1 Sping Version

Spring MVC on 4

2 modules worked on Spring (IOC, AOP ,Security )

3 DI

**Dependency Injection** is a fundamental aspect of the **Spring** framework, through which the **Spring** container "injects" objects into other objects or "**dependencies**". Simply put, this allows for loose coupling of components and moves the responsibility of managing components onto the container.

**Constructor Injection** —enforcing immutability. This is the most straightforward and **recommended** way of **dependency injection**. A dependent class has a **constructor**, where all **dependencies** are set, they will be provided by **Spring** container according to XML, Java or annotation based configurations.

**Dependency injection** is a programming technique that makes a class independent of its **dependencies**. ... That enables you to replace **dependencies** without changing the class that **uses** them. It also reduces the risk that you have to change a class just because one of its **dependencies** changed.

4 Benifits of Spring bean like lifecycle also

The life cycle of a Spring bean is easy to understand. When a bean is instantiated, it may be required to perform some initialization to get it into a usable state. Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required.

there are lists of the activities that take place behind the scene between the time of bean Instantiation and its destruction



Spring bean factory is responsible for managing the life cycle of beans created through spring container.

5 disadvantge if we dnt spring

6 @Componect @Bean diff

It is used to explicitly declare a single **bean**, rather than letting **Spring** do it automatically. If any class is annotated with @**Component** it will be automatically detect by using classpath scan. We should use @**bean**, if you want specific implementation based on dynamic condition.

| **Sr. No.** | **Key** | **@Bean** | **@Component** |
| --- | --- | --- | --- |
| 1 | Auto detection | It is used to explicitly declare a single bean, rather than letting Spring do it automatically. | If any class is annotated with @Component it will be automatically detect by using classpath scan. |
| 2 | Spring Container | Bean can be created even class is outside the spring container | We can’t create bean if class is outside spring container |
| 3 | Class/Method  Level Annotation | It is a method level annotation | It is a class level annotation |
| 4 | @Configuration | It works only when class is also annotated with @Configuration | It works without @Configuration annotation |
| 5 | Use Case | We should use @bean, if you want specific implementation based on dynamic condition. | We can’t write specific implementation based on dynamic condition |

**Spring** @**Configuration** annotation helps in **Spring** annotation based **configuration**. @**Configuration** annotation indicates that a class declares one or more @Bean methods and may be processed by the **Spring** container to generate bean definitions and service requests for those beans at runtime.

7 @Contoller and @Restcontoller

1. The @**Controller** is a common annotation that is used to mark a class as Spring MVC **Controller** while @**RestController** is a special **controller** used in RESTFul web services and the equivalent of @**Controller** + @ResponseBody.

Spring boot really makes it easy to develop REST APIs with spring.

@**Component** : It is a basic auto **component** scan annotation, it indicates annotated class is an auto scan **component**. @**Controller** : Annotated class indicates that it is a **controller component**, and mainly used at the presentation layer. @Service : It indicates annotated class is a Service **component in the** business layer.

8 Constructor injection and Setter based injection diff whcih should we use how to decide that

There are many key **differences** between **constructor injection and setter injection**. Partial **dependency**: can be **injected** using **setter injection** but it is not possible by **constructor**. ... It doesn't create a new bean instance always like **constructor**. So **setter injection** is flexible than **constructor injection**.

**Field injection** should be mostly avoided. As a replacement, you should use either **constructors** or methods to **inject** your dependencies. ... **Constructors** are more suitable for mandatory dependencies and when aiming for immutability. Setters are **better** for optional dependencies.

1. **Constructor** is used to initialize the state of an object. Method is used to expose behaviour of an object. 2) **Constructor** must not have return type. ... **Setters**(), or mutators, are methods that provide the caller with an opportunity to update the value of a particular instance variable.

If **we use both constructor and setter injection**, IOC container will **use** the **setter injection**. Changes: **We can** easily change the value by **setter injection**. It doesn't create a new bean instance always like **constructor**. So **setter injection** is flexible than **constructor injection**.

