https://docs.spring.io/spring-framework/docs/6.0.6/reference/html/

Spring ecosystem

By Vo Van Hai
vovanhai@iuh.edu.vn

1

Introduction

- The Spring ecosystem provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any deployment platform.
- A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

2

Spring Projects

- https://spring.io/projects
- Consists of 23 projects (March 2023):

Spring Boot
Spring Framework
Spring Data
Spring Cloud
Spring Cloud Data Flow
Spring Security
Spring Authorization Server
Spring for GraphQL
Spring Session
Spring Integration
Spring HATEOAS
Spring REST Docs
Spring Batch

Spring Batch
Spring AMQP
Spring CredHub
Spring Flo
Spring for Apache Kafka
Spring LDAP
Spring Shell
Spring Statemachine
Spring Vault
Spring Web Flow
Spring Web Services

3

Spring History

- Rod Johnson
  - "Expert One-on-One J2EE Design and Development," Wrox. ISBN 0-7645-4385-7, 2002
  - "Expert One-on-One J2EE Development without EJB," Wrox. ISBN 0-7645-5831-5, 2024

Version	Date	Notes
0.9	2003	
1.0	March 24, 2004	First production release.
2.0	2006	
3.0	2009	
4.0	2013	
5.0	2017	
6.0	November 16, 2022	

4

Spring Framework Overview

- Although Spring is an ecosystem, the heart of all other projects is based on the Spring Framework.
- Spring makes it easy to create Java enterprise applications. It provides everything you need to embrace the Java language in an enterprise environment, and with the flexibility to create many kinds of architectures depending on an application's needs.
- As of Spring Framework 6.0, Spring requires Java 17+.

"Spring": different things in different contexts

5

Spring Framework components

spring

Overview	History, Design Philosophy, Feedback, Getting Started.
Core	IoC Container, Events, Resources, i18n, Validation, Data Binding, Type Conversion, SpEL, AOP, AOT.
Testing	Mock Objects, TestContext Framework, Spring MVC Test, WebTestClient.
Data Access	Transactions, DAO Support, JDBC, R2DBC, O/R Mapping, XML Marshalling.
Web Servlet	Spring MVC, WebSocket, SockJS, STOMP Messaging.
Web Reactive	Spring WebFlux, WebClient, WebSocket, RSocket.
Integration	REST Clients, JMS, JCA, JMX, Email, Tasks, Scheduling, Caching, Observability.
Languages	Kotlin, Groovy, Dynamic Languages.
Appendix	Spring properties.
Wiki	What's New, Upgrade Notes, Supported Versions, additional cross-version information.

https://docs.spring.io/spring-framework/docs/6.0.6/reference/html/

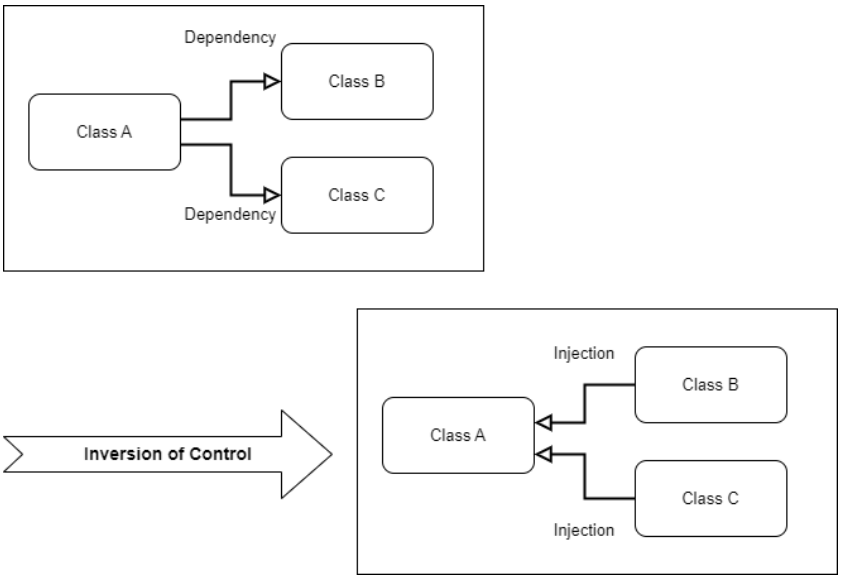
6

Spring Framework Core

- Is the core of the Spring ecosystem.
- Based on it, other projects are developed.
- Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container.

