplicationContext context = new AnnotationConfigApplicationContext
        ("org.JSP.SpringAnnotationPractice");
  Phonepe phonePe = ("phonepe" , Phonepe.class);
  Paytm paytm = ("paytm" , Paytm.class);
  phonepe.open();
  paytm.open();
 }
}
```

@ComponentScan :-

⇒ It is a class level annotation belong to Spring Framework.

⇒ This annotation is use to specify the base packages in which the Spring container has to scan for components.

⇒ This annotation is alternative for spring XML's <context:component-scan/> element.

⇒ We can @ComponentScan ~~use~~ when we want to use @ComponentScan multiple times.

@ComponentScans :-

⇒ It is class level annotation belong to Spring Framework.

⇒ It is a container annotation which is used to aggregate multiple @ComponentScan annotations.

Note :- @ComponentScan and @ComponentScans can be used along with @Configuration

* Example for Spring Configuration using AnnotationConfigApplicationContext (Class<T> ... ~~Configuration~~ configClasses) constructor

myConfig.java

```
@Configuration
@ComponentScan(basePackages={"org.jsp.springannotationpractice"})
public class MyConfig
{
}
```

Test.java

```
public class Test
{ public static void main(String[] args)
  { ApplicationContext context = new
    AnnotationConfigApplicationContext(myConfig.class);
    Paytm paytm = context.getBean("paytm", Paytm.class);
    Phonepe phonepe = context.getBean("phonepe", Phonepe.class);
    paytm.open();
    phonepe.open();
  }
}
```

@Value :⟶

⇒ it is a annotation belong to spring framework.

⇒ This annotation is ~~assign~~ used to assign the default value for an ~~annotated~~ annotated field.

⇒ This annotation can be used on a field, method, and parameter.

@~~pro~~ PropertySource :—

⇒ It is a class level annotation belongs to spring framework.

⇒ This annotation is used to specify the resource from which the spring container has to load the properties.

⇒ we can initialize the springbean by using properties file by using placehalder (${propertyName}) in @value.

⇒ This annotation can be used along with Configuration.

@PropertySources :—

⇒ It is a class level annotation belong to spring framework

⇒ It is a container annotation which is used to aggregate multiple @PropertySource annotation.

Example to understand @propertySource @value :-

credential.java

```
@component
public class Credentials
{ @Value (value = "${jdbc.user.name}")
  private String userName;
  @Value (value = "${jdbc.user.password}")
  private String password;

  // getter & setters
  // toString
}
```

jdbc.properties

1. jdbc.user.name = scott
2. jdbc.user.password = admin

myconfig.java

```
@configuration
@componentScan (basePackages = {" org.jsp" })
@Properties
@PropertySource (value = {" jdbc.properties" })
public class myConfig
{

}
```

TestCredentials.java

```
public class TestCredentials
{
  public static void main(String[] args)
  {
    ApplicationContext context =
    new AnnotationConfigApplicationContext("myConfig.class");
    Credential credentials : context.getBean("credentials", Credentials.class);
    System.out.println(credentials);
  }
}
```

@Autowired

⇒ It is an annotation belong to Spring framework.

⇒ A member ~~which~~ which is annotated with @Autowired will be considered as a Autowiring member and Spring container will inject the resources to the annotation member.

⇒ This annotation can be used on a field, method, constructor as well as parameter.

⇒ By default the required attribute in this annotation is set to be true.

⇒ If required is set to true Spring container must inject the reference by finding the bean which can be qualified as autowire candidate else we will get UnsatisfiedDependencyExaption with a root cause NoSuchBeanDefinationException.

⇒ If required is set to false, Spring container will inject the dependency if it is present, else it will not inject the dependency and we will not get any Exception.

⇒ If there are more than one qualifying bean present for a single reference type, we will get UnsatisfiedDependyException with the root cause NoUniqueBeanDefinitionException.

Example to understand @Autowired.

Car.java

