Spring Core

**IOC vs. DI:**

### IOC (Inversion of Control)

* In traditional programming application flow is decided by objects that are statically created and assigned one to another. IOC is a design principle where giving control to the container to get instance of object is called Inversion of Control., means instead of you are creating object using new operator, let the container do that for you.
* In object oriented programming, we have several basic techniques to implement IOC like
  + Using factory pattern
  + Using service locator pattern
  + Dependency Injection

### DI (Dependency Injection)

* Dependency injection is a design pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.
* Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies. The object does not look up its dependencies, and does not know the location or class of the dependencies
  + Constructor Injection
  + Setter Injection

**Constructor Injection:**

* Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency.
* If we have a constructor with multiple arguments then we will get ambiguity. To resolve that we will use “index” attribute.

**<bean id**=**"exampleBean" class**=**"examples.ExampleBean">**

**<constructor-arg index**=**"0" value**=**"7500000"/>**

**<constructor-arg index**=**"1" value**=**"42"/>**

**</bean>**

**Setter Injection:**

* *Setter-based* DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.
* Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.

**Circular Dependency:**

* If two beans are depends on each one then it will come circular dependency
* For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.
* We can avoid this by configuring “setter injection” rather than “constructor injection”.

**Spring Bean Scopes:**

 Spring Framework supports five scopes, three of which are available only if you use a web-aware ApplicationContext. We can create a Custom Scope also.

* **Singleton**:
  + *The singleton scope is the default scope in Spring*.
  + When we define a bean and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
  + Singleton bean is differing from “singleton” design patter. Singleton design patter will creates only one instance of a particular class per class loader. The scope of the Spring singleton is best described as *per container and per bean*. This means that if you define one bean for a particular class in a single spring container, then the Spring container creates one *and only one* instance of the class defined by that bean definition.
  + Singleton scope bean objects are shared among multiple thread. So it’s suggestible to avoid creating of “instance variables” inside of the singleton class.

**Lazy Loading in case of singleton scoped beans**:

* + By default “ApplicationContext” implementations will creates & initializes ‘singleton’ scope beans as part of initialization process. Pre-instantiation is desirable, because errors in the configurations are identified immediately.
  + A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup. This can be done by “lazy-init = true”

**<bean id**=**"lazy" class**=**"com.foo.ExpensiveToCreateBean" lazy-init**=**"true"/>**

**<bean name**=**"not.lazy" class**=**"com.foo.AnotherBean"/>**

* + However, when a lazy-initialized bean is a dependency of a singleton bean that is not lazy-initialized, the ApplicationContext creates the lazy**-initialized bean at startup**, because it must satisfy the singleton’s dependencies.
* **prototype**
  + When we define a bean and it is scoped as a prototype, the Spring IoC container creates new bean instance for each request. That is, the bean is injected into another bean or you request it through a getBean() method call on the container.
  + As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans.
  + Like other scopes, spring does not manage entire life cycle of prototype bean. Container initialize, configures the prototype object and hands to client with no further record. **“Initialization life cycle call backs”** are called by the container regardless of scope; in the case of prototype destruction life cycle call backs are not called.
  + Manually we have to clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding. To get the spring container to release resources held by prototype-scoped beans, try using a custom “[bean post-processor](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s08.html#beans-factory-extension-bpp)”, which holds a reference to beans that need to be cleaned up.
  + Usually controller and DAO class objects we declared as a singleton because it does not hold any conversational **state**.

**Singleton bean with prototype dependency:**

* + When singleton-scoped beans had dependencies on prototype beans, be aware that *dependencies are resolved at instantiation time*. Thus if you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean. The prototype instance is the sole instance that is ever supplied to the singleton-scoped bean.
  + However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime. You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only *once*, when the Spring container is instantiating the singleton bean and resolving and injecting its dependencies. If you need a new instance of a prototype bean at runtime more than once, see [Section 3.4.7, “Method injection”](spring-framework-reference.pdf)
  + **Method injection**
* **Request:**
  + The request, session, and global session scopes are *only* available if you use a web-aware Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext, you get an IllegalStateException complaining about an unknown bean scope.
  + The spring container creates a new instance of the LoginAction bean by using the loginAction bean definition for each and every HTTP request. That is, the loginAction bean is scoped at the HTTP request level. You can change the internal state of the instance that is created as much as you want, because other instances created from the same loginAction bean definition will not see these changes in state; they are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.