7

Inversion of Control (IoC)



8

Dependency injection (DI)

- Dependency Inject is a technique (a design pattern) that removes hard-code dependencies and makes your application easier to extend and maintain.

9

Without DI

```
package vvh.ioc.example;

public class ICEngine {
    private float cylinder_capacity;
    private String type;

    public void start() {
        System.out.println("Engine is started");
    }
}
```

Tightly-coupled

```
package vvh.ioc.example;
public class Car {

    private ICEngine engine;

    public void start(){
        engine =new Engine();
        engine.start();
    }
}
```

What will happen when the Engine class is changed?

```
package vvh.ioc.example;

public class MyApp {
    public static void main(String[] args) {
        Car c =new Car();
        c.start();
    }
}
```

```
package vvh.ioc.example;

public class ICEngine {
    private float cylinder_capacity;
    private String type;
    public ICEngine(float cylinder_capacity, String type) {
        this.cylinder_capacity = cylinder_capacity;
        this.type = type;
    }

    public void start() {
        System.out.println("Engine is started");
    }
}
```

10

With DI

```
package vvh.ioc.example;
public class ICEngine {
    private float cylinder_capacity;
    private String type;
    public void start() {
        System.out.println("Engine is started");
    }
}
```

```
package vvh.ioc.example;
public class Car {

    private ICEngine engine;

    public Car(ICEngine engine) {
        this.engine = engine;
    }
    public void start(){
        engine.start();
    }
}
```

Inject Engine object to Car class

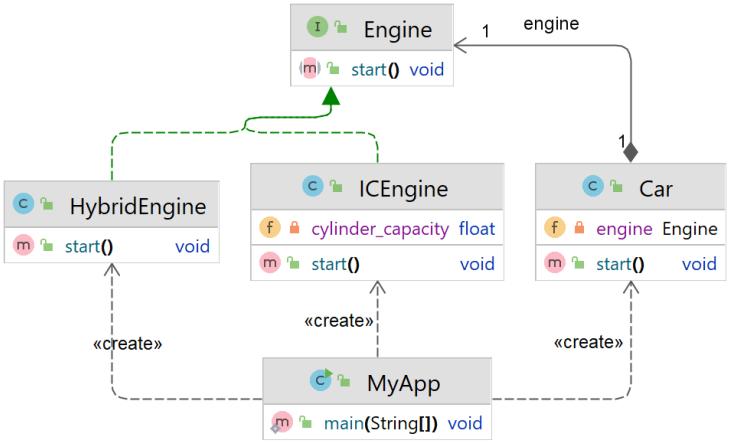
```
package vvh.ioc.example;

public class MyApp {
    public static void main(String[] args) {
        ICEngine engine = new ICEngine();
        Car c =new Car(engine);
        c.start();
    }
}
```

Changes to the Engine class will not affect to Car class.

What will happen when we have another engine type? (E.g., Hybrid Engine, Electricity Engine, ...)

11

Example



12

Injection Types

- Constructor Injection
- Property Injection
- Method Injection

13

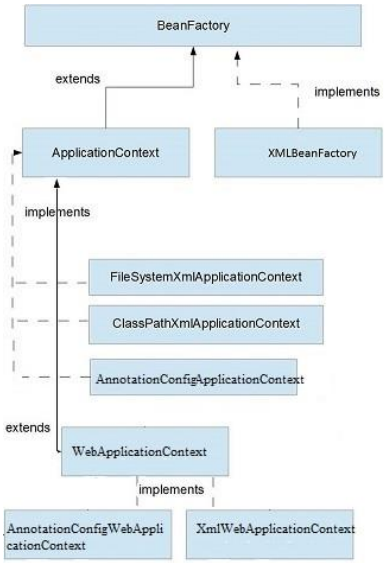
The Spring's IOC Container

- Remind: IoC is also known as dependency injection (DI) - a process whereby objects define their dependencies through constructor arguments, arguments to a factory method, or properties.
- The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container.
- The BeanFactory object provides the configuration framework and basic functionality, and the ApplicationContext object adds more enterprise-specific functionality.

