1. What Is a Circular Dependency?

It happens when a bean A depends on another bean B, and the bean B depends on the 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

2. What Happens in Spring

When Spring context is loading all the beans, it tries to create beans in the order needed for them to work completely. For instance, if we didn’t have a circular dependency, like the following case:

Bean A → Bean B → Bean C

Spring will create bean C, then create bean B (and inject bean C into it), then create bean A (and inject bean B into it).

But, when having 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 you use other types of injections you should not find this problem since the dependencies will be injected when they are needed and not on the context loading.

3. A Quick Example

Let’s define two beans that depend on one another (via constructor injection):

@Component

public class CircularDependencyA {

private CircularDependencyB circB;

@Autowired

public CircularDependencyA(CircularDependencyB circB) {

this.circB = circB;

}

}

@Component

public class CircularDependencyB {

private CircularDependencyA circA;

@Autowired

public CircularDependencyB(CircularDependencyA circA) {

this.circA = circA;

}

}

Now we can write a Configuration class for the tests, let’s call it TestConfig, that specifies the base package to scan for components. Let’s assume our beans are defined in package “com.baeldung.circulardependency”:

@Configuration

@ComponentScan(basePackages = { "com.baeldung.circulardependency" })

public class TestConfig {

}

And finally we can write a JUnit test to check the circular dependency. The test can be empty, since the circular dependency will be detected during the context loading.

@RunWith (SpringJUnit4ClassRunner.class)

@ContextConfiguration (classes = { TestConfig.class })

public class CircularDependencyTest {

@Test

public void givenCircularDependency\_whenConstructorInjection\_thenItFails() {

// Empty test; we just want the context to load

}

}

If you try to run this test, you will get the following exception:

BeanCurrentlyInCreationException: Error creating bean with name 'circularDependencyA':

Requested bean is currently in creation: Is there an unresolvable circular reference?

4. The Workarounds

We will show some of the most popular ways to deal with this problem.

**4.1. Redesign**

When you have a circular dependency, it’s likely you have a design problem and the responsibilities are not well separated. You should try to redesign the components properly so their hierarchy is well designed and there is no need for circular dependencies.

If you cannot redesign the components (there can be many possible reasons for that: legacy code, code that has already been tested and cannot be modified, not enough time or resources for a complete redesign…), there are some workarounds to try.

**4.2. Use @Lazy**

A simple way to break the cycle is saying Spring to initialize one of the beans lazily. That is: 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.

To try this with our code, you can change the CircularDependencyA to the following:

@Component

public class CircularDependencyA {

private CircularDependencyB circB;

@Autowired

public CircularDependencyA (@Lazy CircularDependencyB circB) {

this.circB = circB;

}

}

If you run the test now, you will see that the error does not happen this time.

**4.3. Use Setter/Field Injection**

One of the most popular workarounds, and also what Spring documentation proposes, is using setter injection.

Simply put if you change the ways your beans are wired to use setter injection (or field injection) instead of constructor injection – that does address the problem. This way Spring creates the beans, but the dependencies are not injected until they are needed.

Let’s do that – let’s change our classes to use setter injections and will add another field (message) to CircularDependencyB so we can make a proper unit test:

@Component

public class CircularDependencyA {

private CircularDependencyB circB;

@Autowired

public void setCircB(CircularDependencyB circB) {

this.circB = circB;

}

public CircularDependencyB getCircB() {

return circB;

}

}

@Component

public class CircularDependencyB {

private CircularDependencyA circA;

private String message = "Hi!";

@Autowired

public void setCircA(CircularDependencyA circA) {

this.circA = circA;

}

public String getMessage() {

return message;

}

}

Now we have to make some changes to our unit test:

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes = { TestConfig.class })

public class CircularDependencyTest {

@Autowired

ApplicationContext context;

@Bean

public CircularDependencyA getCircularDependencyA() {

return new CircularDependencyA();

}

@Bean

public CircularDependencyB getCircularDependencyB() {

return new CircularDependencyB();

}

@Test

public void givenCircularDependency\_whenSetterInjection\_thenItWorks() {

CircularDependencyA circA = context.getBean(CircularDependencyA.class);

Assert.assertEquals("Hi!", circA.getCircB().getMessage());

}

}

The following explains the annotations seen above:

@Bean: To tell Spring framework that these methods must be used to retrieve an implementation of the beans to inject.

@Test: The test will get CircularDependencyA bean from the context and assert that its CircularDependencyB has been injected properly, checking the value of its message property.

4.4. Use @PostConstruct

Another way to break the cycle is injecting a dependency using @Autowired on one of the beans, and then use a method annotated with @PostConstruct to set the other dependency.

Our beans could have the following code:

@Component

public class CircularDependencyA {

@Autowired

private CircularDependencyB circB;

@PostConstruct

public void init() {

circB.setCircA(this);

}

public CircularDependencyB getCircB() {

return circB;

}

}

@Component

public class CircularDependencyB {

private CircularDependencyA circA;

private String message = "Hi!";

public void setCircA(CircularDependencyA circA) {

this.circA = circA;

}

public String getMessage() {

return message;

}

}

And we can run the same test we previously had, so we check that the circular dependency exception is still not being thrown and that the dependencies are properly injected.

4.5. 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. 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.

The code of our beans would be:

@Component

public class CircularDependencyA implements ApplicationContextAware, InitializingBean {

private CircularDependencyB circB;

private ApplicationContext context;

public CircularDependencyB getCircB() {

return circB;

}

@Override

public void afterPropertiesSet() throws Exception {

circB = context.getBean(CircularDependencyB.class);

}

@Override

public void setApplicationContext(final ApplicationContext ctx) throws BeansException {

context = ctx;

}

}

@Component

public class CircularDependencyB {

private CircularDependencyA circA;

private String message = "Hi!";

@Autowired

public void setCircA(CircularDependencyA circA) {

this.circA = circA;

}

public String getMessage() {

return message;

}

}

Again, we can run the previous test and see that the exception is not thrown and that the test is working as expected.

5. In Conclusion

There are many ways to deal with circular dependencies in Spring. The first thing to consider is to redesign your beans so there is no need for circular dependencies: they are usually a symptom of a design that can be improved.

But if you absolutely need to have circular dependencies in your project, you can follow some of the workarounds suggested here.

The preferred method is using setter injections. But there are other alternatives, generally based on stopping Spring from managing the initialization and injection of the beans, and doing that yourself using one strategy or another.

Another way to avoid circular dependency injection by field level autowiring.