**Circular Dependency in Spring Boot**

**What is Circular Dependency?**

A circular dependency occurs when two or more Spring beans depend on each other directly or indirectly, causing an infinite loop during dependency injection.

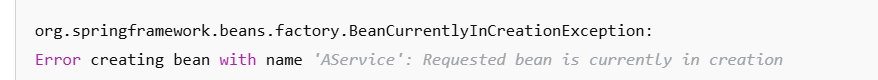
Example of Circular Dependency



 AService depends on BService.

 BService depends on AService.

 Spring fails to resolve this, leading to an error:



***Ways to Solve Circular Dependency in Spring***

1. **Use @Lazy Annotation (Best for Singleton Beans)**

Mark one of the dependencies as @Lazy, so it is **only created when needed**, breaking the loop.



✅ Now, BService is not created **immediately**, avoiding the circular reference.

**2. Constructor Injection (Breaks Circular Dependency)**

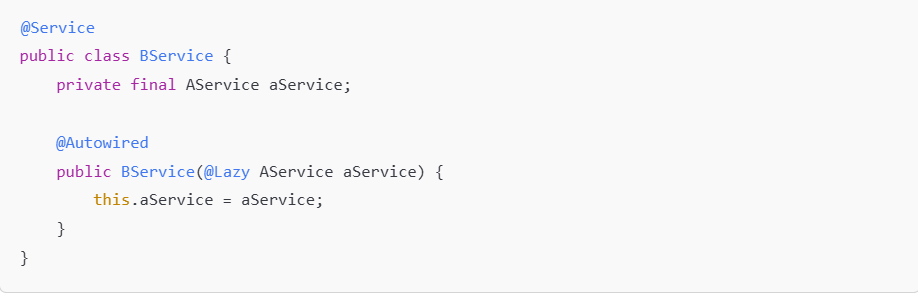
Using **constructor injection** instead of field injection helps **detect circular dependencies early**, but it doesn't solve them directly.

If you use constructor injection:





❌ This **won’t work** because Spring **cannot instantiate either class first**.



✅ Now, AService is created only **when needed**, avoiding the loop.

**3. Use @PostConstruct for Late Initialization**

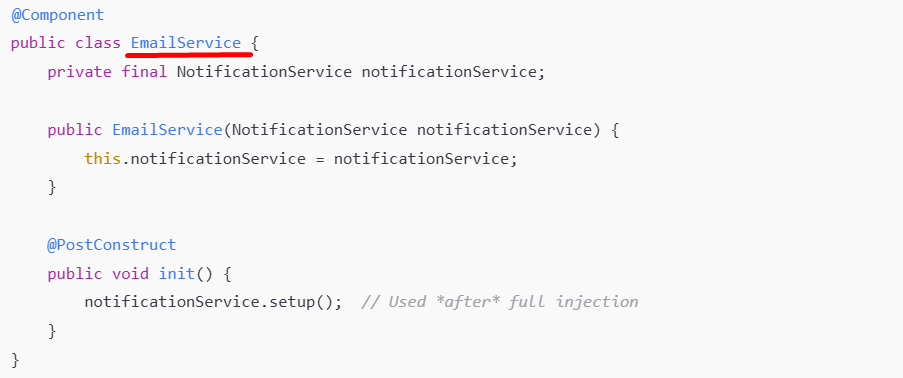
Instead of injecting the dependency directly, initialize it **after the bean is created**.

