Spring Integration

<https://chandanpandey.com/2015/04/27/what-is-spring-integration-and-how-can-it-help-solve-enterprise-integration-challenges/>

<https://chandanpandey.com/2012/10/29/spring-integration-getting-started-tutorial/>

**Spring**

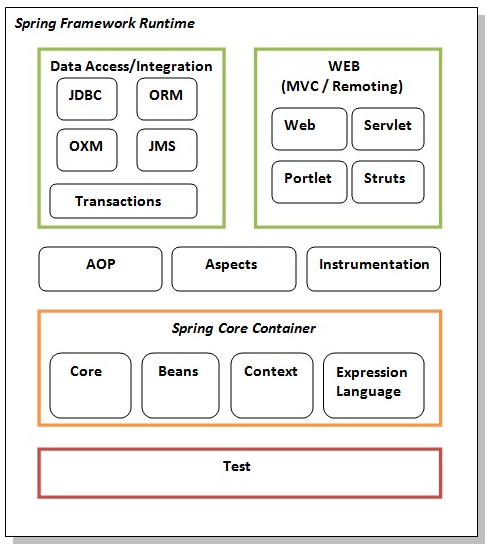
* Developed by Rod jonhson in 2003
* Intent was to replace EJB
* Named after Spring season as in Spring framework logo is leaf
* Heavy use of DI vai Runtime Polymorphism and Association Realtionship, i.e. has-a relationship i.e. aggregation
* Problems with EJB framework
* aim was to created reusable server-side object, so as to make enterprise application development easy.
* It promised to achive the above by only programming to simple java beans
* It achieved that via the concept of beans, e.g. enity-beans, session beans, MDBs
* Though it provided loose coupling vi Inversion of Control, but it had to done in a more explicit fashion
* For example the Session Bean interface defines ejbRemove, ejbPassivate (stored to secondary storage), and ejbActivate (restored from passive state). You don't get to control when these methods are called, just what they do. The container calls us, we don't call it.
* In spring this is handled implicitly via the container.
* EJB supported transctions, messaging, RPC etc. but only with the help of an App Server.
* Spring also needs a app server, its only that Spring is not specific to web applications You can run a standalone spring application. Your main class will be the entrypoint, you can load the bean definitions using ClassPathXmlApplicationContext
* IoC container

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

1. to instantiate the application class
2. to configure the object
3. to assemble the dependencies between the objects

There are two types of IoC containers. They are:

* BeanFactory
* ApplicationContext
* applicationContext vs BeanFactory(core container)
* both are containers
* appContext will eagerly instantiate beans, while Bean Factory will lazily when the 1st request for that bean is made.
* The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.
* How DI works internally in spring
* Spring Modules



* Inheritance in spring

<https://www.mkyong.com/spring/spring-bean-configuration-inheritance/>

done via the **parent** attritube of the bean tag e.g.

<bean id="BaseCustomerMalaysia" class="com.mkyong.common.Customer">

<property name="country" value="Malaysia" /></bean>

<bean id="CustomerBean" parent="BaseCustomerMalaysia">

<property name="action" value="buy" />

<property name="type" value="1" />

</bean>

You can also mark the base class as abstract via abstract=true attribute of bean tag.

Now if you try to create bean of abstract calls, you get bean creation failure exception

* How to inject vector instead of ArrayList in spring
* util schema
* by default spring will inject ArrayList for list, and HashMap for Map collection property.
* To explicitly define your concreate collection Class we use Util schema

<bean id="CustomerBean" class="com.mkyong.common.Customer">

<property name="lists">

<util:list list-class="java.util.ArrayList">

<value>1</value>

<value>2</value>

<value>3</value>

</util:list>

</property> </bean>

* Autowiring in spring

<http://www.javatpoint.com/autowiring-in-spring>

* It’s always good to combine both ‘auto-wire’ and ‘dependency-check’ together, to make sure the property is always auto-wire successfully.