9 IOC and DI diff

Inversion of control is a design principle which helps to invert the control of object creation. **Dependency Injection** is a design pattern which implements **IOC** principle. ... **DI** provides objects that an object needs.

10 Autowire how many ways to create autowiring ()

Spring provides a way to automatically detect the relationships between various beans. This can be done by declaring all the bean dependencies in Spring configuration file. So, Spring is able to utilize the [BeanFactory](https://docs.spring.io/spring-framework/docs/5.1.7.RELEASE/javadoc-api/org/springframework/beans/factory/BeanFactory.html" \t "https://dzone.com/articles/_blank) to know the dependencies across all the used beans.

The XML-configuration-based autowiring functionality has five modes – no, byName, byType, constructor, and autodetect. The default mode is no.

* **no**: It’s the default autowiring mode. It means no autowiring.
* **byName**: The byName mode injects the object dependency according to name of the bean. In such a case, the property and bean name should be the same. It internally calls the setter method.
* **byType**: The byType mode injects the object dependency according to type. So it can have a different property and bean name. It internally calls the setter method.
* **constructor**: The constructor mode injects the dependency by calling the constructor of the class. It calls the constructor having a large number of parameters.
* **autodetect**: In this mode, Spring first tries to autowire by the constructor. If this fails, it tries to autowire by using byType.

If you want to make specific bean **autowiring non-mandatory for a specific bean property**, use **required=”false”** attribute in @Autowired annotation.

11 hw many ways DI can be done

Consrtuctor,Setter

12 Qualifer

if we are using autowiring in ‘byType‘ mode and dependencies are looked for property class types. If no such type is found, an error is thrown. But, what if there are **two or more beans for same class type**.

In this case spring will not be able to choose correct bean to inject into property, and you will need to help the container using qualifiers.

To resolve a specific bean using qualifier, we need to use @Qualifier annotation along with @Autowired annotation and pass the bean name in annotation parameter

13 Bean ID and Bean name

if you keep both id and name to the bean tag, the bean is said to have 2 identifiers. you can get the same bean with any identifier. take config as

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

<bean id="fooById" name="fooByName" class="com.intertech.Foo"></bean>

<bean id="bar" class="com.intertech.Bar"></bean></beans>

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

<bean id="fooIdentifier" class="com.intertech.Foo"></bean>

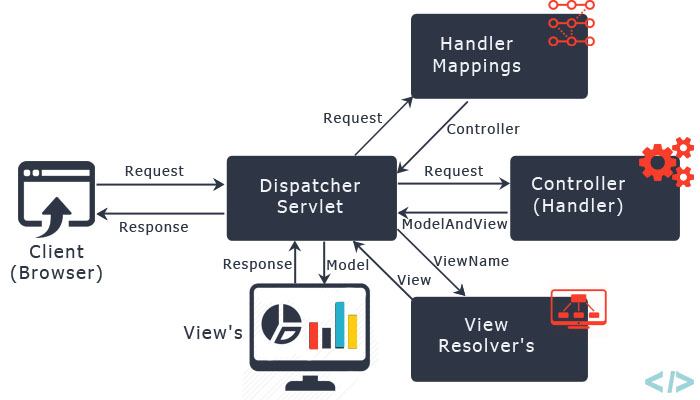
<bean name="fooIdentifier" class="com.intertech.Foo"></bean></beans>

You will get BeanDefinitionParsingException. It will say, Bean name 'fooIdentifier' is already used in this element. By the way, This is the same exception you will see if you have below config   
<bean name="fooIdentifier" class="com.intertech.Foo"></bean>  
<bean name="fooIdentifier" class="com.intertech.Foo"></bean>

14 Scopes of Spring

1. **singleton** – only one instance of the spring bean will be created for the spring container. This is the default spring bean scope. While using this scope, make sure bean doesn’t have shared instance variables otherwise it might lead to data inconsistency issues.
2. **prototype** – A new instance will be created every time the bean is requested from the spring container.
3. **request** – This is same as prototype scope, however it’s meant to be used for web applications. A new instance of the bean will be created for each HTTP request.
4. **session** – A new bean will be created for each HTTP session by the container.
5. **global-session** – This is used to create global session beans for Portlet applications.