14

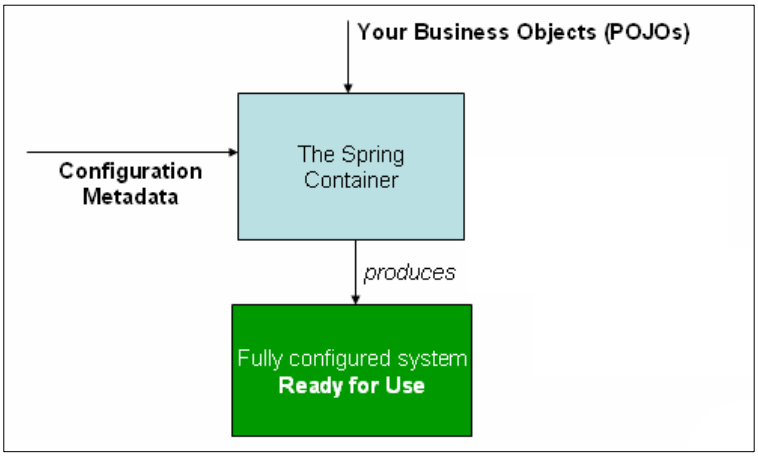
The Spring's IOC Container

- In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans.
- A bean is an object that is instantiated, assembled, and managed by a Spring IoC container.
- Beans, and the dependencies among them, are reflected in the configuration metadata used by a container.



15

The Spring's IoC container



The primary job of the ApplicationContext is to manage beans.

16

Spring Bean

- In Spring, a bean is an object that the Spring container instantiates, assembles, and manages.
- Any Java POJO class can be a Spring Bean if configured and initialized through the container by providing configuration information.
- We should define beans for service layer objects, data access objects (DAOs), presentation objects, infrastructure objects such as Hibernate SessionFactories, JMS Queues, and so forth.

17

Spring Bean Scope

- **Singleton**: (*default*) Only one instance of the bean will be created per container. This is the default scope for spring bean.
- **Prototype**: An instance of the bean will be created for each request.

WEB-CONTEXT

- ***Request***: same as prototype scope, but for web application, an instance of bean will be created for each HTTP request.
- ***Session***: Each bean instance will be created for each HTTP Session
- ***Global-Session***: Used to create global session beans for Portlet applications.

18

Configuring Beans in the Container

- The primary job of the ApplicationContext is to manage beans → application must provide the bean configuration to the ApplicationContext container.
- Type of configurations:
  - XML-Based Configuration
  - Java-Based Configuration
  - Annotation-Based Configuration

Gradle

```
implementation 'org.springframework:spring-context:6.0.6'
```

Maven

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>6.0.6</version>
</dependency>
```

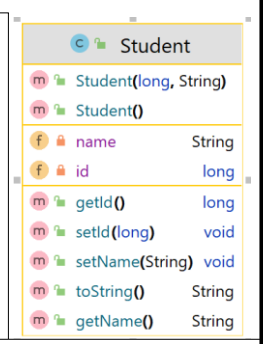
19

Configuring Beans in the Container

XML-Based Configuration

Beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-
beans.xsd">

   <bean id="st1" class="org.example.Student">
      <property name="id" value="001"/>
      <property name="name" value="than thi det"/>
   </bean>

</beans>
```



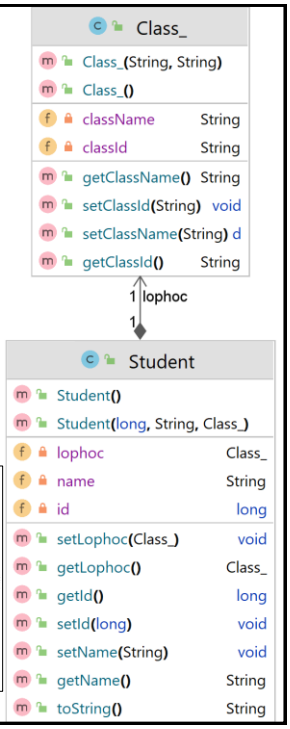

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class Main {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
      Student st1 = context.getBean("st1", Student.class);
      System.out.println(st1);
   }
}
```

20

Object Injection

```
<bean id="lop" class="org.example.beans.Class_">
    <property name="classId" value="DHTH15A"/>
    <property name="className" value="Lop DHTH15A"/>
</bean>

<bean id="st2" class="org.example.beans.Student">
    <property name="id" value="001"/>
    <property name="name" value="than thi det"/>
    <property name="lophoc" ref="lop"/>
</bean>
```

```
public class Main {
    no usages
    public static void main(String[] args) {
        ApplicationContext context =
                new ClassPathXmlApplicationContext( configLocation: "beans.xml");
        Student st1 = context.getBean( name: "st2", Student.class);
        System.out.println(st1);
    }
}
```



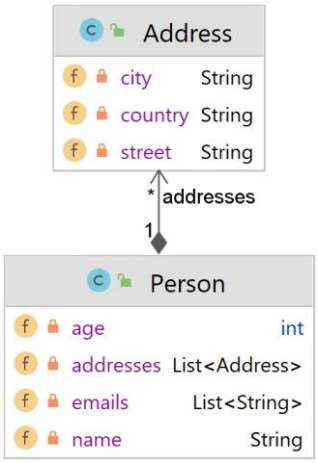
21

Object Injection (cont.)