<bean id="customer" class="com.mkyong.common.Customer"

autowire="autodetect" dependency-check="objects />

<bean id="person" class="com.mkyong.common.Person" />

* Spring properties dependency checks
* In Spring,you can use dependency checking feature to make sure the required properties have been set or injected.
* 4 dependency checking modes are supported:
* none – No dependency checking.
* simple – If any properties of primitive type (int, long,double…) and collection types (map, list..) have not been set, UnsatisfiedDependencyException will be thrown.
* objects – If any properties of object type have not been set, UnsatisfiedDependencyException will be thrown.
* all – If any properties of any type have not been set, an UnsatisfiedDependencyException
* will be thrown.
* P.S The default mode is none
* Component scan
* Dependency Injection with Factory Method in Spring

Spring framework provides facility to inject bean using factory method. To do so, we can use two attributes of bean element.

* factory-method: represents the factory method that will be invoked to inject the bean.
* factory-bean: represents the reference of the bean by which factory method will be invoked. It is used if factory method is non-static.
* Annotations in spring
* <http://www.javatpoint.com/autowiring-in-spring>
* <https://www.mkyong.com/spring/spring-auto-scanning-components/>
* Constructor autowiring is by Type internally,
* Auto-wiring using annotation in spring
* <https://www.mkyong.com/spring/spring-auto-wiring-beans-with-autowired-annotation/>
* <context:annotation-config> vs <context:component-scan>
* <http://howtodoinjava.com/spring/spring-mvc/spring-mvc-difference-between-contextannotation-config-vs-contextcomponent-scan/>
* Inner beans
* <https://www.mkyong.com/spring/spring-inner-bean-examples/>
* Lookup method injection OR
* Injecting a prototype/Session bean into a singleton bean

<bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">

<property name="validator" ref="validator"/>

</bean>

<bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/>

* In Spring, most of the beans we work with are Singletons. If a singleton bean is wired with yet another singleton bean, there is absolutely no problem. But if it is wired with a bean which is of different scope, say prototype. There is a problem when the bean lifecycles are different. Consider a singleton bean A which needs to use a non-singleton (prototype) bean B, perhaps on each method invocation on A. The container will only create the singleton bean A once, and thus only get the opportunity to set the properties once. There is no opportunity for the container to provide bean A with a new instance of bean B every time one is needed.
* To solve this issue, we have 2 approaches:

1. Lookup Method injection
2. Scoped Proxies

**Lookup Method injection:** For this, we have to declare the beans as follows:

|  |  |
| --- | --- |
|  | <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">      <lookup-method name="getValidator" bean="validator"/>  </bean>    <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator"/> |

Whenever we define a bean with lookup methods, Spring creates a subclass of the bean and overrides those methods which are marked as lookup-methods. And this subclassed bean gets registered into the context. The subclass delegates all the non-lookup methods to the original class. For the lookup methods, it overrides the implementation. So in our example, when getValidator() is called, it returns a new validator instance.

We can roughly imagine our new subclass(registered in container) like this:

|  |  |
| --- | --- |
|  | requestProcessor = new RequestProcessor(){      public RequestValidator getValidator(){          return context.getBean("validator");      }  }; |

**Scoped Proxies:** This can be implemented as:

|  |  |
| --- | --- |
|  | <bean id="requestProcessor" class="com.pramati.spring.RequestProcessor">      <property name="validator" ref="validator"/>  </bean>    <bean id="validator" scope="prototype" class="com.pramati.spring.RequestValidator">      <!-- This instructs the container to proxy the current bean-->      <aop:scoped-proxy/>  </bean> |

Remember, in case of look up method injection, proxy is created for singleton bean. But in case of scoped proxies, proxy is created for prototype bean and wired into the singleton bean during the process of registering the singleton bean in the context. The proxy thus created understands the scope and returns instances based on the requirements of the scope. So in our case, *requestProcessor* holds a reference to proxy in place of *validator*.

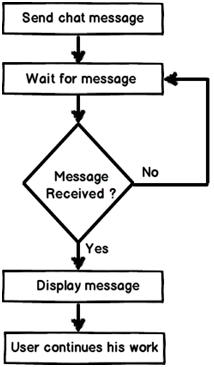
More details are here: <https://prasanthnath.wordpress.com/2013/03/21/injecting-a-prototype-bean-into-a-singleton-bean/>

* Spring AOP to write to file via 2 or more threads
* Spring core vs spring mvc
* How Spring jdbcTemplate works
* Spring tranascation
* How Spring JMSTemplate works
* Spring AOP
* Exception handling in spring
* Garbage collection in spring
* Cyclic dependency in spring
* Spring 3.0 new features
* Spring annotations
* IOC vs DI
* <https://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO>

The main goal of Inversion of control and Dependency Injection is to remove dependencies of an application. This makes the system more decoupled and maintainable.

