***Cyclic Dependency*** *– A circular dependency occurs when a bean A depends on another bean B, and the bean B depends on bean A as well:*

*Bean A → Bean B → Bean A*

*Of course, we could have more beans implied:*

*Bean A → Bean B → Bean C → Bean D → Bean E → Bean A*

*With a circular dependency, Spring cannot decide which of the beans should be created first since they depend on one another. In these cases, Spring will raise a* ***BeanCurrentlyInCreationException*** *while loading context.*

*It can happen in Spring when using constructor injection. If we use other types of injections, we shouldn’t have this problem since the dependencies will be injected when they are needed and not on the context loading.*

***The Workarounds –***

* *Redesign* ***-*** *We should try to redesign the components properly so that their hierarchy is well designed and there is no need for circular dependencies.*
* *Use @Lazy - A simple way to break the cycle is by telling Spring to initialize one of the beans lazily. So, instead of fully initializing the bean, it will create a proxy to inject it into the other bean. The injected bean will only be fully created when it’s first needed.*
* *Use Setter/Field Injection - One of the most popular workarounds.*
* *Use @PostConstruct - Another way to break the cycle is by injecting a dependency using @Autowired on one of the beans and then using a method annotated with @PostConstruct to set the other dependency.*

@Component **public** **class** **CircularDependencyA** {

@Autowired

**private** CircularDependencyB circB;

@PostConstruct

**public** **void** **init**() { circB.setCircA(this); }

**public** CircularDependencyB **getCircB**() { **return** circB; }

}

* *Implement ApplicationContextAware and InitializingBean - If one of the beans implements ApplicationContextAware, the bean has access to Spring context and can extract the other bean from there. By implementing InitializingBean, we indicate that this bean has to do some actions after all its properties have been set. In this case, we want to manually set our dependency.*

***Prototype Bean Injection Problem -*** *A common problem arises when we want to inject a prototype-scoped bean into a singleton-scoped bean. In this case, as the singleton bean is initialized only once per application context, we need to apply extra logic to initialize the prototype-scoped bean so that it is created every time. This problem is mainly encountered when we have to inject the non-thread-safe bean instances into other beans.*

***The Workarounds*** *–*

* *Lookup Method Injection - The easiest way to solve the scoped bean problem is using the lookup method injection using @Lookup annotation. Such lookup methods are overridden by the container to redirect them back to the BeanFactory for a getBean() call via CGLIB.*

@Component

**public** **class** XmlUtils {

@Lookup

**public** JAXBContext getJAXBContext() {

**return** **null**;

}

}

* *Using ApplicationContext- If we already have access to ApplicationContext in the singleton bean, we can use it to obtain the prototype-scoped bean instance whenever we need it. Usually, a new instance is obtained using the getter method.*

**public** **class** **SingletonAppContextBean** **implements** **ApplicationContextAware** {

**private** ApplicationContext applicationContext;

**public** PrototypeBean **getPrototypeBean**() {

**return** applicationContext.getBean(PrototypeBean.class);

}

@Override **public** **void** **setApplicationContext**(ApplicationContext applicationContext) **throws** BeansException

{

this.applicationContext = applicationContext;

}

}

*However, this approach has serious disadvantages. It contradicts the principle of inversion of control, as we request the dependencies from the container directly.*

*Also, we fetch the prototype bean from the applicationContext within the SingletonAppcontextBean class. This means coupling the code to the Spring Framework.*

* *ObjectFactory Interface - Spring provides the ObjectFactory<T> interface to produce on demand objects of the given type:*

public class SingletonObjectFactoryBean {

@Autowired

private ObjectFactory<PrototypeBean> prototypeBeanObjectFactory;

public PrototypeBean getPrototypeInstance() {

return prototypeBeanObjectFactory.getObject();

}

}

*Let’s have a look at getPrototypeInstance() method; getObject() returns a brand new instance of PrototypeBean for each request. Here, we have more control over initialization of the prototype.*

*Also, the ObjectFactory is a part of the framework; this means avoiding additional setup to use this option.*

*Interchanging of stereotype annotations – It actually doesn’t make difference. Only @Repository annotation , It does the same but it also helps us to achieve the exception translation feature so that Spring will wrap up any exception thrown by your persistence layer with a DataAccessException.*

*@Repository’s job is to catch persistence-specific exceptions and re-throw them as one of Spring’s unified unchecked exceptions.*

*For this, Spring provides PersistenceExceptionTranslationPostProcessor, which we are required to add in our application context (already included if we’re using Spring Boot):*

*<bean class=*

*"org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>*

*This bean post processor adds an advisor to any bean that’s annotated with @Repository.*