```
<bean id="lop" class="org.example.xmlbased.Class_">
    <property name="classId" value="DHTH15A"/>
    <property name="className" value="Lop Dai Hoc Tin Hoc 15"/>
</bean>

<!--Inject by setter-->
<bean id="st2" class="org.example.xmlbased.Student">
    <property name="id" value="001"/>
    <property name="name" value="than thi det"/>
    <property name="lophoc" ref="lop"/>
</bean>

<!--Inject by constructor-->
<bean id="st3" class="org.example.xmlbased.Student">
    <constructor-arg name="id" value="002"/>
    <constructor-arg name="name" value="Tran Thi Men"/>
    <constructor-arg name="lophoc" ref="lop"/>
</bean>
```

22

Collection Injection

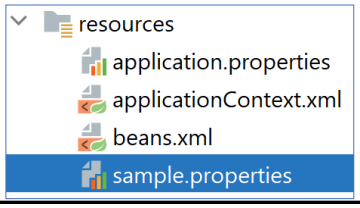


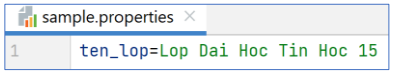

```xml
<bean id="add1" class="org.example.xmlbased.Address">
    <property name="street" value="Nguyen Van Bao 12"/>
    <property name="city" value="Saigon"/>
    <property name="country" value="Vietnam"/>
</bean>

<bean id="add2" class="org.example.xmlbased.Address">
    <property name="street" value="Nguyen Thai Son 13"/>
    <property name="city" value="Saigon"/>
    <property name="country" value="Vietnam"/>
</bean>

<bean id="person" class="org.example.xmlbased.Person">
    <property name="name" value="Nguyen Van Teo"/>
    <property name="addresses">
        <list>
            <ref bean="add1"/>
            <ref bean="add2"/>
        </list>
    </property>
    <property name="emails">
        <list>
            <value>teo@gmail.com</value>
            <value>teo@hotmail.com</value>
        </list>
    </property>
</bean>
```

23

Literal Values Injection

```xml
<bean id="myProperties"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:sample.properties</value>
        </list>
    </property>
    <property name="ignoreResourceNotFound" value="true" />
    <property name="ignoreUnresolvablePlaceholders" value="true" />
</bean>

<bean id="lop" class="org.example.xmlbased.Class_">
    <property name="classId" value="DHTH15A"/>
    <property name="className" value="${ten_lop}"/>
</bean>
```





24

Spring's Auto-wiring

- Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and string values. It works with reference only.

No.	Mode	Description
1	no	The default *autowiring* mode. It means no auto-wiring by *default*.
2	byName	The *byName* mode injects the object dependency according to name of the bean. In such case, property name and bean name must be same. It internally calls setter method.
3	byType	The *byType* mode injects the object dependency according to type. So property name and bean name can be different. It internally calls setter method.
4	constructor	The *constructor* mode injects the dependency by calling the constructor of the class. It calls the constructor having large number of parameters.
~~5~~	~~autodetect~~	~~deprecated since Spring 3.~~

Table: Autowiring Modes

25

Spring's Auto-wiring

```
<bean id="faculty" class="org.example.autowiring.Faculty">
    <property name="f_name" value="Faculty of Information Technology"/>
</bean>

<bean id="dept1" class="org.example.autowiring.Department" autowire="no">
    <property name="d_name" value="SE"/>
    <property name="faculty" ref="faculty"/>
</bean>
```

explicit

can be omitted

```
<bean id="dept2" class="org.example.autowiring.Department" autowire="byName">
    <property name="d_name" value="SE"/>
</bean>

<bean id="faculty" class="org.example.autowiring.Faculty"
    <property name="f_name" value="Faculty of Information Technology"/>
</bean>
```

```
public class Department {
    private String d_name;

    private Faculty faculty;
}
```

Find bean with the id "faculty"

26

Spring's Auto-wiring

```
<bean id="faculty" class="org.example.autowiring.Faculty">
    <property name="f_name" value="Faculty of Information Technology"/>
</bean>

