**SPRING:**

-Framework focusing on creating enterprise applications using JAVA.

-Inversion of control and transactions.

**SPRING BOOT:**

-Auto configuration.

[Custom configuration using @Annotations]

-XML config.

-Standalone APIs.

**IOC [inversion of control]:**

* “Flow of application” is handed over to “spring framework” and not to the programmer.

**Dependency injection:**

-Spring framework can instantiate object for us. [**Not Everytime**]

-Actually, Spring container does it.

**Spring beans: object**

-Instance of a class managed by **Spring container.**

**IOC container:**

**-Managing all beans. [Creating,Deleting,Intializing INSTANCEs].**

**-Then, perform dependency injections.**

**Dependency Injections:**

1. Constructor based DI.

2. Setter based DI.

**Adding dependency to the project? Go to MVN Repository on the internet for dependency.**

**-**Open **pom.xml**

**Use:**

<dependencies>

<dependency> ---------**GroupID,ArtifactID,Version**---------</dependency>

</dependencies>

**If multi-module project,**

**-**Open **pom.xml in parent**

<dependencyManagement>

<dependencies>

<dependency> ---------**GroupID,ArtifactID,Version**---------</dependency>

</dependencies>

<dependencyManagement>

-Open **pom.xml in module**

<dependencies>

<dependency> ---------**GroupID,ArtifactID** ---------</dependency>

</dependencies>

**Adding a module as dependency to another module:**

<dependencies>

<dependency>

<groupId>groupID</groupId>

<artifactId>moduleName</artifactId>

<version>${project.version}</version>

</dependency>

</dependencies>

Click on **Reload button** on Maven pane on the right….

**Create a logback.xml file:**

<configuration>

<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">

<pattern>%date [%thread] [%-5level] %logger{40} - %message%n</pattern>

</encoder>

</appender>

<logger name="com.timothy" level="DEBUG"/>

<logger name="org.springframework" level="DEBUG"/>

<root level="INFO">

<appender-ref ref="STDOUT"/>

</root>

</configuration>

**Create a beans.xml file:**

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

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

<https://www.springframework.org/schema/beans/spring-beans.xsd>">

**<bean id= ”name” class= ”com.package.classNameImpl ” >**

**</bean>**

</beans>

**Dependency Injections:**

**1.Constructor DI:**

**<bean id= ”name” class= ”com.package.classNameImpl ” >**

**<constructor-arg ref="AnotherbeanIdInjectedByContainer"/>**

**</bean>**

**ALSO:**

-Define a **Constructor for the interface object** (The Interface we are going to depending upon.)

**2.Setter DI:**

**<bean id= ”name” class= ”com.package.classNameImpl ” >**

**<property name="interfaceObject(dependencytoourclass)InsideOurClass"**

**ref=" AnotherbeanIdInjectedByContainer "/>**

**</bean>**

**Creating a ApplicationContext container:**

**Main.java:**

private static final Logger log= LoggerFactory.getLogger(className.class);

private static final String CONFIG\_LOCATION="beans.xml";

**In main() method:**

**Container creation:**

ConfigurableApplicationContext context= new ClassPathXmlApplicationContext(CONFIG\_LOCATION);

**Creating bean (Object of interface):**

interfaceName obj = context.getBean("NameforBEAN",className.class);

**Calling a method() inside Interface:**

obj.method();

**Saving a value to variable:**

dataType Variable = Obj.method();

**Bean Lifecycle callbacks:**

**NOTE:**

**-------[BeanCreationException- USE spring-context version before 6.0.0]-----**

**Manually,**

We need to **Initialize or Destroy beans** at Some point, Mention **Init() and dispose()** methods **in XML**.

Also we can **Configure CONTAINER** to do these operations **Automatically** when a bean is generated.

Go to **beans.xml:**

We must have a method in the class for initializing the class first,

**<bean id= ”name” class= ”com.package.classNameImpl ” init-method=”methodName ” />**

It is not a recommended way, because we **need to define init for every bean** we create.

So, **we can also define default init method in top of xml file**:

<**beans** xmlns:”------------------------------------------------"

**default-init-method= ”methodName**”>

**Best Practice** is to use **Annotations**:

-Add dependency in **parent pom.xml** from MVNRepository( javax.annotations ).