```
Package org.JSP;
@Component
public class Car
{
  @Autowired
   private Engine engine;
   ~~priva~~
   public car()
   {
   }
  public car(Engine engine)
  {
    this.engine = engine;
  }
 // getter & setters
}
```

Engine.java

```
Package org.JSP;
@Component
 public class Engine
 {
  public void start()
   {
     System.out.println("Engine Started");
   }
 }
```

TestCar.java

```
publi class TestCar
{
  public static void main (String[] args)
  {
    ApplicationContext context = new
        AnnotationConfigApplicationContext(myconfig.class);
    Car car = context.getBean("car" Car.java);
    car.getEngine().start();
  }
}
```

@Primary :—

⇒ It is an annotation belong to Spring framework.

⇒ This annotation can be used on a class as well as method.

⇒ This annotation is an alternative of Spring XML'S primary attribute which is present in <bean> element.

⇒ This annotation can be used along with @component or @Repository, @service, @configuration, @Controller and @Bean.

⇒ It is use to ~~mat~~ mark the bean as a primary bean that has to be give priority when there are multiple beans qualify as autowiring candidate for a single reference type.

@Qualifier :—

⇒ It is belong to Spring framework.

⇒ It is use to specify the qualifying bean for a reference type

⇒ It can be used along with @Autowire

Example program to understand @Primary & @Qualifier.

BankAccount.java

```
public interface BankAccount
{ public void displayBank();
}
```

SBI.java

```
@Component
public class SBI implements BankAccount
{ public void displayBank()
  { System.out.println("welcome to SBI");
    System.out.println("your Balance: "+25000);
  }
}
```

ICIC.java

```
@Primary
@Component
public class ICIC implements BankAccount
{ @Override
  public void displayBank()
  { System.out.println("welcome to ICIC");
    System.out.println("your Bank Balance: "+35000);
  }
}
```

GooglePay.java

```
@Component
public class GooglePay
{ @Autowired
  @Qualifier(value = "SBI")
  private BankAccount Account;
  // getter & setter
}
```

BankingTest.java

```
public class BankingTest
{
  public static void main(String[] args)
  {
    ApplicationContext context =
        new AnnotationConfigApplicationContext(myconfig.class);
    GooglePay g = context.getBean("googlePay", GooglePay.class);
    g.getAccount().displayBank();
  }
}
```

@Bean :—

⇒ It is a method level annotation belong to Spring framework.

⇒ This annotation indicates that the method returns an object which has to be considered as a Spring bean and it has to be managed by the Spring container.

⇒ This annotation can be used on the method present @Configuration classes.

Example program to understand @Bean

myConfig.java

```
package org.jsp;

@Configuration
@ComponentScan(basePackages = {"org.jsp"})
@PropertySource(value = {"jdbc.properties"})
public class myConfig
{
  @Bean
  @Primary
  Public List<String> getStudentNames(){
    return Arraylist
```

```
    return Arrays.asList("Virat", "Dhoni", "Rohit");
  }
  @Bean
  public List<String> getList()
  {
    return Arrays.asList("Ronaldo", "Messi", "Suriv");
  }
}
```

Batch.java

```
package org.JSP;
@component
public class Batch
{
  @Autowired
  @Qualifier("getList")
  private List<String> StudentNames;

