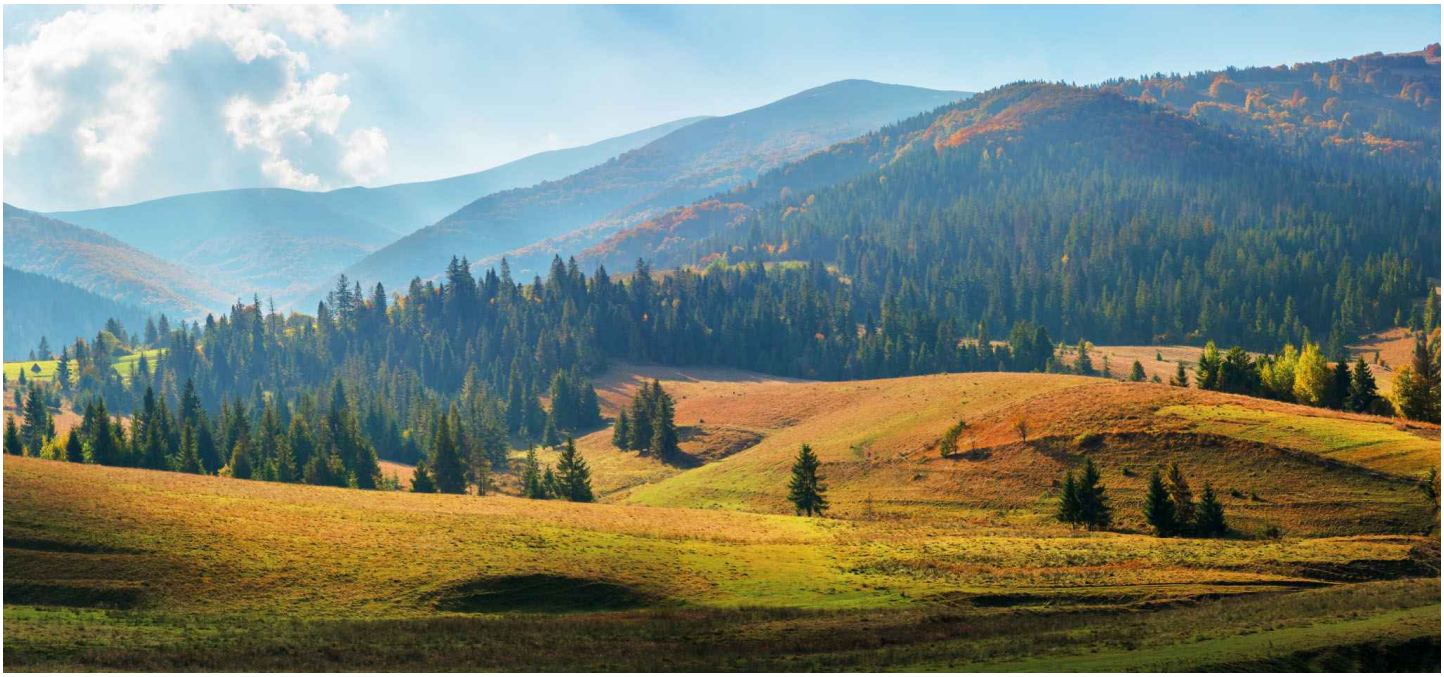

4-Spring Core Implementation

Author: Vincent Lau

Learning Objectives

> Declare Spring beans with XML file and Annotations
>
> Use the correct implementation of ApplicationContext under different circumstances
>
> Construct an application using various forms of Dependency Injection
>
> Explain which injection type is preferred under different circumstances
>
> Describe the ambiguity problem and its solutions

Overview

- There are three options for how dependencies can be injected into a bean:
  - **Constructor Injection**

    - **Setter Injection**
    - **Field Injection**
- This chapter will cover how to implement each injection type, as well as which one to use under different circumstances.

