Spring Boot Architecture

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum efforts. It is developed on top of the core Spring Framework.

Spring Boot follows a layered architecture in which each layer communicates with the layer directly below or above (hierarchical structure) it.

Before understanding the **Spring Boot Architecture**, we must know the different layers and classes present in it. There are **four** layers in Spring Boot are as follows:

* **Presentation Layer**
* **Business Layer**
* **Persistence Layer**
* **Database Layer**



**Presentation Layer:** The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer. In short, it consists of **views** i.e., frontend part.

**Business Layer:** The business layer handles all the **business logic**. It consists of service classes and uses services provided by data access layers. It also performs **authorization** and **validation**.

**Persistence Layer:** The persistence layer contains all the **storage logic** and translates business objects from and to database rows.

**Database Layer:** In the database layer, **CRUD** (create, retrieve, update, delete) operations are performed.

Spring Boot Flow Architecture



* Now we have validator classes, view classes, and utility classes.
* Spring Boot uses all the modules of Spring-like Spring MVC, Spring Data, etc. The architecture of Spring Boot is the same as the architecture of Spring MVC, except one thing: there is no need for **DAO** and **DAOImpl** classes in Spring boot.
* Creates a data access layer and performs CRUD operation.
* The client makes the HTTP requests (PUT or GET).
* The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.
* In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.
* A JSP page is returned to the user if no error occurred.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

What is Dependency Injection:

Dependency Injection is the main functionality provided by Spring IOC(Inversion of Control). The Spring-Core module is responsible for injecting dependencies through either Constructor or Setter methods. The design principle of Inversion of Control emphasizes keeping the Java classes independent of each other and the container frees them from object creation and maintenance. These classes, managed by Spring, must adhere to the standard definition of Java-Bean. Dependency Injection in Spring also ensures loose-coupling between the classes.

Need for Dependency Injection:

Suppose class One needs the object of class Two to instantiate or operate a method, then class One is said to be dependent on class Two. Now though it might appear okay to depend a module on the other but, in the real world, this could lead to a lot of problems, including system failure. Hence such dependencies need to be avoided.

Spring IOC resolves such dependencies with Dependency Injection, which makes the code easier to test and reuse. Loose coupling between classes can be possible by defining interfaces for common functionality and the injector will instantiate the objects of required implementation. The task of instantiating objects is done by the container according to the configurations specified by the developer.

Types of Spring Dependency Injection:

There are two types of Spring Dependency Injection. They are:

1. Setter Dependency Injection (SDI): This is the simpler of the two DI methods. In this, the DI will be injected with the help of setter and/or getter methods. Now to set the DI as SDI in the bean, it is done through the bean-configuration file For this, the property to be set with the SDI is declared under the <property> tag in the bean-config file.

2. Constructor Dependency Injection (CDI): In this, the DI will be injected with the help of constructors. Now to set the DI as CDI in bean, it is done through the bean-configuration file For this, the property to be set with the CDI is declared under the <constructor-arg> tag in the bean-config file.

Example: Let us take the same example as of SDI

• Java

public class EXPL {

IExpl expl;

// Constructor to set the CDI

Expl(IExpl expl)

{

this.expl = expl;

}

}

1. Setting the CDI in the bean-config file:

• XML

<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-2.5.xsd">

<bean id="EXPL" class="com.expl.org. EXPL ">

<constructor-arg>

<bean class="com.expl.org.impl.CsvEXPL " />

</constructor-arg>

</bean>

<bean id="CsvEXPL " class="com.expl.org.impl.CsvEXPL " />

<bean id="JsonEXPL " class="com.expl.org.impl.JsonEXPL " />

</beans>

1. This injects the ‘CsvEXPL’ bean into the ‘EXPL object with the help of a constructor.

Setter Dependency Injection (SDI) vs. Constructor Dependency Injection (CDI)

class Class1{

| Setter DI | Constructor DI |
| --- | --- |
| Poor readability as it adds a lot of boiler plate codes in the application. | Good readability as it is separately present in the code. |
| The bean must include getter and setter methods for the properties. | The bean class must declare a matching constructor with arguments. Otherwise, BeanCreationException will be thrown. |
| Requires addition of @Autowired annotation, above the setter in the code and hence, it increases the coupling between the class and the DI container. | Best in the case of loose coupling with the DI container as it is not even required to add @Autowired annotation in the code.(Implicit constructor injections for single constructor scenarios after spring 4.0) |
| Circular dependencies or partial dependencies result with Setter DI because object creation happens before the injections. | No scope for circular or partial dependency because dependencies are resolved before object creation itself. |
| Preferred option when properties are less and mutable objects can be created. | Preferred option when properties on the bean are more and immutable objects (eg: financial processes) are important for application. |

Example of Spring DI:

• We have used three classes and an interface as beans to exemplify the concepts of CDI and SDI. They are Vehicle, ToyotaEngine, Tyres classes and IEngine interface respectively.