15 Spring MVC flow



Step 1: First request will be received by DispatcherServlet.

Step 2: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request.

Step 3: So request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModelAndView object (contains Model data and View name) back to the DispatcherServlet.

Step 4: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page.

Step 5: Finally DispatcherServlet will pass the Model object to the View page to display the result.

16 Reponsible for Mapping Request URL from dispatcher servlet

@**RequestMapping** is one of the most widely used **Spring** MVC **annotation**. ... **annotation**. **RequestMapping annotation** is used to map web requests onto specific handler classes and/or handler methods. @**RequestMapping** can be applied to the controller class as well as methods

BeanNameUrlHandlerMapping is the default HandlerMapping implementation. BeanNameUrlHandlerMapping maps request URLs to beans with the same name.

@Value("#{servletContext.contextPath}")

private String servletContextPath;

I would avoid creating a dependency on the web layer from your service layer. Get your controller to resolve the path using request.getRequestURL() and pass this directly to the service:

String path = request.getRequestURL().toString();

myService.doSomethingIncludingEmail(..., path, ...);

Spring provides two kinds of IOC container, one is XMLBeanFactory and other is ApplicationContext.

| BeanFactory | ApplicationContext |+---------------------------------------+-----------------+--------------------------------+| Annotation support | No | Yes || BeanPostProcessor Registration | Manual | Automatic || implementation | XMLBeanFactory | ClassPath/FileSystem/WebXmlApplicationContext|| internationalization | No | Yes || Enterprise services | No | Yes || ApplicationEvent publication | No | Yes |+---------------------------------------+-----------------+---------

* FileSystemXmlApplicationContext Beans loaded through the full path.
* ClassPathXmlApplicationContext Beans loaded through the CLASSPATH
* XMLWebApplicationContext and AnnotationConfigWebApplicationContext beans loaded through the web application context.
* AnnotationConfigApplicationContext Loading Spring beans from Annotation based configuration.

Spring Framework comes with two IOC containers – [BeanFactory](https://www.baeldung.com/spring-beanfactory) and [ApplicationContext](https://www.baeldung.com/spring-classpathxmlapplicationcontext). The BeanFactory is the most basic version of IOC containers, and the ApplicationContext extends the features of BeanFactory.

BeanFactory **loads beans on-demand, while** ApplicationContext **loads all beans at startup**. Thus, BeanFactory is lightweight as compared to ApplicationContext. Let's understand it with an example.

LAZY loading and Eagar Loading

**the** Student **object is not initialized**. In other words, **only the** BeanFactory **is initialized**. The beans defined in our BeanFactory will be loaded only when we explicitly call the getBean() method.

Let's check the initialization of our Student bean where we're manually calling the getBean() method:

Student bean loads successfully. Hence, the BeanFactory only loads the bean when it is required.

ApplicationContext enhances BeanFactory in a more framework-oriented style and provides several features that are suitable for enterprise applications.

For instance, it **provides** [messaging (i18n or internationalization)](https://www.baeldung.com/spring-classpathxmlapplicationcontext" \l "2-internationalization-with-messagesource) functionality, [event publication](https://www.baeldung.com/spring-events) functionality, **annotation-based dependency injection**, and **easy integration with Spring AOP features**.

Apart from this, the ApplicationContext supports almost all types of bean scopes, but the BeanFactory only supports two scopes — Singleton and Prototype. Therefore, it's always preferable to use ApplicationContext when building complex enterprise applications.

The ApplicationContext **automatically registers** BeanFactoryPostProcessor **and** BeanPostProcessor at startup. On the other hand, the BeanFactory does not register these interfaces automatically.