  // getter & setter
}
```

TestBatch.java

```
public class TestBatch
{
  public static void main(String[] args)
  {
    ApplicationContext context = new
        AnnotationConfigApplicationContext(myConfig.class);
    Batch batch = context.getBean("batch", Batch.class);
    System.out.println(batch.getStudentNames());
  }
}
```

Spring ORM :⟶

⇒ It is a module of Spring Framework which facilitates the Object Relational Mapping.

⇒ This module simplify the interaction between java application and database servers, using this module we can avoid the basic steps of execution of hibernate, JDBC, JPA, etc.

⇒ We have HibernateTemplate using which we can avoid the basic steps of execution that are supposed to be written before and after performing a CRUD operation.

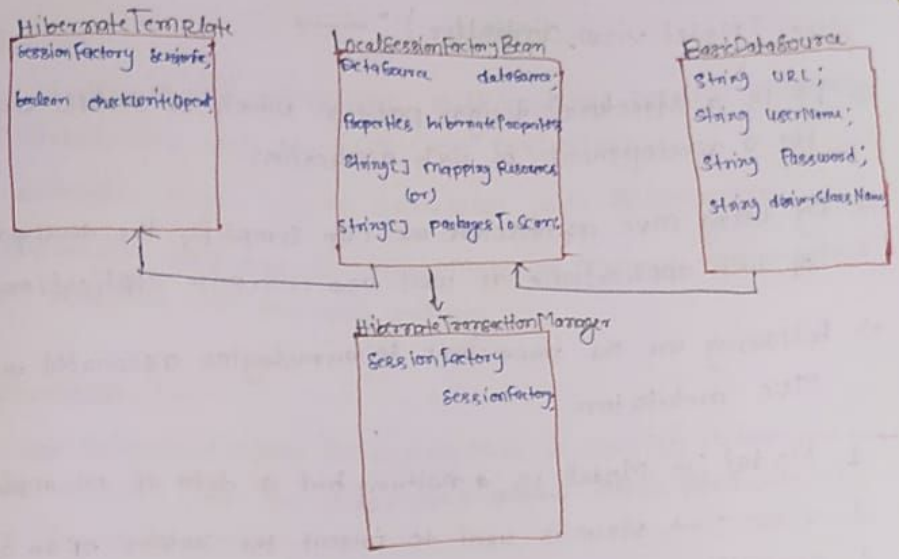
HibernateTemplate :⟶

⇒ It is a class present in Spring ORM module.

⇒ By using this class we can avoid the basic step of execution in hibernate that are supposed to be executed before and after performing a CRUD operation.

⇒ We need to configure the following Spring Beans in order to perform the CRUD operations using HibernateTemplate.

1. HibernateTemplate
2. LocalSessionFactoryBean
3. BasicDataSource
4. HibernateTransactionManager.



Steps to perform CRUD operation using HibernateTemplate

Step 1 : Create Simple maven Project.

Step 2 : Add the following dependencies in pom.xml

~~Step 3~~ : (i) Spring Context (5.3.20)
(ii) Spring ORM (5.3.20)
(iii) Commons-dbcp (1.4)
(iv) MySQL Connector Java (8.0.28)
(v) Hibernate Core Relocation (5.6.15 Final)

Step 3 : Create a Entity class and Map it with the table either by using Hibernate Mapping file or by using JPA annotation

Step 4 : Create a XML file and configure the following Spring Beans

(i) HibernateTemplate
(ii) LocalSessionFactoryBean
(iii) BasicDataSource
(iv) HibernateTransactionManager.

MVC (Model view controller) : ⟶

⇒ It is a structural design pattern which is exclusively used for a development of web application

⇒ By using mvc architecture we can simplify the development of web applications as well as enterprise application.

⇒ Following are the important terminologies associated with MVC architecture

1. Model :- Model is a nothing but a data of an application
2. View :⟶ View is used to present the content of an applicati
3. Controller :⟶ Controller is used to handle the web requests from the client.

Spring web MVC : ——

⇒ It is a model of spring framework which will simplify the development of web application by using MVC architecture.

⇒ With the help of spring MVC we can achieve loose coupling between view and controller.

⇒ Following are the important terminologies associated with Spring MVC

1. Model :—
   ⇒ It is an interface present in spring web MVC framework.
   ⇒ This interface is use to store the data of an application.
   ⇒

2 view :—
   ⇒ It is an interface belong to spring web MVC framework
   ⇒ This interface is use to represent the view of the application (it's used to present the content of an application)

3. Controller :—

⇒ A controller in ~~an~~ the Spring MVC will have the handler methods to handle the web requests from the client.

⇒ A controller class will be annotated with @Controller.

⇒ A controller will have @RequestMapping methods ~~with~~ which are mapping with the web request

4. FrontController :—

⇒ A ~~for~~ FrontController in Spring MVC is used to intercept the web requests from the client and ~~letter~~ later forwards it to controllers.

⇒ In Spring Web MVC DispatcharServlet class is considerd as FrontController and it will intercept the web requests from the client.

⇒ We must create configurate file which is used to provide the information to the frontController about the controllers and ViewResolver.

5. ModelAndView :—

⇒ It is a class defined in Spring MVC framework.

⇒ This class is a combination of model and view interface.

⇒ This class have methods using which we can set the view name also we can add the data to the application.

6. ViewResolver :—

⇒ It is an interface belongs to Spring web MVC.

⇒ This interface is used to achieve the loose coupling between ~~the~~ view and controller.

⇒ Following are the important attribute of ViewResolver

(1.) prefix (2) Suffix

Steps to Create basic Spring MVC Application : —

Step 1: Create a Maven project by selecting the following archetype

Group Id : org.apache.maven.archetypes

Artifact Id : Maven-archetype-webapp

Version : 1.4

If you are unable to find the above archetype, add a remote catalog to your IDE with the help of following url

URL— https://repo.maven.apache.org/maven2/archetype-catalog.xml

Step 2: Add a following dependencies in pom.xml

(i) Spring context (5.3.20)

(ii) Spring web MVC (5.3.20)

(iii) Servlet-api (4.0.1)

Step 3: Open a web.xml file and map the DispatcherServlet with web request.