<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>

* **Session:**
  + <bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
  + The spring container creates a new instance of the UserPreferences bean by using the userPreferences bean definition for the lifetime of a single HTTP Session. In other words, the userPreferences bean is effectively scoped at the HTTP Session level. As with request-scoped beans, you can change the internal state of the instance that is created as much as you want, knowing that other HTTP Session instances that are also using instances created from the same userPreferences bean definition do not see these changes in state, because they are particular to an individual HTTP Session. When the HTTPSession is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.
* **Global Session:**
  + The global session scope is similar to the standard HTTP Session scope ([described above](https://docs.spring.io/spring/docs/3.0.0.M4/reference/html/ch03s05.html#beans-factory-scopes-session)), and applies only in the context of portlet-based web applications. The portlet specification defines the notion of a global Session that is shared among all portlets that make up a single portlet web application. Beans defined at the global session scope are scoped (or bound) to the lifetime of the global portlet Session.
  + If you write a standard Servlet-based web application and you define one or more beans as having global session scope, the standard HTTP Sessionscope is used, and no error is raised.

**Auto wiring:**

* Auto wiring is nothing but allowing spring container to resolves/injects other beans automatically to our bean by inspecting the contents of ApplicationContext.
* Auto wiring can significantly reduce the need to specify properties or constructor arguments.

### Limitations of auto wiring

* Explicit dependencies in property and constructor-arg settings always override auto wiring. I.e. the values injected by auto wiring will be overridden by property/constructor injection.
* You cannot auto wire so-called simple properties such as primitives, Strings & arrays.
* Multiple bean definitions within the container may match with the “type” specified in setter or constructor argument to be auto wired. For example, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.
* Information may not be available to tools that may generate the documentation from a spring container.

### Types of auto wiring:

### No: by this option is default for spring framework

* 1. **byName**: This option enables the dependency injection based on bean names. When auto wiring a property in bean, property name is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, an error is raised.
  2. **byType**: This option enables the dependency injection based on bean types. When auto wiring a property in bean, property’s class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. If no such bean is found, an error is raised.

<bean id="employee" class="com.fss.EmployeeBean" autowire="byType">

<property name="fullName" value="Lokesh Gupta"/>

</bean>

<bean id="department" class="com.fss.DepartmentBean" >

<property name="name" value="Human Resource" />

</bean>

* 1. **Constructor**: Auto wiring by constructor is similar to byType, but applies to constructor arguments. In auto wire enabled bean, it will look for class type of constructor arguments, and then do an auto wire by type on all constructor arguments. Please note that if there isn’t exactly one bean of the constructor argument type in the container, a fatal error is raised.
  2. **Autodetect** : Autowiring by autodetect uses either of two modes i.e. constructor or byType modes. First it will try to look for valid constructor with arguments, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the autowire byType mode is chosen.
* We can apply @autowired on property or getter method of property. In both cases is equivalent to byType autowiring.
* When @autowired on constructor is equivalent to constructor injection.

### Qualifier:

* 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. so spring container can’t able to decide to which bean needs to be inject
* 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.

@Autowired

 @Qualifier ("finance")

### Required:

### If we want to mark a dependency as optional then we will use @required = false. So if no dependency is found container won’t throw any exception.

@Autowired (required=false)

@Qualifier ("finance")

private DepartmentBean departmentBean;

* If you want to apply optional autowiring at global level i.e. for all properties in all beans; use below configuration setting.

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor">

    <property name="requiredParameterValue" value="false" />

</bean>

### Excluding a bean from Autowiring:

* By default, autowiring scan and matches all bean definitions in scope. If you want to exclude some bean definitions so that they cannot be injected through autowiring mode, you can do this using ‘autowire-candidate’ set to false.
* Using ‘**autowire-candidate**‘as false totally exclude a bean from being an autowire candidate. It totally exclude that specific bean definition from being available to the autowiring infrastructure

<!--Will not participate in autowiring-->

     <bean id="finance" class="com.fss.Department" autowire-candidate="false">

         <property name="name" value="Finance" />

    </bean>

* Another option is to limit autowire candidates based on pattern-matching against bean names. The top-level <beans/> element accepts one or more patterns within its ‘**default-autowire-candidates**‘attribute. For example, to limit autowire candidate status to any bean whose name ends with ‘Impl’, provide a value of ‘\*Impl’. To provide multiple patterns, define them in a comma-separated list.

