

18. Step 12 - Autowiring in Depth - by Name and @Primary

And

19. Step 13 - Autowiring in Depth - @Qualifier annotation

**package** com.mycom.springdemo.service;

**public** **interface** MyService {

**public** **int** myMethod();

}

**package** com.mycom.springdemo.serviceimpl;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

**public** **class** MyServiceImplOne **implements** MyService{

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

**return** 1;

}

}

**package** com.mycom.springdemo.serviceimpl;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

**public** **class** MyServiceImplTwo **implements** MyService {

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplTwo");

**return** 2;

}

}

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** MyController {

@Autowired

**public** MyService myService;

}

===================-We get below error=============================

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

APPLICATION FAILED TO START

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Description:

Field myService in com.mycom.springdemo.service.MyController required a single bean, but 2 were found:

- myServiceImplOne: defined in file [D:\learning\springWorkpace\springdemo\target\classes\com\mycom\springdemo\serviceimpl\MyServiceImplOne.class]

- myServiceImplTwo: defined in file [D:\learning\springWorkpace\springdemo\target\classes\com\mycom\springdemo\serviceimpl\MyServiceImplTwo.class]

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed

===================-We can solve this in many ways=============================

**Solution1 (Using @Primary)**

**package** com.mycom.springdemo.serviceimpl;

**import** org.springframework.context.annotation.Primary;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

@Primary

**public** **class** MyServiceImplOne **implements** MyService{

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

**return** 1;

}

}

**Solution2 (Using @Qualifier)**

**package** com.mycom.springdemo.serviceimpl;

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

@Qualifier("one")

**public** **class** MyServiceImplOne **implements** MyService{

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

**return** 1;

}

}

**import** org.springframework.beans.factory.annotation.Qualifier;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

**@Qualifier("two")**

**public** **class** MyServiceImplTwo **implements** MyService {

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplTwo");

**return** 2;

}

}

@RestController

**public** **class** MyController {

@Autowired

@Qualifier("one")

**public** MyService myService;

}

**Solution3 (Using variable name)**

**package** com.mycom.springdemo.serviceimpl;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

**public** **class** MyServiceImplOne **implements** MyService {

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

**return** 1;

}

}

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

**public** **class** MyServiceImplTwo **implements** MyService {

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplTwo");

**return** 2;

}

}

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** MyController {

@Autowired

**public** MyService myServiceImplOne;

}

20. Step 14 - Scope of a Bean - Prototype and Singleton

**singleton**

This scopes the bean definition to a single instance per Spring IoC container (default).

**prototype**

This scopes a single bean definition to have any number of object instances.

**request**

This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.

**session**

This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**global-session**

This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

**Singleton :** This scopes the bean definition to a single instance per Spring IoC container (default).

When we call same bean 2 times, do we get same bean or different bean? see below

**package** com.mycom.springdemo;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

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

**import** com.mycom.springdemo.serviceimpl.MyServiceImplOne;

@SpringBootApplication

**public** **class** SpringdemoApplication {

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

ApplicationContext ctx = SpringApplication.*run*(SpringdemoApplication.**class**, args);

MyServiceImplOne instaceOne=ctx.getBean(MyServiceImplOne.**class**);

MyServiceImplOne instaceTwo=ctx.getBean(MyServiceImplOne.**class**);

System.***out***.println(instaceOne);

System.***out***.println(instaceTwo);

}

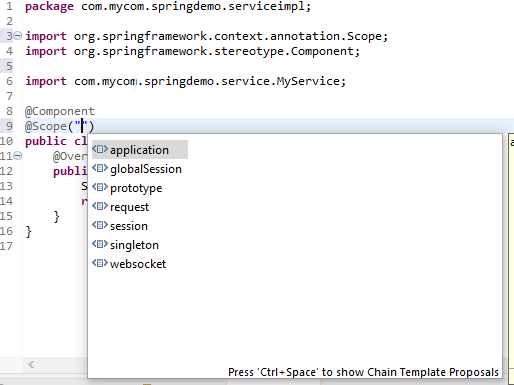
}

Output

com.mycom.springdemo.serviceimpl.MyServiceImplOne@198a0818

[com.mycom.springdemo.serviceimpl.MyServiceImplOne@198a0818](mailto:com.mycom.springdemo.serviceimpl.MyServiceImplOne@198a0818)

Hashcode is same (we are getting same bean, same instance). This type is called Singleton.