```
<Servlet>
    <servlet-name> MvcPractice </servlet-name>
    <servlet-class> org.Springframework.web.servlet.
                    DispatcherServlet </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name> MvcPractice </servlet-name>
    <url-pattern> / </url-pattern>
</servlet-mapping>
```

step 4 :- Create an xml file whose name is same as ~~ServletName~~ ServletName-servlet.xml.

⇒ According to the above web.xml file the file name of configuration file must be MvcPractice-servlet.xml

If DispatcherServlet name is MvcDemo, then file name must be MvcDemo-servlet.xml.

if DispatcherServlet name is MyMvc, then the file name must be MyMvc-servlet.xml.

step 5 : Create a @RequestMapping method inside @Controller class and send the request

@RequestMapping :-

⇒ It is an annotation belongs to Spring Web MVC Framework.

⇒ This annotation is used to map the handler methods in the Controller class with the web request.

⇒ This annotation can be used on a Controller class as well as the methods in the Controller class.

@ ResponseBody :

⇒ It is an annotation belong to Spring Framework.

⇒ This annotation is used with @RequestMapping to indicate that the return value of the method is a response and it must be mapped with web response body.

⇒ This annotation can be used on the method as well as class.

Example to understand @RequestMapping and @ResponseBody

MvcPractice-servlet.xml