Setting up the Project

Maven Dependencies

- To enable ***Spring Core*** with the use of *Spring Beans* and *ApplicationContext* in our project, we need to include these dependencies in our *pom.xml*:

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>5.3.16</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>5.3.16</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.3.16</version>
</dependency>
```

Domain Model

- Before wiring the beans together, suppose we have classes like these:

```
public class FlowerOrder {
    private final String flowerName;
    private final int size;
```

```
    private final String customerName;
    // constructors and setters
}

public class InventoryChecker {
    private final Map<String, Integer> store = new HashMap<>();

    public boolean hasInventory(String flowerName, int size) {
        if (!store.containsKey(flowerName)) {
            return false;
        }
        return store.get(flowerName) >= size;
    }

                // other code
}
```

Autowiring

- Starting with Spring 2.5, the framework introduced annotations-driven ***Dependency Injection***. The main annotation of this feature is `@Autowired` .**It allows Spring to resolve and inject collaborating beans into our bean.**
- The Spring framework enables **automatic dependency injection**. In other words, by declaring all the bean dependencies in a Spring configuration file, Spring container can **autowire relationships between collaborating beans**. This is called ***Spring bean autowiring***.
- We can use **autowiring** on **fields, setters, and constructors**.
- By default, Spring resolves `@Autowired` entries **by type**. If **more than one bean of the same type is available in the container**, the framework will throw a fatal exception.
    - To resolve this conflict, we need to tell Spring explicitly which bean we want to inject. This will be discussed later in the chapter.

Constructor Injection

Configuration via XML (Traditional Way - Spring Application)

- We inject *InventoryChecker* into *FlowerOrderingService* via the constructor to perform some inventory checking before processing the order:

```java
@Service // springboot
public class FlowerOrderingService {
    private InventoryChecker inventoryChecker;

    public FlowerOrderingService(InventoryChecker inventoryChecker) {
        // null check
        this.inventoryChecker = inventoryChecker;
    }

    public void process(FlowerOrder order) {
        if (!inventoryChecker.hasInventory(order.getFlowerName(), order.getSize())) {
            throw new RuntimeException("Insufficient inventory");
        }

        // do something else

        System.out.println("Process completed");
    }
}
```

- The ***applicationContext.xml*** looks something like this, with constructor injection: constructor-arg:

```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:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
               http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="inventoryChecker" class="..." />

    <bean id="flowerOrderingService" class="...">
        <constructor-arg name="inventoryChecker" ref="inventoryChecker" />
    </bean>

</beans>
```

- Now let's test our code:

```java
public class CITest {
    public static void main(String[] args) {
        testCIWithXMLFile();
    }

    private static void testCIWithXMLFile() {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        FlowerOrderingService service = (FlowerOrderingService) ctx.getBean("flowerOrderingService");
        service.process(new FlowerOrder("Daisy", 3, "John Doe"));
    }
}

// Output: Process completed
```

Configuration via Annotations (Autowiring - Springboot Application)

- We add the `@Autowired` annotation to the constructor of *FlowerOrderingService* to inject the *InventoryChecker* dependency:
- Not necessary to add **@Autowired**, if only one constructor is defined. But it is recommended to add for the readability.

```java
// @Service
// we can annotate as Component, if we do not config FlowerOrderingService as Bean in BeanConfig.java
public class FlowerOrderingService {

    private InventoryChecker inventoryChecker;

    @Autowired
    public FlowerOrderingService(InventoryChecker inventoryChecker) {
        this.inventoryChecker = inventoryChecker;
    }

    // other code
}
```

- Instead of using the XML bean definition file, we create a ***BeanConfig* class** to define the *Spring beans, by **@Configuration** annotation*:
- To configure which packages to scan for classes with annotation configuration, we can use the `@ComponentScan` annotation:

```java
@Configuration
// We don't have to @ComponentScan here, if @SpringbootApplication is in root path
@ComponentScan(basePackages = "com.bootcamp.demo")
public class BeanConfig {
    @Bean(name = "inventoryChecker")
    public InventoryChecker createInventoryChecker() {
        return new InventoryChecker();
    }