<bean id="dept3" class="org.example.autowiring.Department" autowire="byType">
    <property name="d_name" value="SE"/>
</bean>
```

Find bean with the type "Faculty"

```
public class Department {
    private String d_name;
    private Faculty faculty;
```

```
<bean id="faculty" class="org.example.autowiring.Faculty">
    <property name="f_name" value="Faculty of Information Technology"/>
</bean>

<bean id="dept4" class="org.example.autowiring.Department"
                                        autowire="constructor">
    <property name="d_name" value="IS"/>
</bean>
```

Find bean with the type "Faculty"

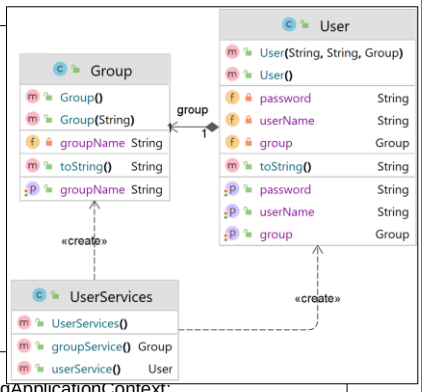
```
public class Department {
    private String d_name;
    private Faculty faculty;

    public Department() {
    }
    public Department(Faculty faculty) {
        this.faculty = faculty;
    }
```

27

Configuring Beans in the Container

Java-Based Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class UserServices {
    @Bean
    public Group groupService() {
        return new Group("Admin Group");
    }
    @Bean
    public User userService() {
        return new User("teo", "123", groupService());
    }
}
```




```
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
public class Main {
    public static void main(String[] args) {
        ApplicationContext ctx =
                new AnnotationConfigApplicationContext(UserServices.class);
        User u = ctx.getBean(User.class);
        System.out.println(u);
    }
}
```

28

Configuring Beans in the Container

Annotation-Based Configuration

applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>
    <context:component-scan base-package="org.example.annotationbased"/>

</beans>
```

<context:annotation-config/> only looks for annotations on beans in the same application context in which it is defined.

You can also use @Configuration annotation with the same purpose

```java
@Configuration
@ComponentScan("your.package")
public class AppConfig {
    //no-op
}
```

29

The @Autowired annotation

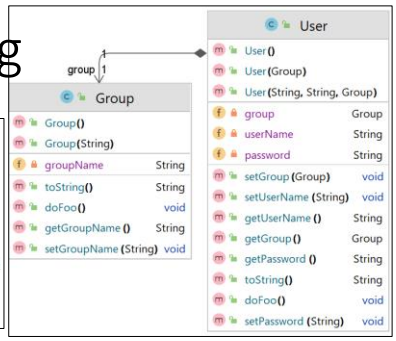
- The *@Autowired* annotation allows Spring to resolve and inject collaborating beans into another bean.

30

@Autowired xml config

```
<bean id="user" class="org.example.annotationbased.User">
    <property name="userName" value="vovanhai"/>
    <property name="password" value="s3cr3t"/>
</bean>

<bean id="group" class="org.example.annotationbased.Group">
    <property name="groupName" value="Administrator"/>
</bean>
```

applicationContext.xml



Field

```
@Autowired
private Group group;
```

Constructor

```
@Autowired
public User(Group group) {
    this.group = group;
}
```

Setter

```
@Autowired
public void setGroup(Group group) {
    this.group = group;
}
```

```
public static void main(String[] args) {
    ApplicationContext ctx =
            new ClassPathXmlApplicationContext("applicationContext.xml");
    User us = (User) ctx.getBean("user");
    us.doFoo();
    System.out.println(us);
}
```

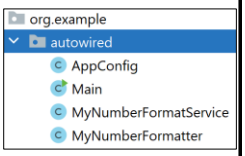
31

@Autowired Java-based config

```
@Component
public class MyNumberFormatter {
    public String format(double number) {
        return "My Number Format - " + number;
    }
}
```

```
@Configuration
@ComponentScan("org.example.autowired")
public class AppConfig {
    //no-op
}
```

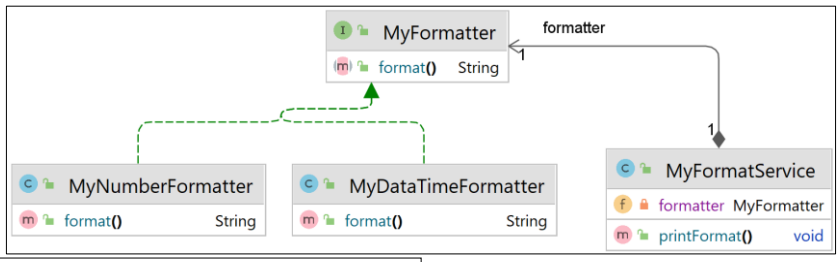
```
@Component
public class MyNumberFormatService {
    private final MyNumberFormatter myNumberFormatter;

