**SPRINGBOOT**

1. **Define IOC (Inversion of Control).**

IOC is a **principle** by which control of objects is transferred to a container or framework. DI is a **pattern** through which IOC is implemented, where the control being inverted is nothing but setting object’s dependencies.

Injection is at the **micro** level and Inversion of Control is at the **macro** level. You have to eat every bite (implement DI) in order to finish a meal (implement IoC).

The Inversion-of-Control**(IoC)** pattern, is about providing *any kind* of callback, instead of acting ourselves directly (in other words, inversion and redirecting control to the external handler/controller).

A class should not be configured by itself but from outside done by container.

IOC makes application loosely controlled, easy to test.

In layman language, you hire a cab everyday you should give instruction to reach office which can be pretty annoying, it won’t be the case when cab is provided by the company as they know what to do.

So if you follow these simple two steps, you have done inversion of control:

1. Separate **what**-to-do part from **when**-to-do part.
2. Ensure that **when** part knows as *little* as possible about **what** part; and vice versa.

1. **Define DI (Dependency Injection)?**

The Dependency-Injection**(DI)** is a software design pattern that deals with how component gets hold of their dependencies and is all about removing dependencies from your code.

For DI example, say your application has a text-editor component, and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor {

private SpellChecker checker;

public TextEditor() {

this.checker = new SpellChecker();

}

}

What we've done here creates a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor {

private IocSpellChecker checker;

public TextEditor(IocSpellChecker checker) {

this.checker = checker;

}

}

In the first code example we are instantiating SpellChecker (this.checker = new SpellChecker();), which means the TextEditor class directly depends on the SpellChecker class.

In the second code example we are creating an abstraction by having the SpellChecker dependency class in TextEditor's constructor signature (not initializing dependency in class). This allows us to call the dependency then pass it to the TextEditor class like so:

SpellChecker sc = new SpellChecker(); // dependency

TextEditor textEditor = new TextEditor(sc);

Now the client creating the TextEditor class has control over which SpellChecker implementation to use because we're injecting the dependency into the TextEditor signature.

The original dependency graph looks like (main -> TextEditor -> SpellChecker), but in the second code snippet, it looks like (main -> TextEditor, main -> SpellChecker). The dependency from TextEditor to SpellChecker has been broken, which is useful.

SpellChecker is an interface, not a defined class. Therefore any class implementing that interface can be 'injected' into TextEditor. Whoever creates TextEditor can determine what implementation of ISpellChecker it uses. And this answer describes Dependency Injection

You have a **laptop** computer and you accidentally break the screen. And darn, you find the same model laptop screen is nowhere in the market. So you're stuck.

You have a **desktop** computer and you accidentally break the screen. You find you can just grab almost any desktop monitor from the market, and it works well with your desktop.

<https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>

1. **How does Springboot works internally?**

Spring does not generate any code automatically and not using any xml configuration file. so spring uses internally pragmatically configuration done by spring boot developer that are provided by jar. we are using just pre-configured jar and those jar available in META\_INF/spring.factories

You can enable preconfigured jars by adding dependency in pom.xml

When Run method is called,

1. create application context

2. check Application Type

3. Register the annotated class beans with the context

4. Creates an instance of TomcatEmbeddedServletContainer and adds the context. Used to deploy our jar automatically.

<https://gainjavaknowledge.medium.com/how-spring-boot-application-works-internally-dd9bd3ecc487>

1. What is BeanFactory?

In spring framework IOC container is responsible to inject dependencies, we provide metadata to the IOC container either by XML file or annotation.

IOC container is responsible to instantiate, configure and assemble the objects

Two of the IOC containers used in spring framework are Beanfactory and applicationContext.

BeanFactory is the simplest container that provide basic support for DI.

It is defined by org.springframework.benas.factory.BeanFactory interface

There are many implementation, one being XmlBeanFactory

Further Resource has many implementations.

Ex: Resource resource = new ClassPathResource(“Beans.xml”);

BeanFactory factory = new XmlBeanFactory(resource);

Student student = factory.getBean(“StudentBean”);

student.getDetails();

https://www.baeldung.com/spring-beanfactory

1. **What is application context and what are its different types?**

Application context is an another IOC container in spring framework .

Some of the function of application context are:

Loading of configuration files

Triggering a classpath scan

Programmatically registering bean definitions and annotated classes Registering functional bean definitions

Ex:

ApplicationContext context = new ClassPathXmlContext(“a.xml”);

Student student = context.getBean(“studentBean”);

Types of application context:

<https://www.geeksforgeeks.org/spring-applicationcontext/>

<https://zetcode.com/springboot/applicationcontext/#:~:text=ApplicationContext%20is%20a%20corner%20stone,assemble%20by%20reading%20configuration%20metadata>.