```
<bean>
    <context:component-scan base-package="org.jsp.springmvcpract
    />
    <context:annotation-config />
</bean>
```

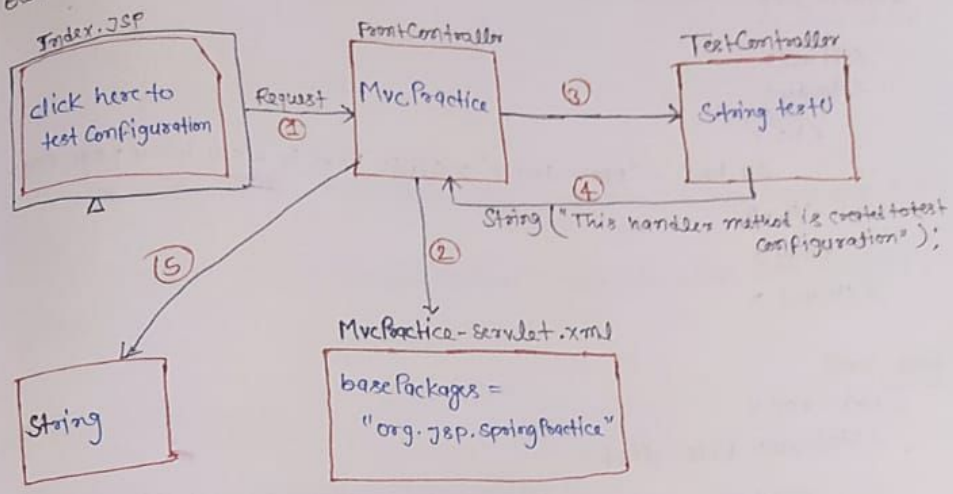
TestController.java

```
@Controller
public class TestController
{ @RequestMapping(value="/test")
  @ResponseBody
  public String Test()
  {
    return "<h1>This handler method is created to test the configuration</h1>";
  }
}
```

Index.jsp

```
<html>
<body>
    <h2>
        <a href="test">Click Here to test Configuration</a>
    </h2>
</body>
</html>
```

Example to order



Example to understand view Resolver :-

Home.JSP

```
<body>
    <h1> Welcome to the world of Spring Mvc </h1>
</body>
```

TestController.java

```
@RequestMapping(value = "/open-home")
public String OpenHome()
{
    return "Home";
}
```

Index.jsp

```
<Html>
<body>
  <h2>
    <a href = "open-home">Click here to view home page</a>
  </h2>
</body>
</Html>
```

Web.xml

```
<Web-app>
  <Welcome-file-list>
    <Welcome-file>/WEB-INF/view/index.jsp</Welcome-file>
  </Welcome-file-list>
</Web-app>
```

Mvcpractice-servlet.xml

```
<bean id = "viewResolver"
  class = "org.springframework.web.servlet.view.InternalResourceViewResolver
                                                    ">
    <property name = "prefix" value = "/WEB-INF/view/" />
    <property name = "suffix" value = ".jsp" />
</bean>
```

@ RequestParam :-

⇒ It is an annotation present in Spring web MVC.

⇒ This Annotation is used to bind the web request parameter with the method parameter present in request handling classes (controller class)

Example to Understand @RequestParam

Index.jsp

~~<h2>~~
~~<a href = "test">~~

```
<h2>
    <a href = "open-view?view=home"> click here to view home page </a>
</h2>
<h2><a href = "open-view?view=findSum"> click here to find the sum </a>
</h2>
```

TestController.java