    @Bean(name = "flowerOrderingService")
    public FlowerOrderingService createService(InventoryChecker inventoryChecker) {
        return new FlowerOrderingService(inventoryChecker);
    //  return new FlowerOrderingService(new InventoryChecker);
    }
}
```

- For test code, instead of using `ClassPathXmlApplicationContext`, this time we use `AnnotationConfigApplicationContext` to create the *Application Context.* A similar result should be yielded when running the test class:

```java
public class CITest {
    public static void main(String[] args) {
        testCIWithAnnotations();
    }

    private static void testCIWithAnnotations() {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(BeanConfig.class);
        FlowerOrderingService service = (FlowerOrderingService) ctx.getBean("flowerOrderingService");
        service.process(new FlowerOrder("Daisy", 3, "John Doe"));
    }
}
```

Setter Injection

- In setter-based injection, **the required dependencies are set using the setter methods**.

Configuration via XML

- Because we're implementing ***`Setter Injection`*** , this time we use the `<property>` tag to define the *InventoryChecker* dependency for *FlowerOrderingService* in *applicationContext.xml*, instead of using the `<constructor-arg>` tag.

```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-3.0.xsd">

    <bean id="inventoryChecker" class="..."/>

    <bean id="flowerOrderingService" class="...">
        <property name="inventoryChecker" ref="inventoryChecker"/>
    </bean>

</beans>
```

- The *FlowerOrderingService* class then becomes:

```java
public class FlowerOrderingService {
    private InventoryChecker inventoryChecker;

    public void setInventoryChecker(InventoryChecker inventoryChecker) {
        this.inventoryChecker = inventoryChecker;
    }
    // other code
}
```

- Now let's test our code. It should yield the same result as when using *Constructor Injection*:

```java
public class SITest {
    public static void main(String[] args) {
        testSIWithXMLFile();
    }

    private static void testSIWithXMLFile() {
```

```
        ApplicationContext ctx = new
ClassPathXmlApplicationContext("applicationContext.xml");
        FlowerOrderingService service = (FlowerOrderingService)
ctx.getBean("flowerOrderingService");
        service.process(new FlowerOrder("Daisy", 3, "John Doe"));
    }
}

// Output: Process completed
```

Configuration via Annotations (Autowiring)

- We have to annotate the setter method with the `@Autowired` annotation.
- This time, we use the `@Service` and `@Component` annotation to simplify the code, hence avoiding the need to define the beans with a separate *BeanConfig* class.
- Note that this time we pass the *base package name to* `AnnotationConfigApplicationContext`, instead of passing the *BeanConfig* class type, because there is no *BeanConfig* class anymore.

```
@Component
public class InventoryChecker {
                // other code
}

@Service
public class FlowerOrderingService {
//    @Autowired
    private InventoryChecker inventoryChecker;

    @Autowired
    public void setInventoryChecker(InventoryChecker inventoryChecker) {
        this.inventoryChecker = inventoryChecker;
    }
    // other code
}

public class SITest {
    public static void main(String[] args) {
        testSIWithAnnotations();
    }

    private static void testSIWithAnnotations() {
```

```
24          ApplicationContext ctx = new
    AnnotationConfigApplicationContext("com.bootcamp.springdemo.SIExample");
25          FlowerOrderingService service = (FlowerOrderingService)
    ctx.getBean("flowerOrderingService");
26          service.process(new FlowerOrder("Daisy", 3, "John Doe"));
27      }
28  }
```

Field Injection

- With field-based injection, Spring **assigns the required dependencies directly to the fields** on annotated with `@Autowired` .
- This will have the same effect as *Constructor Injection* and *Setter Injection.*