Above the **init method** defined by user for the class,

- use **@PostConstruct**

And for **destroy method**, like **Finalize() keyword in java**,

Make a method called **preDestroy(),**

-use **@PreDestroy**

**Auto-wiring: [DI] -IOC container injects dependency for us.**

**-**Add in **<beans>**

xmlns:context="http://www.springframework.org/schema/context"

Add in **existing xsi:schemaLocation**:

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd

-Add below **<beans>**

**<context: annotation-config/>**

**-Remove method for the instance variable[Setter we added for setter DI] in the class.**

-**Remove <constructor> or <property>** inside the bean.

**-Remove PostProcessor** bean from xml.

**-**Add **@Autowired**  above **the instance variable in the class.**

We are injecting dependency **via Instance variable --> Field Injection**

**NOTE: <- @Autowired can also be used with Cons or Setter DI->**

­**-Init() and Destroy()** will be called using autowiring.

**Best practice:**

-**Use constructor @Autowired** or Use **Field Injection if you can’t.**

**Component?**

**-**Making a class itself a bean, aKa **Annotation based configuration.**

**- Remove bean from XML**

**- Add @Component above classes,**

**-To scan the Annotated components** we have to make some changes to

the **<context> tag** as follows**:**

**<context:component-scan base-package="packageName"/>**

**NOTE:**

**-**We should not **annotate interfaces**, because interfaces will become dependent on spring and it needs to be **decoupled from the implemetations.**

**-Remove bean names** in context.getBean() while creating bean in Class Main.

**Annotation Based Configuration: [no more xml config]**

**-Remove beans.xml file.**

**-Make a Class(AppConfig) and add annotation(s),**

**@Configuration – Makes the class as a component**

**@ComponentScan(basePackages=”packageName”) -Scans for component and methods like init and destroy will be executed automatically.**

**-As XML file being removed, we must change the way of creating the context(CONTAINER),**

context= new **AnnotationConfigApplicationContext(AppConfig.class);**

**Bean method? @Bean?:**

**Bean method contains the bean definition. We use it when we need additional configuration for a bean.**

**-Remove @Component from classes.**

**-Add @Bean methods inside AppConfig Class,**

**@Bean**

**public interfaceName beanName/ojectName (){**

**return new className();**

**}**

**Note:**

The @Import Indicates one or more [@Configuration](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html) classes to import. @Bean definitions declared in imported @Configuration classes should be accessed by using [@Autowired](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Autowired.html) injection. Either the bean itself can be autowired, or the configuration class instance declaring the bean can be autowired. The latter approach allows for explicit, IDE-friendly navigation between @Configuration class methods.

**Event Listeners in Spring:**

**- The ApplicationContext** interface has a parent interface **ApplicationEventPublisher.**

**- EventHandling** in the ApplicationContext is provided through **ApplicationEvent Class and ApplicationListener Interface.**

**-** If a bean that implements ApplicationListener interface is deployed into the context, everytime an ApplicationEvent gets published to the ApplicationContext, that bean is notified.

**Method -1:**

**-**Make a **class** implementing **ApplicationListener<typeOfEvent>**

**-**Implement the method()

@Override

public void onApplicationEvent(typeOfEvent event) {

log.info("Message");

}

**Method -2:**

**-**Using Annotations

-Use **@EventListener** above the method

@EventListener

public void onApplicationEvent(typeOfEvent event) {

log.info("Message");

}

**OR**

@EventListener(typeOfEvent)

public void anyName() {

log.info("Message");

}

**Custom Annotations? Qualifiers?**

-Right click on package -> new class -> Type name and select Annotations.

**public @interface Name {**

**}**

**-**Add these @Annotations,

@Target({ElementType.PARAMETER,ElementType.METHOD,ElementType.FIELD})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

-Now Add the **@Name** below **@Bean** in the config class and also below **@Autowired** fields in classes.

-Target()=> Context in which annotations are applicable, i.e where we can add this annotation.

-Retention()=> How long annotations with the annotated type are to be retained.

RetentionPolicy.RUNTIME=> Annotations are being recorded into class during compile time and retained by JVM at runtime.

-Qualifier=> Used to annotate other custom annotations than can be used as Qualifiers.

**@Value:**

-**Inject values for the fields**, without hardcoding in the program.

-Make **.properties file** in Resources ->New Directory(config) ->File.properties, basically the file contains key value pairs.

----After saving key-values,

How to load ? => **@PropertySource**

**-Go to** ConfigClass, below the **@Configuration**,

Add **@PropertySource(“classpath:config/File.properties”)**

Add @**Value(“${File.Variable}”) //Variable is the key in the properties file.**

**Note:**

* We can also add a default value for a field.

@**Value(“${File.Variable: Value}”)**