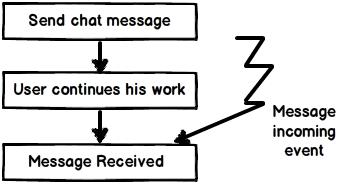
First let’s try to understand IOC (Inversion of control). If you go back to old computer programming days, program flow used to run in its own control. For instance let’s consider a simple chat application flow as shown in the below flow diagram.

1. End user sends chat message.
2. Application waits for the message from the other end.
3. If no message is found it goes to Step 2 or else moves to Step 4.
4. Displays the message.
5. User continues with his work ahead.

[](http://3.bp.blogspot.com/-Sb34Uxuvjmk/UZCCb_05h1I/AAAAAAAAEgA/qOSoSUw04e0/s1600/1st+image.jpg)

Now if you analyze the program flow closely, it’s sequential. The program is in control of himself. Inversion of control means the program delegates control to someone else who will drive the flow. For instance if we make the chat application event based then the flow of the program will go something as below:-

1. End user sends chat message.
2. User continues with his work ahead.
3. Application listens to events. If a message arrives event is activated and message is received and displayed.

[](http://3.bp.blogspot.com/-4Y4krJiAn0U/UZCCxNPNDjI/AAAAAAAAEgI/p1bqKzyZPw4/s1600/2nd+image.jpg)

If you see the program flow it’s not sequential, its event based. So now the control is inverted. So rather than the internal program controlling the flow, events drive the program flow. Event flow approach is more flexible as their no direct invocation which leads to more flexibility.

A word of caution here, do not conclude that IOC are implemented by only events. You can delegate the control flow by callback delegates, observer pattern, events, DI (Dependency injection) and lot of other ways.

IOC (Inversion of control) is a general parent term while DI (Dependency injection) is a subset of IOC. IOC is a concept where the flow of application is inverted. So for example rather than the caller calling the method.

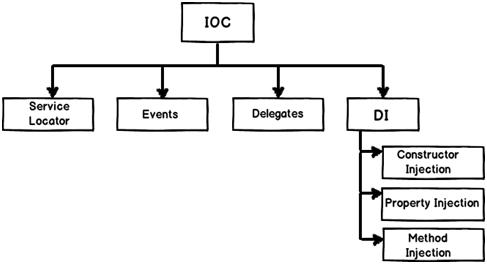
SomeObject.Call();

Will get replaced with an event based approach as shown below.

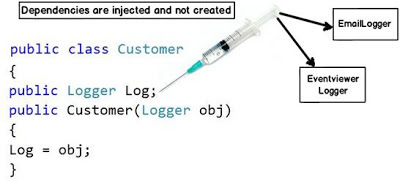
SomeObject.WhenEvent += Call();

In the above code the caller is exposing an event and when that event occurs he is taking action. It’s based on the Hollywood principle “Don’t call us we will call you”. In Hollywood when artists used to give auditions the judges would say them “Don’t call us we will call you”.

The above approach makes code more flexible as the caller is not aware of the object methods and the object is not aware of caller program flow.

[](http://1.bp.blogspot.com/-2JkcEGnJVrY/UZCDAoPjk8I/AAAAAAAAEgQ/ilz1zM5fc78/s1600/3rd+image.jpg)

DI provides objects that an object needs. So rather than the dependencies construct themselves they are injected by some external means. For instance let’s say we have the following below class “Customer” who uses a “Logger” class to log errors. So rather than creating the “Logger” from within the class, you can inject the same via a constructor as shown in the below code snippet.

[](http://4.bp.blogspot.com/-vUOcNsQcrUk/UZCDROepyWI/AAAAAAAAEgY/8NbtFggPfh4/s1600/4th+image.jpg)

The biggest benefit achieved by the above approach is “Decoupling”. You can now invoke the customer object and pass any kind of “Logger” object as shown in the below code.

