Spring Day2

Spring using Annotations:

Using annotations we can provide the metadata (extra information) to a class.

Annotations can be applied on the top of either:

1. class/interface: class-level annotation
2. variables: field-level/ variable level annotation
3. method: method-level annotations
4. constructor: constructor-level annotations

Framework software uses the annotations to get the information about the element or manipulate the elements using reflection API.

From spring 2.5 onwards we can configure Spring beans and their dependencies using annotations

Spring supplies various types of annotations in its all modules to simplify the spring application development.

Spring Core module annotations:

We can categorize the spring core module annotation in the following 3 categories:

1. **Stereotype annotations**
2. **Auto-wiring annotations**
3. **Miscellaneous annotations**

Stereotype annotations:

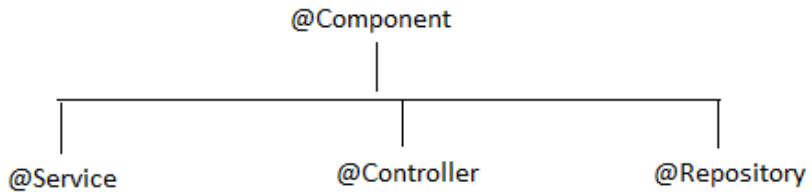
It is a special type of annotation that indicates the "**role of a class"** in a project.

These annotations are class-level annotations.

We have total 4 stereotype annotations:

1. @Component
2. @Service
3. @Repository
4. @Controller

Here @Component is the base/super annotation of the remaining 3.



@Service:

It indicates that this class is a service layer class which means, this class contains the main business logic.

@Repository:-

It indicates that this class is a Data access layer class, i.e. this class contains the data access logic/ persistence logic.

@Controller:-

It indicates that this class is a controller layer/presentation layer, which contains logic to control the flow of our application or contains the presentation logic.

Note: If we are not able to identify a class, from which layer it belongs then we can apply **@Component** annotation on the top of that class.

Component auto-scanning:

Spring framework provides an excellent feature called "**component auto-scanning**", in this feature spring container enters into a package (base package) and it will search for that package and all its sub-packages for a class, which is annotated with **stereotype annotation**, and pick those classes, create those object and register those objects as a "**spring bean**" with them(container).

With the component auto-scanning, the spring bean configuration burden is reduced on developers.

In **component auto-scanning** container will register a class as a spring bean with the class name (starting with a small letter) as its **id**.

Example:

```
@Repository
class AccountDao{

}
```

Here the above class will be registered as a Spring bean with the Spring container by the id “accountDao”.

In order to tell the container to register the above class with the different id then we use the below approach:

```
@Repository(value="dao")
class AccountDao{

}
```

Here the above class will be registered with "**dao"** id.

To enable the component auto-scanning, in the Spring configuration (applicationContext.xml) file we need to configure it by using the following tag:

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

The above configuration will auto-scan all the classes which are annotated with Stereotype annotations inside the following packages:

com.masai;
com.masai.dao;
com.masai.service;
com.masai.utiltity;

Example2 :

If we configure like :

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

com.masai // not scan
com.masai.dao //scan
com.masai.service //not scan
com.masai.utiltity //not scan

com.masai.dao.p1 // scan

Example:

```
package com.masai;
@Service
public class A{

}


package com.masai.services;
@Component
public class B{

}

package com.abc;
@Repository
```

```
public class C{


}

package com.masai.repo;
public class D{

}


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

Here only A and B classes will be registered with the spring container with the id "a" and "b".

Here C class is in a different package and the D class does not have stereotype annotation. so they
will not be registered with the container as a spring bean.

Example 2:

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

Now A, B, and C classes will be registered as a Spring bean with the container.

We Problem:

Let’s register a Java class with the Spring container using Annotations and make it a Spring bean.

```
A.java:-
--------

package com.masai;
```

```
import org.springframework.stereotype.Service;

@Service(value = "a1")
public class A {

  public void funA() {
    System.out.println("inside funA of A");
  }

}
```

applicationContext.xml:

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

<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd" >

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

</beans>
```

Demo.java:

```
package com.masai;

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

public class Demo {

  public static void main(String[] args) {


    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A obj =  ctx.getBean("a1",A.class);

    obj.funA();

  }
}
```

Auto-wiring annotations:

there are 2 auto-wiring annotations :

1. @Autowired
2. @Quilifier

1. @Autowired

- This annotation we can apply on the fields/variable or the setter method or on the constructor
  of a spring bean.
- The variable on which we apply the @Autowired annotation must be a reference type variable. not the primitive variable.

Example:

```
@Autowired   //valid
private Student student;

@Autowired    // invalid
private int x;
```

- If we apply @Autowired annotation on the top of the reference variable then no need to explicitly define the setter method. (setter injection point). at runtime container implicitly defines the setter method for that field.
- If we apply this @Autowire annotation on the reference field, then the container will search the
  "registered bean"(which spring bean is already registered with the container) by using either **byName** approach or by using **byType** approach, and if found then the container will inject that
  Spring bean object to this reference field.
- If not found any registered bean, then the container will throw an exception.