<https://docs.spring.io/spring-framework/reference/core/beans/basics.html>

1. **What is the difference between BeanFactory and application context, Can you use BeanFactory instead of application context?**

Because an ApplicationContext includes all the functionality of a BeanFactory, it is generally recommended over a plain BeanFactory, except for scenarios where full control over bean processing is needed.

Within an ApplicationContext several kinds of beans are detected by convention (that is, by bean name or by bean type — in particular, post-processors), while a plain DefaultListableBeanFactory is agnostic about any special beans.

It is possible to create more than one Application Context for a single server and the circumstance is to break down a huge application into separate containers for better modularity aka loose coupling.

But all the application contexts are registered in one parent application context which the spring looks for on app startup.

<https://docs.spring.io/spring-framework/reference/core/beans/beanfactory.html>

1. **What are the different Scopes of a Bean?**
2. Singleton
3. Prototype
4. Session
5. Request
6. Application
7. Websocket

* Other than using default custom scopes can be created
* Singleton is the default scope
* Spring’s concept of a singleton bean differs from the singleton pattern as defined in the Gang of Four (GoF) patterns book.The scope of the Spring singleton is best described as being per-container and per-bean
* DAO object does not hold any state, it can be singleton scope
* Any object that holds state is created as prototype
* Singleton: All requests for that bean name will return the same object, which is cached. Any modifications to the object will be reflected in all references to the bean. one instance per bean definition

say 4 employee bean,2 each refers to 2 department bean IT and HR.. so single bean referred by IT is returned everytime

* Prototype;A bean with prototype scope will return a different instance every time it is requested from the container request,session,application,websocket.

<https://docs.spring.io/spring-framework/reference/core/beans/factory-scopes.html>

<https://www.baeldung.com/spring-inject-prototype-bean-into-singleton>

1. **What are the ways of instantiating a Bean?**

By Class property using Class Name and Nested class name

By Constructor

By static factory method

Instantiation by Using an Instance Factory Method

(GO THROUGH THIS DOCUMENT COMPLETELY)

<https://docs.spring.io/spring-framework/reference/core/beans/definition.html#beans-factory-class>

1. **What is Lazy initialization of Bean?**

<https://springhow.com/lazy-initialization-in-spring-boot/>

1. **Life cycle of a Bean**

<https://howtodoinjava.com/spring-core/spring-bean-life-cycle/>

1. **Ways of creating a bean**

<https://www.geeksforgeeks.org/how-to-create-a-spring-bean-in-3-different-ways/>

<https://www.linkedin.com/pulse/what-bean-how-many-ways-we-can-create-spring-abid-anjum>

1. **Types of Dependency Injection**

<https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-collaborators.html>

1. **Autowiring**

<https://docs.spring.io/spring-framework/reference/core/beans/dependencies/factory-autowire.html>

<https://docs.spring.io/spring-framework/reference/core/beans/annotation-config/autowired.html>

1. **JSRX annotations**

<https://docs.spring.io/spring-framework/reference/core/beans/standard-annotations.html>

1. **Transactional management in spring**

<https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth>

1. **How exception handling done in spirngboot**

<https://bushansirgur.in/spring-mvc-exceptionhandler-annotation-with-example/>

1. **What is AOP in Springboot?**
2. **Reflection utils in Springboot?**

<https://www.baeldung.com/spring-reflection-test-utils>

1. **Different annotations of Springboot?**

<https://javatechonline.com/spring-boot-annotations-with-examples/>

@Requuestbody--> Spring will bind the incoming HTTP request body(for the URL mentioned in @RequestMapping for that method) to that parameter

While doing that, Spring will [behind the scenes] use HTTP Message converters to convert the HTTP request body into domain object [deserialize request body to domain object], based on Accept header present in request.

method(@requestbody User user){ obj.createuser(user);}

@ResponseBody-->If a method is annotated with @ResponseBody, Spring will bind the return value to outgoing HTTP response body. While doing that, Spring will [behind the scenes] use HTTP Message converters to convert the return value to HTTP response body [serialize the object to response body], based on Content-Type present in request HTTP header

@Pathvariable-->when sending the value in url

1. **Spring Complete Tutorial**

<https://www.geeksforgeeks.org/spring/?ref=lbp>

1. **How do you connect to two databases?**
2. **What is springboot Actuator? What are the endpoints?**

* /Bean actuator endpoint provide list of all the beans

1. **What is HateOAS?**
2. **How to disable web server in springboot ?**

Spring.main.web-application-type = none

1. **Use of @Configuration,@AutoConfiguration and @ComponentScan of @SpringbootApplication.**

<https://www.digitalocean.com/community/tutorials/spring-configuration-annotation>

1. **Advantages of springboot**

* Standalone project with minimum configuration
* Embedded servers such as tomcat,jetty
* Production ready features like health check , metrics
* No requirement of XML configuration
* Security,logging,caching,validation

1. **Four spring boot components**

Four spring boot components

* Auto-configuration
* CLI
* Starter POM
* Actuators