**import** org.springframework.context.annotation.Scope;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

@Scope("prototype")

**public** **class** MyServiceImplOne **implements** MyService {

@Override

**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

**return** 1;

}

}

Output :

com.mycom.springdemo.serviceimpl.MyServiceImplOne@5d2f5217

[com.mycom.springdemo.serviceimpl.MyServiceImplOne@738373cc](mailto:com.mycom.springdemo.serviceimpl.MyServiceImplOne@738373cc)

Hashcode is different (we are getting different bean, different instance). This type is called Prototype.

Instead of hardcoding prototype, we can use like below

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

**import** org.springframework.context.annotation.Scope;

**import** org.springframework.stereotype.Component;

**import** com.mycom.springdemo.service.MyService;

@Component

@Scope(value = ConfigurableBeanFactory.***SCOPE\_PROTOTYPE***)

**public** **class** MyServiceImplOne **implements** MyService {

@Override

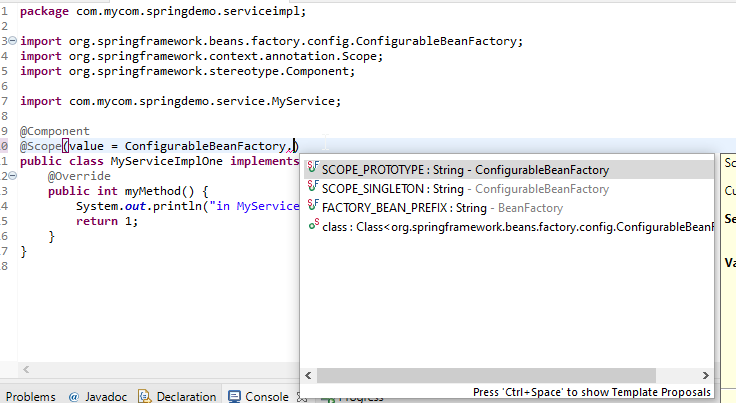
**public** **int** myMethod() {

System.***out***.println("in MyServiceImplOne");

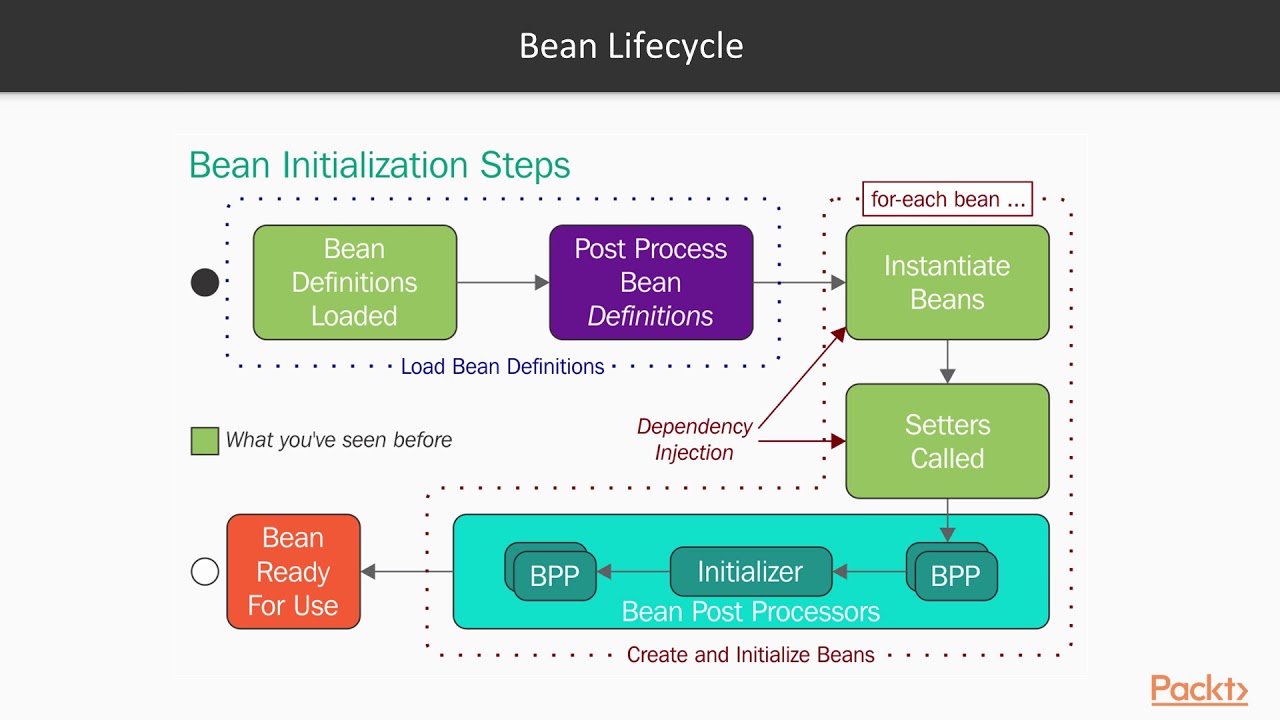
**return** 1;

