Spring 6 Configuration

Autor: Hong Le Nguyen Update : **05.2024**

Code to example on Github : https://github.com/hong1234/testBeanConfig

1 - Spring Framework terminology

Bean

A bean is a Java object instantiated, managed by the Spring container. Bean represents a reusable component that can be wired together with other beans to create the Spring application's functionality.

Bean wiring

Is realized through dependency (reference to another bean) injection. The Spring Container creates an instance of a bean based on request, then dependencies are injected.

Dependency injection happens at runtime, when the application is being put together after being compiled, and this allows a lot of flexibility, because the functionality of an application can be extended by modifying an external configuration without a recompile of the application.

Spring container

Spring container (object of type ApplicationContext) creates beans, wires them together, manages their lifecycle and visibility.

Bean configuration

To create and manage beans container needs bean metadata/ bean configuration that you supply to the container. Each component provides the bean configuration for itself.

Configuration contains the information needed for the container

How to make a bean incl
- how to get Spring to use your class as a bean
- how to inject dependencies and externally stored values into bean properties

Bean's lifecycle details

Following are the three important methods to provide bean configuration to the Spring Container −

Annotation-based configuration / implicit configuration => see (2)
Java-based configuration / explicit configuration => see (3)
XML based configuration file.
or all of them together.

2 - Annotation-based configuration

@Component annotation

You annotate class with stereotype @Component or its specializations @Service, @Repository, and @Controller

```
package hong.demo.service;
@Component
public class Boy {
	private Outfit outfit;
}
```

Component scanning enable

you write explicit an configuration class annotated with @ComponentScan to tell Spring to detect out classes annotated with @Component and to create beans from them.

```
package hong.demo.config;
@Configuration
@ComponentScan(basePackages = {"hong.demo.service"})
public class AppConfig {
	...
}
```

Bean wiring / dependency injection

To define a bean with dependencies, we have to decide how those dependencies are injected. Spring supports 3 types of dependency injection.

For example a bean of type Outfit named "boyDress" is injected in the property outfit of a bean type Boy.

There are 2 beans/objects of type Outfit. Use @Qualifier("bean-name") annotation to select which object should be injected.

```
public class GirlDress implements Outfit {...}
public class BoyDress  implements Outfit {...}

package hong.demo.service;
@Component
public class Boy {

	//  field injection ----------
	@Autowired
	@Qualifier("boyDress")
	private Outfit outfit;

	// constructor injection ----
	private Outfit outfit;
	public Boy(@Qualifier("boyDress") Outfit outfit) {
		this.outfit = outfit;
	}

	// setter injection -----------
	@Autowired
	public void setOutfit(@Qualifier("boyDress") Outfit outfit) {
		this.outfit = outfit;
	}
```

3 - Java-based configuration

Although annotation-based configuration with component scanning and automatic wiring is preferable in many cases, there are times when annotation-based configuration isn't an option and you must configure explicitly.

For instance, you want to wire components from some third-party library into your application, you don't have the source code for that library, there's no opportunity to annotate its classes with @Component and @Autowired.

Let's assume that 2 beans should be declared from GirlDress, Girl classes and the value of parameter gdress in constructor GirlDress(String gdress) is stored in file application.properties

```
src/main/resources/application.properties
        girl.dress=ROCK


package com.third.service;

public class GirlDress implements Outfit {
        private String gdress;
        public GirlDress(String gdress){
                this.gdress = gdress;
        }
}

public class Girl {
        private Outfit outfit;
        public Girl(Outfit outfit){
                this.outfit = outfit;
        }
}
```

Configuration properties injection into configuration class

Value "ROCK" is inject into property gdress of class BeanConfig

```
@Configuration
@PropertySource("classpath:application.properties")
public class BeanConfig {
        @Value("${girl.dress}")
        private String gdress;
```

Bean declaration inside a configuration class

Method girlDress() annotated with @Bean returns an instance of class GirlDress as a bean of type Outfit named girlDress (see below)

Bean wiring / dependency injection

A bean named girlDress should be injected in the property outfit of bean type Girl named girl per constructor injection

```
package hong.demo.config;
import com.third.service.*;
```

```
@Configuration
public class BeanConfig {
        ...
        private String gdress;

        @Bean
        public Outfit girlDress() {
                return new GirlDress(gdress);
        }

        // constructor injection ----------
        @Bean
        public Girl girl(@Qualifier("girlDress") Outfit girlDress) {
                return new Girl(girlDress);
        }
```

Using @Import annotation, configuration classes can be combined as desired

```
@Import({BeanConfig.class,})
@Configuration
@ComponentScan(basePackages = {"hong.demo.service"})
public class AppConfig {
}
```

The @Import annotation imports the configuration from BeanConfig.class into the AppConfig class annotated with it.

4 - show all beans configured

```
import hong.demo.config.*;

public class MainRunner {

   public static void main(String[] args) {

      Class<?>[] configurations = new Class<?>[]{AppConfig.class, BeanConfig.class};

      // a ApplicationContext is made with configs

      ApplicationContext context = new AnnotationConfigApplicationContext(configurations);


      for(String name: context.getBeanDefinitionNames()) {
                System.out.println(name);
      }


      // get a bean using Type

      AppService asv = context.getBean(AppService.class);

      asv.displayAllOutFits();
```