```
1   @Service
2   public class FlowerOrderingService {
3
4       @Autowired
5       private InventoryChecker inventoryChecker;
6
7       public void process(FlowerOrder order) {
8           if (inventoryChecker.hasInventory(order.getFlowerName(),
    order.getSize())) {
9               throw new RuntimeException("Insufficient inventory");
10          }
11
12          // do something else
13
14          System.out.println("Process completed");
15      }
16  }
```

Which One to Use?

- For mandatory dependencies or when aiming for immutability, use ***Constructor Injection***.
- For optional or changeable dependencies, use ***Setter Injection***. But it is risky, the dependency can be overwritten in runtime.

- ***Field Injection*** is the most common use in all spring boot microservice development, because most of the classes (controller/ service/ repository/ configuration) with DI are created as Bean. The Dependency check is well checked **during the "test" cycle (mvn clean test)**. Besides, field injection is the most convenient for developers.

Challenge

```
// @Component // if this is not spring context bean
class MyComponent {

 // answer -> @Autowired
  MyCollaborator collaborator;

   public void myBusinessMethod() {
     collaborator.doSomething(); // NPE
   }
}
// main
MyComponent component = new MyComponent(); // when this is not a spring context bean
component.myBusinessMethod(); // -> NullPointerException, why? design issue? how to solve?
```

Ambiguity Issue

- If for some reason, multiple beans of the same type have been defined (e.g. different implementations are available for the same interface):

```
@Configuration
public class BeanConfig {
    @Bean
    public InventoryChecker createInventoryChecker() {
        return new InventoryChecker();
    }

    @Bean
    public InventoryChecker createAnotherInventoryChecker() {
        return new InventoryChecker();
    }

    @Bean(name = "flowerOrderingService")
```

```
    public FlowerOrderingService createService() {
        // Error occur when configure FlowerOrderingService as a Bean,
        // because DI requires InventoryChecker
        return new FlowerOrderingService();
    }
}
```

- These exceptions will be thrown:

```
org.springframework.beans.factory.UnsatisfiedDependencyException:
Error creating bean with name 'flowerOrderingService':
Unsatisfied dependency expressed through method 'setInventoryChecker'
parameter 0;

org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type
'com.bootcamp.springdemo.SIExample.InventoryChecker' available:
expected single matching bean but found 2...
```

- There are **two solutions** to the ambiguity issue. You can either:
  a. Use the `@Primary` annotation to mark the primary bean to be injected in case of ambiguous injection.
  b. Use `@Qualifer` along with `@Autowired` to provide the bean id or bean name we want to use in ambiguous situations.

Solution 1 - Use @Primary

```
@Configuration
public class BeanConfig {
    @Bean
    @Primary
    public InventoryChecker createInventoryChecker() {
        return new InventoryChecker(123);
    }

    @Bean
    public InventoryChecker createAnotherInventoryChecker() {
        return new InventoryChecker(1234);
    }

    @Bean(name = "flowerOrderingService")
```

```
    public FlowerOrderingService createService() {
        return new FlowerOrderingService();
    }
}
```

Solution 2 - Use @Qualifier + Specific Bean Name

- Note that we have added the *bean name* attribute in the `@Bean` annotation, as well as `@Qualifer` along with the name of the bean that we want to use.

```
@Configuration
public class BeanConfig {
    @Bean("InventoryCheckerA")
    public InventoryChecker createInventoryChecker() {
        return new InventoryChecker();
    }

    @Bean("InventoryCheckerB")
    public InventoryChecker createAnotherInventoryChecker() {
        return new InventoryChecker();
    }

    @Bean(name = "flowerOrderingService")
    public FlowerOrderingService createService() {
        return new FlowerOrderingService();
    }
}

public class FlowerOrderingService {
    private InventoryChecker inventoryChecker;

    @Autowired
    @Qualifier("InventoryCheckerA") // For DI, indicate which InventoryChecker should be injected
    public void setInventoryChecker(InventoryChecker inventoryChecker) {
        this.inventoryChecker = inventoryChecker;
    }
    // other code
}
```

Questions

- What are the different ways to declare a Spring bean?
- What are the three types of dependency injection in Spring?
- How do you inject a bean into another bean?
- Which injection type should we use under different situations?
- What are the two ways to resolve the ambiguity issue?