• From our example, we can see that class Vehicle depends on the implementation of the Engine, which is an interface. (So, basically, a Vehicle manufacturer wants a standard Engine which complies to Indian emission norms.) Class ToyotaEngine implements the interface and its reference is provided in the bean-configuration file mapped to one of Vehicle class’s properties.

• In the Vehicle class, we invoke the application context and bean instantiation is executed. Two objects of class Vehicle are instantiated. ‘obj1’ is instantiated via bean with name InjectwithConstructor. The bean name could be located in the bean configuration file. Similarly ‘obj2’ is instantiated via bean with name InjectwithSetter.

• It can be observed that ‘obj1’ is injected via the constructor and ‘obj2’ uses setter injection.

• In the bean configuration file below, we have used two Vehicle beans’ declarations.

• InjectwithConstructor bean makes use of element constructor-arg, with attributes name and ref. ‘Name’ attribute correlates with the constructor argument name given in the Vehicle class definition. And ‘ref’ attribute points to the bean reference which can be used for injecting.

• InjectwithSetter makes use of property element to provide the ‘name’ of the property and the ‘value’ for the property. In place of value attribute ‘ref’ can be used to denote a reference to a bean.

• In the configuration details, we are injecting ToyotaBean reference into the IEngine reference in Vehicle class constructor-arg, where IEngine is an interface and needs an implementing class reference for bean injection.

• We have used two separate bean references for Tyres class, to inject via setter and constructor respectively. We can observe that ‘tyre1Bean’ and ‘tyre2Bean’ are initialized with String literal values for each of the properties.

pom.xml

<dependencies>

<!-- https:// mvnrepository.com/artifact

/org.springframework/spring-core -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>4.3.11.RELEASE</version>

</dependency>

<!-- https:// mvnrepository.com/artifact

/org.springframework/spring-context -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>4.3.11.RELEASE</version>

</dependency>

</dependencies>

Enigne.java

interface IEngine {

String EMISSION\_NORMS = "BSIV";

String importOrigin();

double cost();

}

ToyotaEngine.java

public class ToyotaEngine implements IEngine {

String company;

double cost;

public double getCost()

{

return cost;

}

public void setCost(double cost)

{

cost = this.cost;

}

public String getCompany()

{

return company;

}

public void setCompany(String company)

{

this.company = company;

}

@Override

public String importOrigin()

{

return "Japan";

}

@Override

public double cost()

{

return cost;

}

@Override

public String toString()

{

return "This is Engine object from: "

+ company;

}

}

Tyres.java

public class Tyres {

String name;

String place;

String message;

public String getName()

{

return name;

}

public void setName(String name)

{

this.name = name;

}

public String getPlace()

{

return place;

}

public void setPlace(String place)

{

this.place = place;

}

public String getMessage()

{

return message;

}

public void setMessage(String message)

{

this.message = message;

}

@Override

public String toString()

{

return "This is Tyre object: "

+ name + " " + place

+ " " + message;

}

}

Vehicle.java

public class Vehicle {

IEngine engine;

Tyres tyre;

public Tyres getTyre()

{

return tyre;

}

public void setTyre(Tyres tyre)

{

System.out.println("tyre instantiated via setter");

this.tyre = tyre;

}

public Vehicle(IEngine engine, Tyres tyre)

{

System.out.println("instantiated via constructor");

this.engine = engine;

this.tyre = tyre;

}

public Vehicle() {}

public IEngine getEngine()

{

return engine;

}

public void setEngine(IEngine engine)

{

System.out.println("instantiated via setter");

this.engine = engine;

}

@Override

public String toString()

{

return engine + " " + tyre;

}

public static void main(String a[])

{

ApplicationContext rootctx

= new ClassPathXmlApplicationContext(

"springContext.xml");

// Instantiating the obj1 via Constructor DI

Vehicle obj1= (Vehicle)

rootctx.getBean("InjectwithConstructor");

// Instantiating the obj1 via Setter DI

Vehicle obj2= (Vehicle)

rootctx .getBean("InjectwithSetter");

System.out.println(obj1);

System.out.println(obj2);

System.out.println(obj1 == obj2);