```
@RequestMapping(value = "/open-view")
public String openView(@RequestParam(name = "view") String view)
{
    return view;
}
@RequestMapping("/sum")
public String findSum(@RequestParam(name="n1") int n1, @RequestParam(name="n2") int n2
                      ,Model model)
{
    String result = n1 + "+" + n2 + "=" + (n1 + n2);
    model.addAttribute("res", result);
    return "print";
}
```

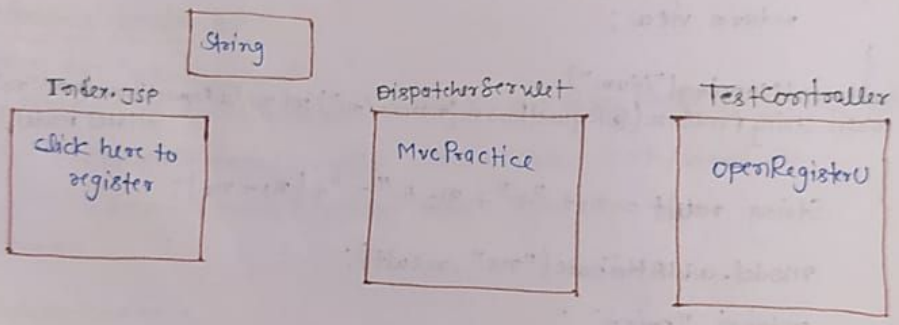
findSum.jsp

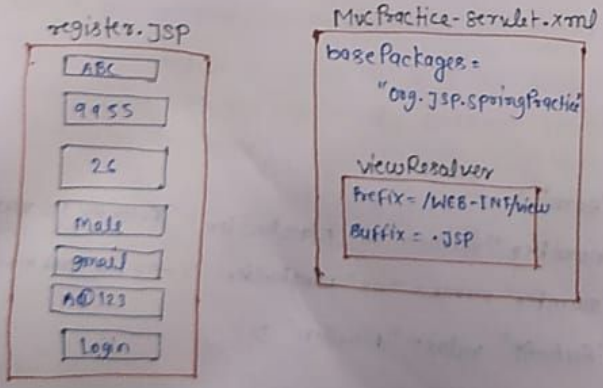
```
<html>
  <body>
    <form action = "sum">
      <input type="number" name="n1" placeholder = "Enter the 1st number">
      <input type = "number" name = "n2" placeholder = "Enter the 2nd number">
      <input type = "submit" value = "findSum">
    </form>
  </body>
</html>
```

point.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" isELIgnored="false" %>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <h1>${res}</h1>
</body>
</html>
```

isELIgnored="false" → write that line





@ModelAttribute :—

⇒ It is an annotation belongs to Spring Mvc Framework.

⇒ This annotation is used to map the named modal attribute with method parameter of handler method.

Example to understand @ModelAttribute

register.JSP

```
<body>
  <form:form modelAttribute="user" action="register" method="post">
    <form:label path="name">Name</form:lable>
    <form:input path="name" />
    <br>
    <form:lable path="phone">Phone Number</form:label>
    <form:input path="~~phone~~" "phone" />
    <br>
    <form:label path="age">Age</form:label>
    <form input path="age" />
    <br>
```

<form:label path="gender">Gender</form:label>
Male <form:radiobutton path="gender" value="Male" />
Female <form:radiobutton path="gender" value="female" />

```
    <form:label path="email">Email Id</form:label>
    <form:input path="email" />
    <br>
    <form:label path="password">Password</form:label>
    <form:password path="~~path~~password" />
    <form:button>Register</form:button>
    <br>
  </form:form>
</body>
```

UserRequest.java

```
public class UserRequest
{
    private String name;
    private long phone;
    private int age;
    private String gender;
    private String email;
    private String password;
    // getter & setter
    // toString
}
```

TestController.java

```
@RequestMapping("/open-register")
public ModelAndView openRegister()
{
  ModelAndView modelAndView = new ModelAndView();
  modelAndView.setViewName("register");
  modelAndView.addObject("user", new UserRequest());
  return modelAndView;
}
//@RequestMapping(method=RequestMethod.POST, value="/register")
@PostMapping("/register")
@ResponseBody
public String register(@ModelAttribute(name="user")
                       UserRequest request){
    return request.toString();
}
```

index.JSP

```
<h2>
  <a href="open-register">Click Here to Register</a>
</h2>
```

@PostMapping :—

⇒ It is a method level Annotation belongs to Spring web MVC.

⇒ This annotation is used to map Http POST requests into specific handler methods

⇒ Specifically it is a composed annotation ~~with~~ which acts as a shortcut for annotation @RequestMapping(method = RequestMethod.POST)

@GetMapping :—

⇒ It is a method level annotation belongs to ~~Sppri~~ Spring web mvc.

⇒ This annotation is used to map Http GET requests ~~into sp~~ into specific handler methods

⇒ Specially it is a composed annotation which acts as a shortcut for annotation @RequestMapping(method = RequestMethod.GET)