Customer obj = new Customer(new EmailLogger());

Customer obj1 = new Customer(new EventViewerLogger());

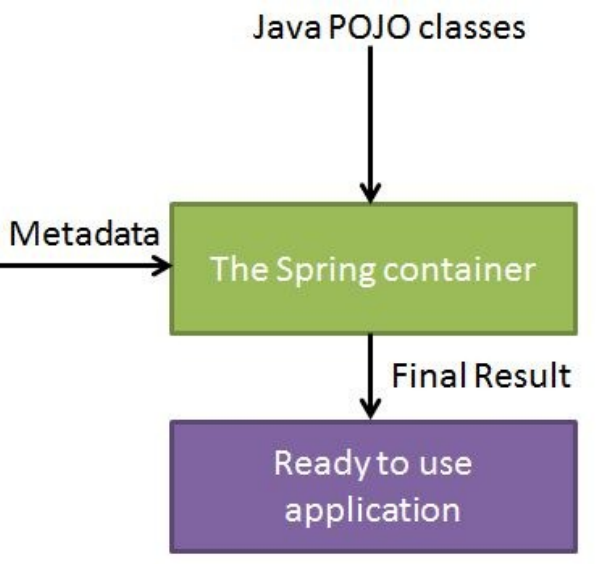
So summarizing the differences.

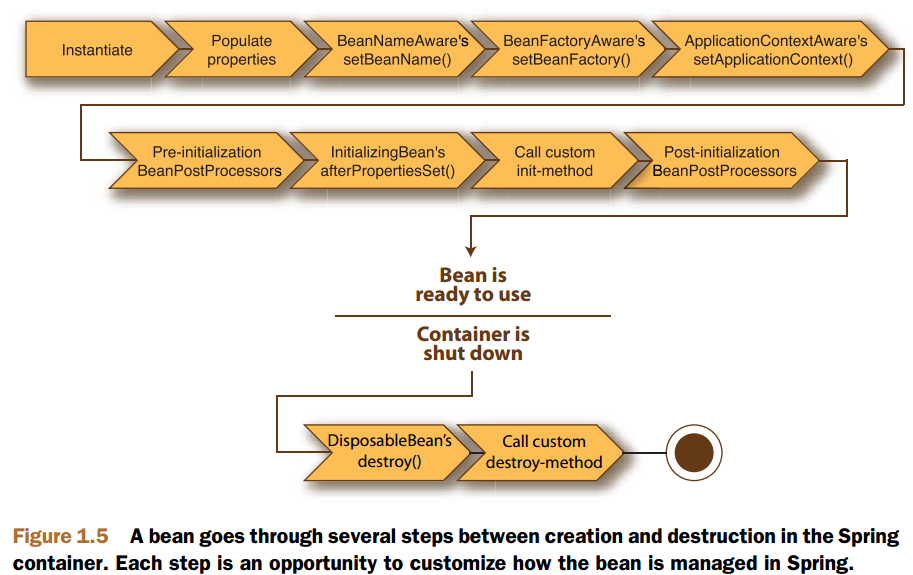
**Inversion of control** :- It’s a generic term and implemented in several ways (events, delegates etc).

**Dependency injection** :- DI is a subtype of IOC and is implemented by constructor injection, setter injection or method injection.

Below is a nice video which demonstrates IOC ( Inversion of control) and how its is different from DI ( Dependency injection)

* why use spring framework : Because it simplifies java esp enterprise application development via
* Lightweight and minimally invasive development with POJOs
* Loose coupling via DI and interface orientation
* Declarative programing via Aspects and common conventions
* Boilerpate code reducing via templates and aspects
* One of the ways Spring empowers POJOs is by assembling them using dependency injection.
* DI for a 5yr old: When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired. What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.
* DI: let’s look at both these words separately, dependency is some sort of association between things, e.g. x->Y let’s say x is dependent on Y. The word injection states that the dependency y will be automatically injected in X at runtime via IoC. Try to stick an eg. Say theTextEditor and spellChecker one
* The act of creating associations between application components is commonly referred to as wiring.
* System services such as logging, transaction management, and security often find their way into components whose core responsibility is something else. These system services are commonly referred to as cross-cutting concerns because they tend to cut across multiple components in a system.
* In short, aspects ensure that POJOs remain plain.
* **IoC Container/Spring Container**: the spring container is at the core of the spring framework, the container reads the beans definitions are creates them, manages their lifecycle and also wires dependencies among beans vi DI.