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

<beans default-autowire-candidates="\*Impl,\*Dao">

### Spring Bean Life Cycle:

* Spring bean factory is responsible for managing the life cycle of beans created through spring container. Spring framework provides following 4 ways for controlling life cycle events of bean:
  + InitializingBean and DisposableBean interfaces
  + Other Aware interfaces.
  + Custom init() and destroy() methods in bean configuration file
  + @PostConstruct and @PreDestroy annotation
* **InitializingBean & DisposableBean**:
* This interface allows a bean to perform initialization work after all necessary properties on the bean have been set by the container. The InitializingBean interface specifies a single method:
  + void afterPropertiesSet() throws Exception;
  + void destroy() throws Exception;
* This is not a preferable way to initialize the bean because it tightly couples your bean class with spring container.
* **Custom init() & destroy()**
* The default init and destroy methods in bean configuration file can be defined in two ways
  + Bean local definition applicable to a single bean
  + Global definition applicable to all beans defined in beans context

<beans>

    <bean id="demoBean" class="com.DemoBean" init-method="customInit" destroy- method="customDestroy"></bean>

</beans>

* Whereas global definition is given as below. These methods will be invoked for all bean definitions given under <beans> tag.

<beans default-init-method="customInit" default-destro-ethod="customDestroy">

        <bean id="demoBean" class="com.DemoBean"></bean>

</beans>

* **@PostConstruct & @PreDestroy:**
* @PostConstruct annotated method will be invoked after the bean has been constructed using default constructor and just before its instance is returned to requesting object.
* @PreDestroy annotated method is called just before the bean is about be destroyed inside bean container.
* **Other Aware interfaces:**
  + - Spring offers a range of Aware interfaces that allow beans to indicate to the container that they require a certain infrastructure dependency. Each interface will require you to implement a method to inject the dependency in bean.

|  |  |  |
| --- | --- | --- |
| **AWARE INTERFACE** | **METHOD TO OVERRIDE** | **PURPOSE** |
| ApplicationContextAware | void **setApplicationContext** (ApplicationContext applicationContext) throws BeansException; | Interface to be implemented by any object that wishes to be notified of the ApplicationContext that it runs in. |
| ApplicationEventPublisherAware | void **setApplicationEventPublisher** (ApplicationEventPublisher applicationEventPublisher); | Set the ApplicationEventPublisherthat this object runs in. |
| BeanClassLoaderAware | void setBeanClassLoader(ClassLoader classLoader); | Callback that supplies the bean class loader to a bean instance. |
| BeanFactoryAware | void setBeanFactory(BeanFactory beanFactory) throws BeansException; | Callback that supplies the owning factory to a bean instance. |
| BeanNameAware | void setBeanName(String name); | Set the name of the bean in the bean factory that created this bean. |

**Final Static Bean:**

* If you have final static fields in some bean and you want to use those references as beans in application context file to inject into another beans, you can do so using **<util:constant>** tag.

<util:constant id="MANAGER" static-field ="com.MANAGER" />

<util:constant id="DIRECTOR" static-field="com.DIRECTOR" />

**ContextLoaderListener**:

* In Spring Web MVC, the ContextLoaderListener reads the Spring configuration file where all the beans are declared. It will then initialize all the singleton beans defined in this file. Kindly note it will not initialize any “prototype” beans because the prototype beans are initialized every time when requested.
* The location and name of the spring configuration file are mentioned in the web.xml.

**Handler Mapping**:

* Map a request to a handler along with a list of [interceptors](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-handlermapping-interceptor) for pre- and post-processing. The mapping is based on some criteria the details of which vary by HandlerMapping implementation.
* The two main HandlerMapping implementations
  + RequestMappingHandlerMapping which supports @RequestMapping annotated methods.
  + SimpleUrlHandlerMapping which maintains explicit registrations of URI path patterns to handlers.

**Handler** **Adapter**:

* Help the DispatcherServlet to invoke a handler mapped to a request regardless of how the handler is actually invoked. For example, invoking an annotated controller requires resolving annotations. The main purpose of a HandlerAdapteris to shield the DispatcherServlet from such details.