}

}



24. Step 17 - Lifecycle of a Bean - @PostConstruct and  
@PreDestroy



**package** com.mycom.springdemo.service;

**import** javax.annotation.PostConstruct;

**import** javax.annotation.PreDestroy;

**import** org.slf4j.Logger;

**import** org.slf4j.LoggerFactory;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** MyController {

Logger logger =LoggerFactory.*getLogger*(**this**.getClass());

@Autowired

**public** MyService myServiceImplOne;

@PostConstruct

**public** **void** anyName()

{

logger.info("I am called after all dependant beans are loaded");

}

@PreDestroy

**public** **void** anyName2()

{

logger.info("I am called when server stops");

}

}

25. Step 18 - Container and Dependency Injection (CDI) -  
@Named, @Inject

This is similar to JPA and Hibernate (JPA is a Java specification and Hibernate is implemention)

Javax.inject.. is an API and springframework can implement it.

**JSRs: Java Specification Requests**

**JSR 330: Dependency Injection for Java**

Since Spring 3.0, Spring supports for the standard [JSR 330: Dependency Injection for Java](https://www.jcp.org/en/jsr/detail?id=330). In Spring 3 application, you can use standard

1. @Inject instead of Spring’s @Autowired to inject a bean.
2. @Named instead of Spring’s @Component to declare a bean.

Those JSR-330 standard annotations are scanned and retrieved the same way as Spring annotations, the integration just happened automatically, as long as the following jar in your classpath.

pom.xml

<dependency>

<groupId>javax.inject</groupId>

<artifactId>javax.inject</artifactId>

<version>1</version>

</dependency>

|  |  |
| --- | --- |
| Spring Annotations | JSR-330 Annotations |
| import org.springframework.stereotype.Repository;  @Repository public class CustomerDAO  {  public void save() {  System.out.println("CustomerDAO save method...");  }  } | import javax.inject.Named;  @Named public class CustomerDAO  {    public void save() {  System.out.println("CustomerDAO save method...");  }  } |
| import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service;  @Service public class CustomerService  {  @Autowired  CustomerDAO customerDAO;   public void save() {    System.out.println("CustomerService save method...");  customerDAO.save();    }   } | import javax.inject.Inject; import javax.inject.Named;   @Named public class CustomerService  {  @Inject  CustomerDAO customerDAO;   public void save() {    System.out.println("CustomerService save method...");  customerDAO.save();    }   } |
| @Scope is for setting the scope, @Qualfier (similar to what we learned)  4. JSR-330 Limitations There are some limitations on JSR-330 if compare to Spring :  @Inject has no “required” attribute to make sure the bean is injected successful. In Spring container, JSR-330 has scope singleton by default, but you can use Spring’s @Scope to define others. No equivalent to Spring’s @Value, @Required or @Lazy. | |
| 5. Go for JSR-330 In fact, Spring’s annotations are more powerful, but only available on Spring framework. The JSR-330 is a standard spec, and it’s supported on all J2ee environment that follow the JSR-330 spec.  For new or migration project, it’s always recommended to use JSR-330 annotations, and remember, it works on Spring 3 as well. | |

-======================================================================

@ SpringBootApplication is same as

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters =

{ @Filter(type = FilterType.***CUSTOM***, classes = TypeExcludeFilter.**class**),

@Filter(type = FilterType.***CUSTOM***, classes = AutoConfigurationExcludeFilter.**class**)

})

**public** **@interface** SpringBootApplication {

-==============================================================================

32. Step 24 - IOC Container vs Application Context vs Bean  
Factory

Tightly coupled

**public** **interface** MyService {

**public** **int** myMethod();

}

**public** **class** MyServiceImplOne **implements** MyService {

}

**public** **class** MyController {

MyService service = new MyServiceImplOne();