* **Common IoC falvours are: SpringBeanFactoryContainer, SpringApplicationContextConainer**, the later provides more enhanced functionalities over the BeanFactoryContainer as in reading properties files etc. But BeanFactory is still used in application like mobile or applet where memory and speed are significant
* SpringBeanFactroyContainer vs SpringApplicationContextContainer ?
* Commonly used ApplicationContext implementations:
* **Spring XML Config vs Annotations Config**
* Annotation scanning & auto-wiring help us on big projects, where many hundreds or thousands of beans may need to be wired.
* However, tend to lead to an “implicit” configuration model with property files & property-placeholder injection being the only source of configurability. This is a weaker & less flexible model than the full power of Spring.
* Using Spring XML for key configuration enables more direct & explicit control over application configuration. Subclassed & extensible configuration can be supported, enabling richer & more powerful OO techniques such as the strategy & delegate patterns.
* With an XML file per application, or per customer, this enables powerful application-/ customer-specific configuration. Large shared codebases can be enhanced with small sections of pluggable configuration, to deliver widely configurable applications.
* Bean Life-cycle:
* 
* 1 Spring instantiates the bean.
* 2 Spring injects values and bean references into the bean’s properties.
* 3 If the bean implements BeanNameAware, Spring passes the bean’s ID to the setBeanName()method.
* 4 If the bean implements BeanFactoryAware, Spring calls the setBeanFactory()method, passing in the bean factory itself.
* 5 If the bean implements ApplicationContextAware, Spring will call the setApplicationContext()method, passing in a reference to the enclosing application context.
* 6 If any of the beans implement the BeanPostProcessorinterface, Spring calls their postProcessBeforeInitialization()method.
* 7 If any beans implement the InitializingBeaninterface, Spring calls their afterPropertiesSet()method. Similarly, if the bean was declared with an init-method, then the specified initialization method will be called.
* 8 If there are any beans that implement BeanPostProcessor, Spring will call their

postProcessAfterInitialization()method.

* 9 At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed.
* 10 If any beans implement the DisposableBeaninterface, then Spring will call their destroy()methods. Likewise, if any bean was declared with a destroymethod, then the specified method will be called.
* **Partical usage of xxxAware interfaces ?**

The xxxAware interface is a common pattern used within the Spring framework. They are typically used to allow a Spring managed bean to be given an object (via the interfaces setXxx method) at Spring bootstrap time.

Springs documentation says this about the [Aware](http://docs.spring.io/spring/docs/3.2.x/javadoc-api/org/springframework/beans/factory/Aware.html) interface, which is a super interface to the two you mention:

Marker super-interface indicating that a bean is eligible to be notified by the Spring container of a particular framework object through a callback-style method.

As Sotirious points out, the Aware interface has the feel of the listener, callback, or observer design patterns.

Usage would look like this:

@Component

public MyBean implements BeanFactoryAware {

private BeanFactory beanFactory;

@Override

public void setBeanFactory(final BeanFactory beanFactory) {

this.beanFactory = beanFactory;

}

public void myMethod() {

//I can now use beanFactory here

}

}

During bootstrapping, Spring will examine each bean to determine if it implements any of the xxxAware interfaces. When one is found, it invokes the interface method, providing the piece of information that is being asked for. In the example above, Spring calls MyBean#setBeanFactoryproviding its BeanFactory.

Of course, in many situations, it is not entirely necessary to use these interfaces. For example, the ApplicationContextAware interface can be circumvented by simply @Autowireding an ApplicationContext into a bean.

@Component

public class MyOtherBean {

@Autowired

private ApplicationContext applicationContext;

public void someMethod() {

//I can use the ApplicationContext here.

