**Q: Difference between @Primary vs @Autowired with @Qualifier annotations**

Read [@Primary](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Primary.html) is the "default".

If a bean has @Autowired without any @Qualifier, and multiple beans of the type exist, the candidate bean marked @Primary will be chosen, i.e. it is the default selection when no other information is available, i.e. when @Qualifier is missing.

A good use case is that initially you only had one bean of the type, so none of the code used @Qualifier.

When you then add another bean, you then also add @Qualifier to both the old and the new bean, so any @Autowired can choose which one it wants.

By also adding @Primary to the old original bean, you don't have to add @Qualifier to all the existing @Autowired. They are "grandfathered" in, so to speak.

@Primary is also good if e.g. 95% of @Autowired wants a particular bean. That way, only the @Autowired want the other bean(s) need to specify @Qualifier. That way, you have primary beans that all autowired wants, and @Qualifier is only used to request an "alternate" bean.

**Q: What will happen if we define same bean(Scope=”Singleton”) twice in Spring applicationContext.xml with different id?**

Ans: in spite of setting scope is “Singleton”, spring will create two instances.

**applicationContext.xml:**

<bean id=*"helloBean"* class=*"com.spring.core.HelloWorld"* scope=*"singleton"*>

<property name=*"name"* value=*"Srihith"* />

</bean>

<bean id=*"helloBean2"* class=*"com.spring.core.HelloWorld"* scope=*"singleton"*>

<property name=*"name"* value=*"Prasad"* />

</bean>

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"SpringBeans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloBean");

obj.printHello();

System.***out***.println("Obj=-->"+obj);

HelloWorld obj2 = (HelloWorld) context.getBean("helloBean2");

System.***out***.println("Obj2=-->"+obj2);

obj2.printHello();

Output:

Spring 3: Hello ! Srihith

Obj=-->com.spring.core.HelloWorld@2d209079

Obj2=-->com.spring.core.HelloWorld@6bdf28bb

Spring 3: Hello! Prasad

**Q: what will happen if the constructor is private in spring bean class> will spring create an instance?**

Ans: yes spring will create an instance.

Ex:

**package** com.spring.core;

/\*\*

\* Spring bean

\*

\*/

**public** **class** HelloWorld {

**private** String name;

**private** HelloWorld() {

System.***out***.println("Private Constructor...>");

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** printHello() {

System.***out***.println("Spring 3 : Hello ! " + name);

}

}

**package** com.spring.core;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** App {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext(

"SpringBeans.xml");

HelloWorld obj = (HelloWorld) context.getBean("helloBean");

obj.printHello();

System.***out***.println("Obj=-->"+obj);

HelloWorld obj2 = (HelloWorld) context.getBean("helloBean2");

System.***out***.println("Obj2=-->"+obj2);

obj2.printHello();

}

}

Output:

Private Constructor...>

Private Constructor...>

Spring 3 : Hello ! Srihith

Obj=-->com.spring.core.HelloWorld@2d209079

Obj2=-->com.spring.core.HelloWorld@6bdf28bb

Spring 3 : Hello ! Prasad

Q: **BeanFactory vs ApplicationContext?**

**Bean Factory**

Bean instantiation/wiring

**Application Context**

Bean instantiation/wiring

Automatic BeanPostProcessor registration

Automatic BeanFactoryPostProcessor registration

Convenient MessageSource access (for i18n)

ApplicationEvent publication

So if you need any of the points presented on the Application Context side, you should use ApplicationContext.

Q: **What is Automatic registration of BeanPostProcessor?**

You shouldn't actually care unless you have a very specific use case. In certain environments, you may want to turn off some additional features that ApplicationContext has over BeanFactory. These are summarized [in this table](http://docs.spring.io/spring/docs/4.0.2.RELEASE/spring-framework-reference/html/beans.html#context-introduction-ctx-vs-beanfactory)

As for "Automatic registration of BeanPostProcessor", the ApplicationContext will detect beans that implement BeanPostProcessor and will use them to enrich the context while it is loaded. This is a very powerful feature of Spring, check [Container extension points](http://docs.spring.io/spring/docs/4.0.2.RELEASE/spring-framework-reference/html/beans.html#beans-factory-extension)

/\*\*

\*

\*/

**package** com.spring.core;

**import** org.springframework.beans.BeansException;

**import** org.springframework.beans.factory.config.BeanPostProcessor;

**public** **class** CustomBeanPostProcessor **implements** BeanPostProcessor {

**public** Object postProcessAfterInitialization(Object arg0, String arg1) **throws** BeansException {

System.***out***.println("postProcessAfterInitialization arg0:"+arg0+", arg1:"+arg1);

**return** arg0;

}

**public** Object postProcessBeforeInitialization(Object arg0, String arg1) **throws** BeansException {

System.***out***.println("postProcessBeforeInitialization arg0:"+arg0+", arg1:"+arg1);

**return** arg0;

}

}

**Output:**

Private Constructor...>

postProcessBeforeInitialization arg0:com.spring.core.HelloWorld@54c562f7, arg1:helloBean

postProcessAfterInitialization arg0:com.spring.core.HelloWorld@54c562f7, arg1:helloBean

Private Constructor...>

postProcessBeforeInitialization arg0:com.spring.core.HelloWorld@318ba8c8, arg1:helloBean2

postProcessAfterInitialization arg0:com.spring.core.HelloWorld@318ba8c8, arg1:helloBean2

Q: Is Singleton Spring bean is thread-safe?

Ans: no. Ideally spring beans do not have state.

Q: when should I use JTA transaction manager and when JPA, and what benefit and disadvantages does they have?

If you want to delegate managed transactions to your Application Server and handle complex transactions across multiple resources you need to use the JtaTransactionManager,

The Spring Framework gives you the choice of when to scale your application to a fully loaded application server. Gone are the days when the only alternative to using EJB CMT or JTA was to write code with local transactions such as those on JDBC connections, and face a hefty rework if you need that code to run within global, container-managed transactions. With the Spring Framework, only some of the bean definitions in your configuration file, rather than your code, need to change

**Spring Framework transaction management**

### **Global transactions**

Global transactions enable you to work with multiple transactional resources, typically relational databases and message queues. The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA UserTransaction normally needs to be sourced from JNDI, meaning that you *also* need to use JNDI in order to use JTA. Obviously the use of global transactions would limit any potential reuse of application code, as JTA is normally only available in an application server environment.

### Local transactions

Local transactions are resource-specific, such as a transaction associated with a JDBC connection. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Because the application server is not involved in transaction management, it cannot help ensure correctness across multiple resources. (It is worth noting that most applications use a single transaction resource.) Another downside is that local transactions are invasive to the programming model.

The TransactionDefinition interface specifies:

* *Isolation*: The degree to which this transaction is isolated from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
* *Propagation*: Typically, all code executed within a transaction scope will run in that transaction. However, you have the option of specifying the behavior in the event that a transactional method is executed when a transaction context already exists. For example, code can continue running in the existing transaction (the common case); or the existing transaction can be suspended and a new transaction created. *Spring offers all of the transaction propagation options familiar from EJB CMT*. To read about the semantics of transaction propagation in Spring, see [Section 11.5.7, “Transaction propagation”](https://docs.spring.io/spring-framework/docs/4.0.x/spring-framework-reference/html/transaction.html#tx-propagation).
* *Timeout*: How long this transaction runs before timing out and being rolled back automatically by the underlying transaction infrastructure.
* *Read-only status*: A read-only transaction can be used when your code reads but does not modify data. Read-only transactions can be a useful optimization in some cases, such as when you are using Hibernate.