}

}

springContext.xml

< bean id="tyre1Bean" class="com.techgene.Tyres">

<property name="name" value="MRF">

</ property>

<property name="place" value="India">

</ property>

<property name="message" value="Make in India">

</ property>

</ bean>

< bean id="ToyotaBean" class="com.techgene.ToyotaEngine">

<property name="company" value="Toyota">

</ property>

<property name="cost" value="300000.00">

</ property>

</ bean>

< bean id="tyre2Bean" class="com.techgene.Tyres">

<property name="name" value="TVS">

</ property>

<property name="place" value="India">

</ property>

<property name="message" value="Make in India">

</ property>

</ bean>

< bean id="InjectwithSetter" class="com.techgene.Vehicle">

<property name="engine" ref="ToyotaBean">

</ property>

<property name="tyre" ref="tyre1Bean">

</ property>

</ bean>

< bean id="InjectwithConstructor" class="com.techgene.Vehicle">

<constructor - arg name="engine" ref="ToyotaBean">

</ constructor - arg>

<constructor - arg name="tyre" ref="tyre2Bean">

</ constructor - arg>

</ bean>

Process Flow: The process flow of bean instantiation and injection of dependencies is given in the picture below:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*End\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Autowiring in Spring

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Advantage of Autowiring

It requires the **less code** because we don't need to write the code to inject the dependency explicitly.

Disadvantage of Autowiring

No control of programmer.

It can't be used for primitive and string values.

Autowiring Modes

There are many autowiring modes:

|  |  |  |
| --- | --- | --- |
| **No.** | **Mode** | **Description** |
| 1) | No | It is the default autowiring mode. It means no autowiring bydefault. |
| 2) | byName | The byName mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method. |
| 3) | byType | The byType mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method. |
| 4) | constructor | The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters. |
| 5) | autodetect | It is deprecated since Spring 3. |

Example of Autowiring

Let's see the simple code to use autowiring in spring. You need to use autowire attribute of bean element to apply the autowire modes.

1. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

Let's see the full example of autowiring in spring. To create this example, we have created 4 files.

1. **B.java**
2. **A.java**
3. **applicationContext.xml**
4. **Test.java**

**B.java**

This class contains a constructor and method only.

1. **package** org.sssit;
2. **public** **class** B {
3. B(){System.out.println("b is created");}
4. **void** print(){System.out.println("hello b");}
5. }

**A.java**

This class contains reference of B class and constructor and method.

1. **package** org.sssit;
2. **public** **class** A {
3. B b;
4. A(){System.out.println("a is created");}
5. **public** B getB() {
6. **return** b;
7. }
8. **public** **void** setB(B b) {
9. **this**.b = b;
10. }
11. **void** print(){System.out.println("hello a");}
12. **void** display(){
13. print();
14. b.print();
15. }
16. }

**applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="b" **class**="org.sssit.B"></bean>
10. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>
12. </beans>

**Test.java**

This class gets the bean from the applicationContext.xml file and calls the display method.

1. **package** org.sssit;
2. **import** org.springframework.context.ApplicationContext;
3. **import** org.springframework.context.support.ClassPathXmlApplicationContext;
4. **public** **class** Test {
5. **public** **static** **void** main(String[] args) {
6. ApplicationContext context=**new** ClassPathXmlApplicationContext("applicationContext.xml");
7. A a=context.getBean("a",A.**class**);
8. a.display();
9. }
10. }

Output:

b is created

a is created

hello a

hello b

1) byName autowiring mode

In case of byName autowiring mode, bean id and reference name must be same.

It internally uses setter injection.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

But, if you change the name of bean, it will not inject the dependency.

Let's see the code where we are changing the name of the bean from b to b1.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

2) byType autowiring mode

In case of byType autowiring mode, bean id and reference name may be different. But there must be only one bean of a type.

It internally uses setter injection.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="byType"></bean>

In this case, it works fine because you have created an instance of B type. It doesn't matter that you have different bean name than reference name.

But, if you have multiple bean of one type, it will not work and throw exception.

Let's see the code where are many bean of type B.

1. <bean id="b1" **class**="org.sssit.B"></bean>
2. <bean id="b2" **class**="org.sssit.B"></bean>
3. <bean id="a" **class**="org.sssit.A" autowire="byName"></bean>

In such case, it will throw exception.

3) constructor autowiring mode

In case of constructor autowiring mode, spring container injects the dependency by highest parameterized constructor.

If you have 3 constructors in a class, zero-arg, one-arg and two-arg then injection will be performed by calling the two-arg constructor.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="constructor"></bean>

4) no autowiring mode

In case of no autowiring mode, spring container doesn't inject the dependency by autowiring.

1. <bean id="b" **class**="org.sssit.B"></bean>
2. <bean id="a" **class**="org.sssit.A" autowire="no"></bean>