# Spring Dependency Injection

**IoC Container:** IoC container gets information from the XML file/java configuration class and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

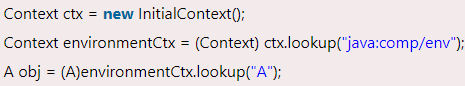
There are two types of IoC containers. They are:

1. BeanFactory
2. ApplicationContext

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**

**Dependency Injection:** Dependency Injection makes our programming code loosely coupled.

There can be various ways to get the resource:

* we get the resource (instance of A class) directly by new keyword.
* Another way is factory method.
* we can get the resource by JNDI (Java Naming Directory Interface)

These are called dependency lookup. There are mainly two problems of dependency lookup.

* **tight coupling** the dependency lookup approach makes the code tightly coupled. If resource is changed, we need to perform a lot of modification in the code.
* **Not easy for testing** This approach creates a lot of problems while testing the application especially in black box testing.

I’ll use java code for configuration (there is xml and annotation configuration as well). Here we don’t need any xml configuration at all. For this, we need a class with **@configuration**, **@componentscan** annotations. In that class we can define our beans with **@bean** annotation.

# Constructor and Setter Injection

* We can use constructor-based dependency injection for mandatory dependencies and setter-based injection for optional dependencies.
* Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
* 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.



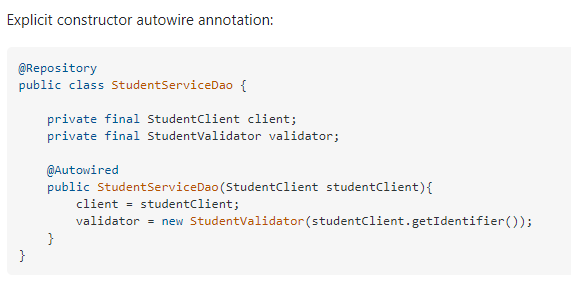
In general, it is suggested to use constructor injection.

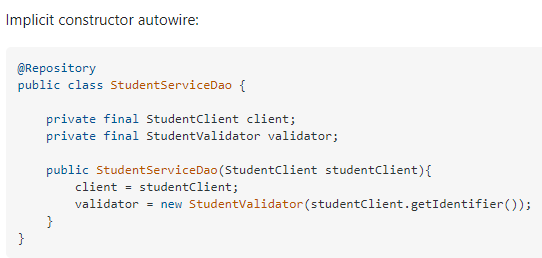


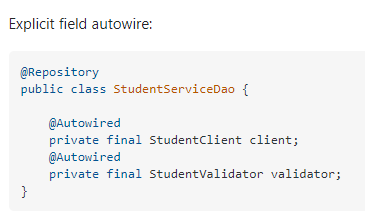
# Auto-Wiring

**@Autowired** annotation is part of the Spring framework. After enabling annotation injection, we can use autowiring on properties, setters, and constructors.

Note: Spring uses the bean's name as a default qualifier value.



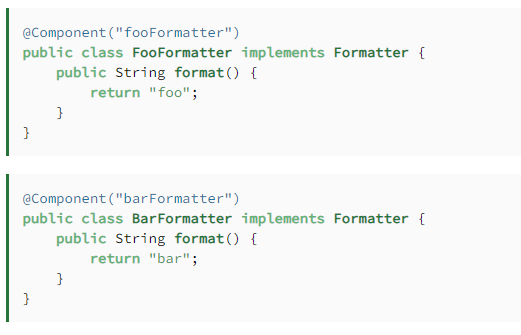
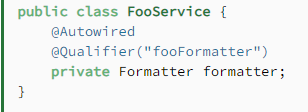




# Auto wiring with qualifier

In case we have multiple implementations of a bean (for example multiple classes implement an interface, and we want a bean of that interface reference), spring won’t know which implementation to choose and Spring will throw a **NoUniqueBeanDefinitionException** exception.

In this case, we can specify the name of our desired bean with **@Qualifier** annotation.



# Spring JDBC

For raw query use **JdbcTemplate** or **NamedParameterJdbcTemplate.** They are wrappers around raw JDBC and handles the boilerplate code. They also make sure to handle the checked exception of JDBC to unchecked exception so we can handle them at runtime.

They both work similarly with minor difference. NamedParameterJdbcTemplate is recommended to use as we don’t need to specify the order of the variable in the raw query here. We can add the variables by name using helper methods. So, it is much more convenient.

**NamedParameterJdbcTemplate** class is a template class with a basic set of JDBC operations, allowing the use of named parameters rather than traditional '?' placeholders

Check Documentation for available methods.

**Best Use cases:**

* **Row Mapper**: When each row of a ResultSet maps to a domain Object, can be implemented as private inner class.

It is used to process one record of ResultSet at a time.

* **RowCallbackHandler**: When no value is being returned from callback method for each row, e.g., writing row to a file, converting rows to an XML, Filtering rows before adding to collection. Very efficient as ResultSet to Object mapping is not done here.
* **ResultSetExtractor**: When multiple rows of ResultSet map to a single Object. Like when doing complex joins in a query one may need to have access to entire ResultSet instead of single row of **rs** to build complex Object and you want to take full control of ResultSet. Like Mapping the rows returned from the join of TABLE1 and TABLE2 to a fully-reconstituted TABLE aggregate.

It is used to process multiple records of ResultSet at a time.

* **ParameterizedRowMapper** is used to create complex objects