  Note:- if we use @Autowired annotation on the top of a reference field then that dependency will be mandatory.

  In order to make the dependency optional, we need to use

@Autowired(required = false)

- Here if the Spring bean object is there (registered with the container) then it will be injected otherwise the field will remain with the **null** value.

Example:

```
B.java:-
----------

package com.masai;

import org.springframework.stereotype.Service;

@Service
public class B {

  public void funB() {
    System.out.println("inside funB of B");

  }
}


A.java:-
-----------

package com.masai;

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

@Service
public class A {

  @Autowired
  private B b1;

  public void funA() {
    System.out.println("inside funA of A");
    System.out.println(b1);
  }

}
```

applicationContext.xml:

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

```
<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">


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

</beans>
```

Demo.java:

```
package com.masai;

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

public class Demo {

  public static void main(String[] args) {


    ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");

    A obj =  ctx.getBean("a",A.class);

    obj.funA();

  }
}
```

In the above application if we register one more object of B class as a spring bean using XML also with different id, then the Spring container will throw an ambiguity exception.

Example

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


<beans
xmlns="http://www.springframework.org/schema/beans"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
```

```
        https://www.springframework.org/schema/context/spring-context.xsd">


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

<bean id="b5" class="com.masai.B"/>

</beans>
```

Here to resolve the following dependency:

```
@Autowired
  private B b1;
```

Spring container will first use **byName** strategy to search the registered Spring bean, since it will not found then Spring container will use **byType** strategy to search the registered Spring bean, here it will found 2 spring bean one registered using stereotype annotation and another is registered with XML configuration, and Spring container would not qualify the appropriate bean to inject so it will raise an ambiguity error.

To solve the above problem we can make use **@Qualifier** annotation with the specific bean id which we want to inject.

Example:

```
@Autowired
@Qualifier("b5") // to choose one spring bean obj among multiple registered obj.
private B b1;
```

Note:- we can apply the @Autowired annotation on the top of the setter method also, using this we have more control over the our defined setter method compare to container defined setter method at runtime. i.e. we can place some other logics also inside our defined setter method.

Miscellaneous annotations:

1. @Scope:

Using this annotation we define scope of a spring bean.

Default scope for a spring bean is "**singleton**" scope.

Example:

```
@Service
@Scope("prototype")
public class A {
--
}
```

2. @PostConstruct and @PreDestroy:

It is similar to "**init-method**" and "**destroy-method**".

Both are the method level annotations.

In order to use these annotation we need to add following dependency inside the **pom.xml** file: **javax.annotation-api:**

Example:

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Example:

```
A.java :-
---------


package com.masai;

import javax.annotation.PostConstruct;

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

@Service
public class A {
```

```
  @Autowired
  private B b1;


  @PostConstruct
  public void setUp() {
    System.out.println("inside set up method");
  }

  @PreDestroy
  public void tearDown(){
    System.out.println("inside tear down method");
  }

  //main business logic
  public void funA() {
    System.out.println("inside funA of A");
    System.out.println(b1);
  }

}
```

3. @Configuration:

- From spring 4.x onward we can define a configuration class, instead of spring configuration file (XML file)
- To make a java class as a spring configuration class we need to annotate that class by using **@Configuration** annotation.
- It is a class level annotation.

Example:

```
@Configuration
public class AppConfig{

}
```

Note:- If we are using annotation based configuration class instead of the XML based configuration file then we need to use **"AnnotationConfigApplicationContext**" container instead of "**ClassPathXmlApplicationContext**" container.

Example:

```
incase of XML based configuration file:-

ApplicationContext ctx= new ClassPathXmlApplicationContext("applicationContext.xml");


incase of annotation-based configuration class:-

ApplicationContext ctx= new AnnotationConfigApplicationContext(AppConfig.class);
```

Note:- we can write more than one spring configuration class also.

Example:

```
@Configuration
public class AppConfig2{

}

in that case:-

ApplicationContext ctx= new AnnotationConfigApplicationContext(AppConfig.class,AppConfig2.class);
```

4.@ComponentScan:

- It is also a class level annotation
- This annotation will enable the component auto-scanning feature.

Note: this annotation should be used on the spring configuration class only, i.e. the class which is annotated with **@Configuration** annotation.

Example:

```
@Configuration
//@ComponentScan(basePackages = "com.masai")
@ComponentScan(basePackages ={ "com.masai" ,"com.abc" })
public class AppConfig{
```

```
}
```

I Problem:

Creating an Spring application which will make use of Annotation only(Without using XML file)

```
A.java:-
---------

package com.masai;

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

@Service
public class A {

  @Autowired
  private B b1;


  public void funA() {
    System.out.println("inside funA of A");
    System.out.println(b1);
  }