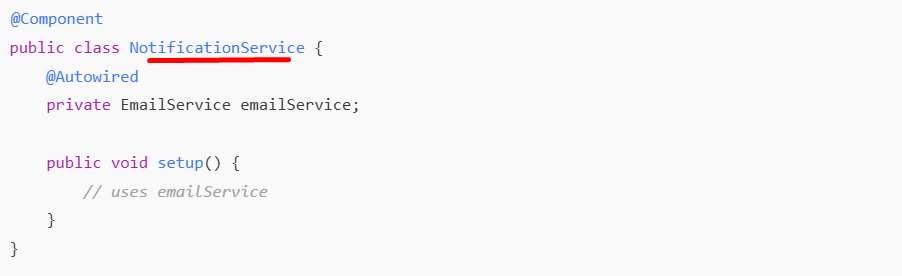
**🔍 How it Works (Mechanism)**

* During Spring's bean lifecycle, it **constructs all beans first**, then performs **dependency injection**.
* After injection, it calls the **@PostConstruct method**.
* So if you delay the use of a dependency until after full context initialization, you avoid the circular use during creation.

📘 Example – Delayed Use via @PostConstruct

🔁 Scenario: Circular Dependency Between EmailService and NotificationService





When you use @Autowired **on a field (in notification service)**, Spring doesn't inject it during the constructor. Instead, Spring:

1. **Creates the bean instance first** (using a no-arg or constructor),
2. Then **injects fields and setter dependencies afterward** in a **separate step** during the bean initialization phase.

**✅ Why It Works**

* NotificationService uses @Autowired, which is **field injection** and happens **after** EmailService is constructed.
* EmailService doesn’t use the notificationService in the constructor; it uses it only after full injection (@PostConstruct), breaking the loop at runtime.

**⚠️ Caution**

* This works only when **you can delay using the dependency** until after all beans are constructed.
* If both classes need to use each other **in their constructors**, @PostConstruct won’t help — you'd still get a circular reference error.

✅ Spring creates the bean first, then **injects the dependency later**, avoiding circular injection.

**4. Refactor to Remove the Circular Dependency**

Sometimes, the best solution is to **redesign your classes** to remove direct dependencies.

**Use an Interface or a Common Bean**



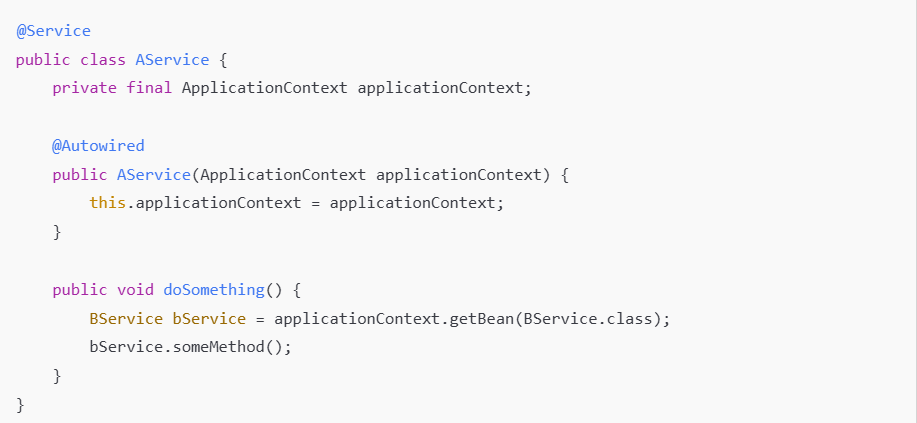




✅ Now, AService and BService **both depend on CommonService**, breaking the cycle.

**5. Use ApplicationContext (Manual Bean Fetching)**

You can manually fetch the dependency from Spring's ApplicationContext, but this is not recommended unless necessary.



✅ This works, but **it makes unit testing harder**.

From the above 5 solutions, which solution is Best?

| **Solution** | **Use Case** |
| --- | --- |
| @Lazy Annotation | Works best for **singleton beans** |
| Constructor + @Lazy | Best for **avoiding field injection issues** |
| @PostConstruct | Works for **late initialization** |
| Redesigning Classes | Best if **architecture allows** |
| ApplicationContext | Last resort (not recommended for general use) |

**Final Thoughts**

 **Circular dependencies are a sign of bad design**—try refactoring first.

 If refactoring isn't possible, **use @Lazy or @PostConstruct**.

 Avoid using ApplicationContext.getBean() unless necessary.