}

}

* BeanNameAware makes the object aware of its bean name. It is best used in pre annotation config spring (2.x). You could reference the bean from a locator by its name then.
* BeanFactoryAware gives the bean access to the beanfactory that created it,The BeanFactoryAware interface is used during the initialization of an ApplicationContext. It has been used in Spring 2 to customize the weaving of beans before they get initialized. An example would be in order to additionally add pointcuts at load time (LTW) e.g. for autogenerated find methods of DAOs. Another usage could be to load a minimized context for test, to speed up tests (lazy initialization would be a better practice)
* **Inner Beans:** In Spring framework, whenever a bean is used for only one particular property, it’s advise to declare it as an inner bean. And the inner bean is supported both in setter injection ‘property‘ and constructor injection ‘constructor-arg‘.they shouldn’t be accessible from anywhere else and should be defined as [inner beans](http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/beans.html#beans-inner-beans).
* By using inner beans, those beans are implicitly made anonymous but also scoped prototype, which doesn’t mean squat since they won’t be reused anywhere else.
* With annotations configuration, this is something that is done under the cover when you set a new instance in the body of the method
* I acknowledge it renders the Spring beans definition file harder to read **but** with the graphical representation feature brought by Spring IDE, this point is moot

* Bean Defintion:

- class

- id/name

- scope

- constructor-args

- properties

-autowiring-mode

- lazy-init-mode

-init-mode

-destory-mode

* More details in attached xls:



* **@postConstruc vs init-method vs beanPostProcessor ?**

No practically I don't think there is any difference but there are priorities in the way they work. PostConstruct, init-method are BeanPostProcessors

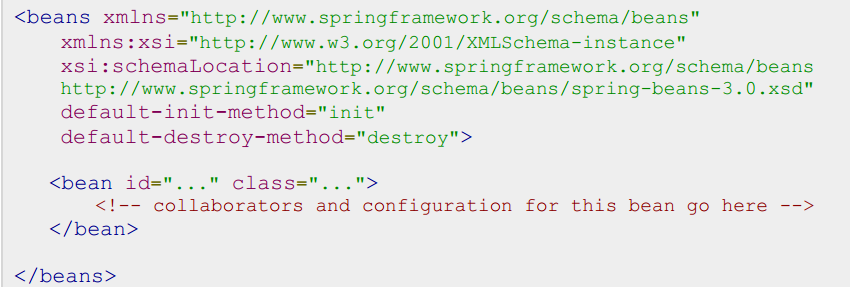
1. PostConstruct is a JSR-250 annotaion while init-method is Spring's way of having an initializing method .
2. If you have postconstruct , this will be called first before the init methods are called
3. If your bean implements initializing bean and overrides afterPropertiesSet , first post constructed is called, then the afterPropertiesSet and then init-method.

* **Default initialization and destroy methods**

If you have too many beans having initialization and or destroy methods with the same name,

you don't need to declare init-method and destroy-method on each individual bean. Instead

framework provides the flexibility to configure such situation using default-initmethod and default-destroy-methodattributes on the <beans> element as follows:



* Inheritance in spring via parent attribute, also we can create super-class templates via abstract=true bean definition
* **Value vs value-ref and key vs key-ref**

<https://pranavkumarblog.wordpress.com/tag/value-ref/>

* P-namespace

<https://pranavkumarblog.wordpress.com/2013/08/18/springs-p-namespace-in-xml-configuration/>

* Injecting Spring collections
* Auto-wiring: The Spring container

can autowire relationships between collaborating beans without using <constructor -arg> and <property>

e.g. *<bean id="customer" class="com.mkyong.common.Customer" autowire="byName"/>*

5 Auto-wiring modes are supported.

no – Default, no auto wiring, set it manually via “ref” attribute

byName – Auto wiring by property name. If the name of a bean is same as the name of other bean property, auto wire it.

byType – Auto wiring by property data type. If data type of a bean is compatible with the data type of other bean property, auto wire it.

constructor – byType mode in constructor argument.

autodetect – If a default constructor is found, use “autowired by constructor”; Otherwise, use “autowire by type”.

In my view, Spring ‘auto-wiring’ make development faster with great costs – it added complexity for the entire bean configuration file, and you don’t even know which bean will auto wired in which bean.

In practice, i rather wire it manually, it is always clean and work perfectly, or better uses[@Autowired annotation](http://www.mkyong.com/spring/spring-auto-wiring-beans-with-autowired-annotation/), which is more flexible and recommended.

* Spring Batch Notes:
* Components of a Flat File Item Reader:
* 