}

B.java:-
---------

package com.masai;

import org.springframework.stereotype.Service;

@Service
public class B {

  public void funB() {
    System.out.println("inside funB of B");

  }

}


AppConfig.java:-
```

```
-------------------

package com.masai;

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

@Configuration
@ComponentScan(basePackages = "com.masai")
public class AppConfig {

}



Demo.java:-
---------------

package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Demo {

  public static void main(String[] args) {

    ApplicationContext ctx= new AnnotationConfigApplicationContext(AppConfig.class);

    A obj =  ctx.getBean("a",A.class);

    obj.funA();


  }
}
```

You Problem:

Modify the above application to place @PostConstruct, @Scope as prototype annotations and call the method of A class object.

5. @Bean:

- We have 3 ways to register a java class with the spring container, to make a spring bean:

1. by using <bean> tag.

2. by using Stereotype annotation
3. by using @Bean annotation

- It is a method level annotation.
- On which method, we apply this annotation should return any object, spring container will register the returned object with itself(with container) with the id as method name itself.
- We can define @Bean annotated method either inside @Configuration annotated class or inside any spring bean class(which is annotated with stereotype annotation and scanned by the spring container).
- While auto-scanning, when container encounters with @Bean annotated method, it will call that method and register the returned object with the spring container by the **id** as the method name itself.

Example:

```
@Configuration
@ComponentScan(basePackages = "com.masai")
public class AppConfig {

@Bean
public  B getB(){

B b1=new B();
return b1;

}
```

Here the getB() method will be called automatically at the time of scanning and returned object of that method will be registered with the container with the id "**getB**".

If we want to change the id name as "**b1**" then:

```
@Bean("b1")
public  B getB(){

B b1=new B();
return b1;

}
```

With the help of @Bean annotation we can register any predefined classes also with the container.

Example:

```
@Bean("clist")
public List<String>  getCities(){

    List<String> cities = new ArrayList<String>();

    cities.add("delhi");
    cities.add("chennai");
    cities.add("mumbai");
    cities.add("kolkata");

    return cities;

}
```

A.java:

```
package com.masai;

import java.util.List;

import javax.annotation.PostConstruct;

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

@Service
public class A {

  @Autowired
  private List<String> cities;

  public void funA() {
    System.out.println("inside funA of A");
    System.out.println(cities);

  }

}
```

6. @Value:

This annotation is used to inject simple values type variables.

Example:

```
A.java:-
---------

package com.masai;

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

@Service
public class A {

  @Value("100")
  int roll=100;

  @Value("Ram")
  String name;

  @Value("780")
  int marks;


  public void funA() {
    System.out.println("inside funA of A");
    System.out.println("Roll is "+roll);
    System.out.println("Name is "+name);
    System.out.println("Marks is "+marks);
  }

}
```

Note:- this @Value annotation is not used to supply dummy data as above program.

It is used to supply the dynamic value from the **properties files**.

properties file: It is a textual file with the (dot).properties extension.

This file contains the entries in the form of key-value pair.

Example:

Step 1: create **a1.properties** file inside **src/main/resources** folder in maven application

```
db.driverName=com.mysql.cj.jdbc.Driver
db.url=jdbc:mysql://localhost:3306/ratandb
db.username=root
db.password=root
```

Step 2: Now we can inject these values to the simple type variable of our Bean.

Example:

```
A.java:-
-----------

package com.masai;

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

@Service
public class A {

  @Value("${db.driverName}")
  private String dname;

  @Value("${db.url}")
  private String url;

  @Value("${db.username}")
  private String uname;

  @Value("${db.password}")
  private String pass;



  public void funA() {
    System.out.println("inside funA of A");
    System.out.println("Driver name  is "+dname);
    System.out.println("Connection URL  is "+url);
    System.out.println("Username  is "+uname);
    System.out.println("Password  is "+pass);

  }

}
```

Step 3: specify the name of the properties file using **@PropertySource** annotation on the top of **@Configuration** annotated class.

Example:

```
AppConfig.java:-
-------------------

package com.masai;

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

@Configuration
@ComponentScan(basePackages = "com.masai")
@PropertySource("a1.properties")
public class AppConfig {

}
```

If we have multiple properties files:

```
@Configuration
@ComponentScan(basePackages = "com.masai")
@PropertySources({
  @PropertySource("a1.properties"),
  @PropertySource("a2.properties")
})
public class AppConfig {

}
```

Note:- we can read the properties files details by using "**Environment"** object also, in much more easier way.

Example:

```
A.java:-
----------

package com.masai;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Service;

@Service
public class A {
```

```
  @Autowired
  private Environment env;


  public void funA() {
    System.out.println("inside funA of A");
    System.out.println("Driver name  is "+env.getProperty("db.driverName"));
    System.out.println("Connection URL  is "+env.getProperty("db.url"));
    System.out.println("Username  is "+env.getProperty("db.username"));
    System.out.println("Password  is "+env.getProperty("db.password"));

  }

}
```