    //@Autowired - not required. SpringFX is smart enough to have known.
    public MyNumberFormatService(MyNumberFormatter myNumberFormatter) {
        this.myNumberFormatter = myNumberFormatter;
    }
    public void printFormat(double number) {
        System.out.println(myNumberFormatter.format(number));
    }
}
```




```
public static void main(String[] args) {
    ApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppConfig.class);
    MyNumberFormatService svc = ctx.getBean(MyNumberFormatService.class);
    svc.printFormat( number: 100d);
}
```

32

@Autowire Disambiguation




```
@Component
public class MyFormatService {

    private MyFormatter formatter;

    public MyFormatService(MyFormatter formatter) {
        this.formatter = formatter;
    }

    public void printFormat() {
        System.out.println(formatter.format());
    }
}
```

Error

```
Caused by: org.springframework.beans.factory
 .NoUniqueBeanDefinitionException: No qualifying bean of
  type 'org.example.autowired.MyFormatter' available:
 expected single matching bean but found 2:
 myDataTimeFormatter,myNumberFormatter
```

```
public static void main(String[] args) {
    ApplicationContext ctx =
            new AnnotationConfigApplicationContext(AppConfig.class);
    MyFormatService svc = ctx.getBean(MyFormatService.class);
    svc.printFormat();
}
```

33

Disambiguation solution: @Qualifier

```
@Component("myNumberFormatter") //bean-name
public class MyNumberFormatter implements MyFormatter{
    @Override
    public String format() {
        return "My Number Format";
    }
}
```

```
@Component("myDataTimeFormatter")
public class MyDataTimeFormatter implements MyFormatter{
    @Override
    public String format() {
        return "My Date Time Formatter";
    }
}
```

```
@Component
public class MyFormatService {
    2 usages
    private MyFormatter formatter;
    no usages
    public MyFormatService(@Qualifier("myDataTimeFormatter") MyFormatter formatter) {
        this.formatter = formatter;
    }
    1 usage
    public void printFormat() {...}
}
```

When there are multiple beans of the same type → use @Qualifier to avoid ambiguity.
Spring uses the bean's name as a default qualifier value.

34

Disambiguation solution: @Primary

- @Primary indicates that a particular bean should be given preference when multiple beans are candidates to be autowired to a single-valued dependency.

```
@Component
@Primary
public class MyNumberFormatter implements MyFormatter{
    @Override
    public String format() {
        return "My Number Format";
    }
}
```

```
@Component
public class MyDataTimeFormatter implements MyFormatter{
    @Override
    public String format() {
        return "My Date Time Formatter";
    }
}
```

```
@Component
public class MyFormatService {
    2 usages
    private MyFormatter formatter;
    no usages
    public MyFormatService(MyFormatter formatter)
        this.formatter = formatter;
    }
    1 usage
    public void printFormat() {...}
}
```

Automatically select MyNumberFormater class

35

Inject resources with @Value

```
@Configuration
@ComponentScan("org.example.resources")
@PropertySource("classpath:application.properties")
public class AppConfig {
   @Bean
   public ClientBean clientBean() {
      return new ClientBean();
   }
}
```

```
public class ClientBean {
   @Value("classpath:beans.xml")
   private Resource myResource;

   @Value("${foo.permission}")
   private String permission;

   public void doSomething() throws IOException {
      File file = myResource.getFile();
      String s = new
String(Files.readAllBytes(file.toPath()));
      System.out.println(s);
      System.out.println(permission);
   }
}
```

application.properties

```
foo.permission=this is sample permission
```

```
public class Main {
   public static void main(String[] args) throws IOException {
      AnnotationConfigApplicationContext context =
            new
AnnotationConfigApplicationContext(AppConfig.class);
      ClientBean bean = context.getBean(ClientBean.class);
      bean.doSomething();
   }
}
```

```
@Configuration
@ComponentScan("org.example.resources")
@PropertySource("classpath:application.properties")
public class AppConfig {
   @Bean
   public ClientBean clientBean() {
      return new ClientBean();
   }
}
```

37

38