**Interception:**

* All HandlerMapping implementations support handler interceptors that are useful when you want to apply specific functionality to certain requests. Interceptors must implement HandlerInterceptor from the org.springframework.web.servlet package with three methods
  + preHandle(..) — *before* the actual handler is executed
  + postHandle(..) — *after* the handler is executed
  + afterCompletion(..) — *After the complete request has finished* and view was generated
* The preHandle(..) method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns true, the handler execution chain will continue; when it returns false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain. See [Interceptors](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-config-interceptors) in the section on MVC configuration for examples of how to configure interceptors. You can also register them directly via setters on individual HandlerMapping implementations.
* Note that postHandle is less useful with @ResponseBody and ResponseEntity methods for which the response is written and committed within the HandlerAdapter and before postHandle. That means it’s too late to make any changes to the response such as adding an extra header. For such scenarios you can implement ResponseBodyAdvice and either declare it as an [Controller Advice](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-controller-advice) bean or configure it directly on RequestMappingHandlerAdapter

<mvc:interceptors>

    <bean id="loggerInterceptor" class="interceptor class name"/>

</mvc:interceptors>

Spring is light weight and loosely coupled frame work.

Dependency injection has 3 ways

Setter Injection

Constructor Injection

Interface Injection.

Spring bean is no way related to java bean. Where java bean should have default constructor but in spring bean it is not a mandatory.

We have 4 ways to control bean life cycle.

InitializingBean and DisposableBean call back Interface.

Custom init() and destroy() in configuration file.

@PostConstruct and @PreDestroy annotation.

Other Aware Interfaces.



InitializinBean interface contain one method: void afterPropertiesSet().

DisposableBean interface contain one method: void destroy()

This is not preferable way to initialize bean bcz our bean class should implement this interface. So bean class and spring container is tightly coupled.

AOP

* By using aop we can separate cross-cutting functionalities i.e. adding different types of services to the application at runtime automatically.
* Using AOP the business logic and cross-cutting functionalities (Services) are implemented separately and executed at run time as combine.
* Aspect is nothing but the name of the cross cutting functionality not its implementation and all.
* Throw Advice is the marker interface in spring framework.

MVC

**Spring internal flow:**

**Dispatcher Servlet:**

* The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), and as such is declared in the web.xml of your web application.

<servlet>

<servlet-name>example</servlet-name>

<servletc-lass>org.springframework.web.servlet.DispatcherServlet</s-c>

<load-on-startup>1</load-on-startup>

</servlet>

* Upon initialization of a DispatcherServlet, Spring MVC looks for a file named [servlet-name]-servlet.xml in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.
* The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications like “capable of resolving themes (see [Section 17.9, “Using themes”](https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-themeresolver)), and that it knows which Servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it.

### Handler Mapping:

* In previous versions of Spring, users were required to define one or more HandlerMapping beans in the web application context to map incoming web requests to appropriate handlers. With the introduction of annotated controllers, you generally don't need to do that because the RequestMappingHandlerMapping automatically looks for @RequestMapping annotations on all @Controller beans.

### Handler Adapter:

* Helps the DispatcherServlet to invoke a handler mapped to a request regardless of the handler is actually invoked. For example, invoking an annotated controller requires resolving various annotations. Thus the main purpose of a HandlerAdapter is to shield the DispatcherServlet from such detail.

**Overview**:

1. The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.
2. The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
3. The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
4. If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process. See [Section 17.10, “Spring's multipart (file upload) support”](https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-multipart) for further information about multipart handling.
5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
6. If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

Handler exception resolvers that are declared in the WebApplicationContext pick up exceptions that are thrown during processing of the request. Using these exception resolvers allows you to define custom behaviors to address exceptions.

The Spring DispatcherServlet also supports the return of the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request is straightforward: the DispatcherServlet looks up an appropriate handler mapping and tests whether the handler that is found implements the LastModified interface. If so, the value of the long getLastModified(request) method of the LastModified interface is returned to the client.

You can customize individual DispatcherServlet instances by adding Servlet initialization parameters (init-param elements) to the Servlet declaration in the web.xml file. See the following table for the list of supported parameters.

**Table 17.2. DispatcherServlet initialization parameters**

| **Parameter** | **Explanation** |
| --- | --- |
| contextClass | Class that implements WebApplicationContext, which instantiates the context used by this Servlet. By default, theXmlWebApplicationContext is used. |
| contextConfigLocation | String that is passed to the context instance (specified by contextClass) to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. In case of multiple context locations with beans that are defined twice, the latest location takes precedence. |
| namespace | Namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet. |

**Way to read Property File in spring:**

* In XML, new properties files can be made accessible to Spring via the context:property-placeholder.

<context:property-placeholder location="classpath:foo.properties" />

* Spring 3.1 also introduces **the new @PropertySource annotation**, as a convenient mechanism for adding property sources to the environment.

@Configuration

@PropertySource("classpath:foo.properties")

public class PropertiesWithJavaConfig { }

* Usnig @Value

@Value( "${jdbc.url}" )

private String jdbcUrl;