}

Loosely coupled

**public** **interface** MyService {

**public** **int** myMethod();

}

**public** **class** MyServiceImplOne **implements** MyService {

}

**public** **class** MyController {

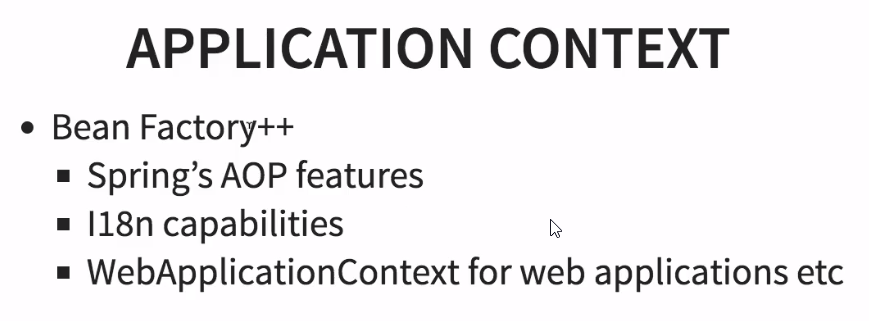
@Autowired

**public** MyService myServiceImplOne;

}

The control of creating the service is with Bean itself. Spring is creating the instance for us using Spring container. The control moves out of bean to create the service, this is called inversion of control.

The Ioc Container is responsible to create the objects

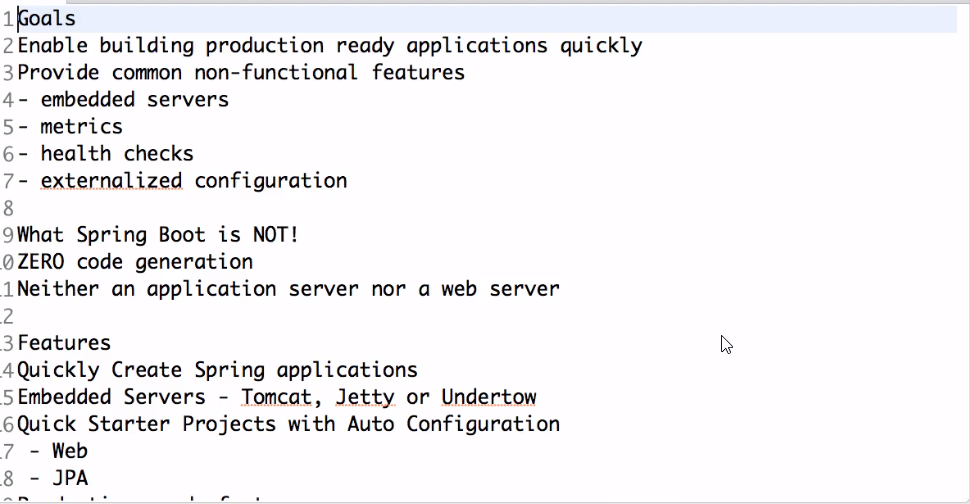


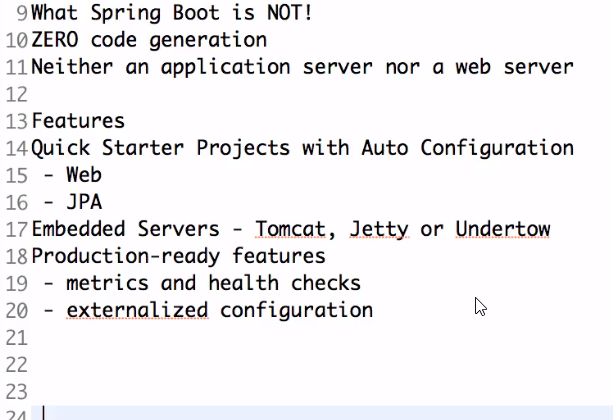
99% of time spring suggests to use ApplicationContext instead of BeanFactory

33. Step 25 - @Component vs @Service vs @Repository vs  
@Controller



Springboot advantages





Springboot actuator

management:

endpoints:

enabled-by-default: true

web.exposure:

include: "\*"

jmx:

exposure:

include: "\*"

endpoint:

info:

enabled: true

beans:

enabled: true

prometheus:

enabled: true

metrics:

enabled: true

<http://localhost:8080/actuator>

<http://localhost:8080/actuator/env>

<http://localhost:8080/actuator/health>

<http://localhost:8080/actuator/info>

<http://localhost:8080/actuator/loggers>

<http://localhost:8080/actuator/beans>