**Circular Dependencies in Spring**

[**https://www.baeldung.com/circular-dependencies-in-spring**](https://www.baeldung.com/circular-dependencies-in-spring)

## 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.

@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;

    }

}

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.

@Component

public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired

    public CircularDependencyA(@Lazy CircularDependencyB circB) {

        this.circB = circB;

    }

}

One of the most popular workarounds, is using setter injection. This way Spring creates the beans, but the dependencies are not injected until they are needed.

@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;

    }

}

## Create a Deployable War File

# [Spring boot wep application to IBM Websphere](https://stackoverflow.com/questions/33266186/spring-boot-wep-application-to-ibm-websphere)

The first step in producing a deployable war file is to provide a SpringBootServletInitializer subclass and override its configure method. Doing so makes use of Spring Framework’s Servlet 3.0 support and lets you configure your application when it is launched by the servlet container. Typically, you should update your application’s main class to extend SpringBootServletInitializer, as shown in the following example:

*@SpringBootApplication*

**public** **class** Application **extends** SpringBootServletInitializer {

*@Override*

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(Application.**class**);

}

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

SpringApplication.run(Application.**class**, args);

}

}

The next step is to update your build configuration such that your project produces a war file rather than a jar file. If you use Maven and spring-boot-starter-parent(which configures Maven’s war plugin for you), all you need to do is to modify pom.xml to change the packaging to war, as follows:

The final step in the process is to ensure that the embedded servlet container does not interfere with the servlet container to which the war file is deployed. To do so, you need to mark the embedded servlet container dependency as being provided.

If you use Maven, the following example marks the servlet container (Tomcat, in this case) as being provided:

<dependencies>

*<!-- … -->*

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

*<!-- … -->*

</dependencies>

If you use the [Spring Boot build tools](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins), marking the embedded servlet container dependency as provided produces an executable war file with the provided dependencies packaged in a lib-provided directory. This means that, in addition to being deployable to a servlet container, you can also run your application by using java -jaron the command line.