스프링 Bean

Spring IoC 컨테이너가 관리하는 자바 객체.

`ApplicationContext.getBean()`으로 얻어올 수 있는 객체를 Spring Bean 이라고 한다.

`ApplicationContext`가 관리하는 객체임. (`ApplicationContext`가 기억이 안난다면 IoC 교재를 참고하자.)

Spring-boot, Spring Framework 둘 모두에 사용되는 기술이다.

대체적으로 `싱글톤 객체`로 생성해서 사용해야 하는 객체를 Bean으로 등록하여 사용한다.

1. Bean으로 관리되는 주요 컴포넌트

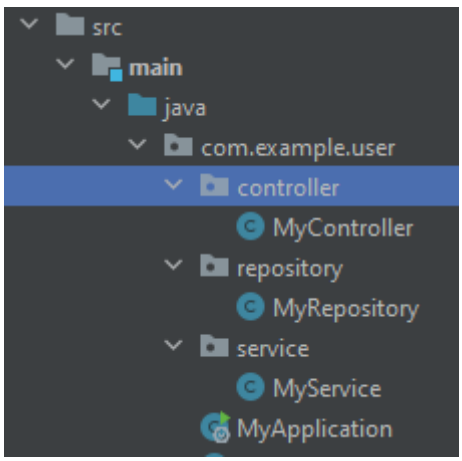
- Controller : 사용자 요청을 1차적으로 받는 역할. (우리가 알고있는 그 컨트롤러)
- Repository : Dao
- Service : 비지니스 로직을 담당
- Configuration : 서버의 동적 설정을 담당

2. Bean 등록 방법 3가지

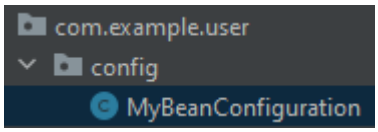
1. XML 설정 파일 사용 : https://www.baeldung.com/spring-application-context 의 **4.3 항목 참고**
2. `@Configuration` 클래스에 `@Bean`을 등록
3. `@ComponentScan` 과 `@Autowired`

1) `@Configuration` 클래스에 `@Bean`을 등록

Bean으로 등록하고자 하는 클래스가 다음과 같이 3개라고 가정해보자.
1. MyController.class
2. MyRepository.class
3. MyService.class



MyBeanConfiguration.class 추가 (이 곳에 위 3개의 Bean들을 지정할 것이다.)



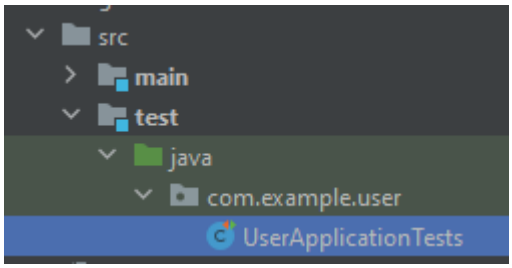
MyBeanConfiguration 클래스

```java
package com.example.user.config;

import com.example.user.controller.MyController;
import com.example.user.repository.MyRepository;
import com.example.user.service.MyService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyBeanConfiguration {

    @Bean
    public MyController myController(){

        MyController myController = new MyController();
        // myController 에 대한 설정 작업을 진행한다.
        return myController;
    }

    @Bean
    public MyService myService(){
        MyService myService = new MyService();
        // myService 에 대한 설정 작업을 진행한다.
        return myService;
    }

    @Bean
    public MyRepository myRepository(){
        MyRepository myRepository = new MyRepository();
        // myRepository 에 대한 설정 작업을 진행한다.
        return myRepository;
    }
}
```

Test

빈이 정말 잘 생성되는지 확인해보자.

test 폴더의 XXApplicationTests 를 찾아 다음을 작성




```
package com.example.user;


import com.example.user.config.MyBeanConfiguration;
import com.example.user.service.MyService;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.ApplicationContext;
import org.springframework.test.context.ContextConfiguration;

@SpringBootTest
@ContextConfiguration(classes = MyBeanConfiguration.class)
class UserApplicationTests {

    @MockBean
    ApplicationContext applicationContext; // Bean이 들어있는 ApplicationContext 객체


    @Test
    void contextLoads() {

        MyService userService = applicationContext.getBean(MyService.class);
        Assertions.assertNotNull(userService);
        // userService 가 null 이 아닌지 확인해라.
        // null 이라면 예외 발생!

    }
}
```

Test 코드 설명

- @SpringBootTest : 이 클래스는 Test 용 클래스 라는 뜻 (실제 서비스 X)
- @ContextConfiguration(classes = MyBeanConfiguration.class)

  : 웹서버 로드할 때 설정 파일로 이것들을 사용하겠다는 뜻. 근데 그 설정 유형이 class 라는 뜻. (class 말고도 xml, properties 등 옵션이 더 있음.)

  참고로 `classes` 속성에는 `Class[]` 이 들어간다. (여기서는 `MyBeanConfiguration` 1개만 넣었다.)
- @MockBean : 가짜 빈을 생성하겠다. (테스트용 프록시 객체.... 인데 자세한 것은 필요없고 그냥 테스트용 가짜 객체를 자동으로 만들어 준다고 생각하자.)

- @Test : Test로 실행할 메서드. 메인 메서드 자동 실행 되는 것 처럼, `@Test`로 지정된 모든 메서드들이 순차적으로 실행된다. (https://dotheright.tistory.com/294)
- MyService userService = applicationContext.getBean(MyService.class);

  얻어올 Bean의 타입(자료형)을 주입하면서 Bean 객체를 얻어온다. (getBean()은 팩토리 패턴으로 객체를 생성한다.)
- Assertions.assertNotNull(userService);

  유닛테스트에서 아주 자주 사용되는 assertion 계열 메서드를 활용해보았다. (junit5)

  `assertNotNull()` 은 인자객체가 `null`이면 `Exception`을 발생시킨다.

Test를 main() 에서 하고 싶다면?

```
@SpringBootApplication
public class MyApplication {

    public static void main(String[] args) {
        SpringApplication application = new
SpringApplication(MyApplication.class);
        application.setWebApplicationType(WebApplicationType.NONE);
        application.run(args);

        ApplicationContext ctx =
            new AnnotationConfigApplicationContext(MyBeanConfiguration.class);
        MyService myService = ctx.getBean(MyService.class);
        System.out.println(myService);

    }

}
```

2) `@ComponentScan` 과 `@Autowired`

앞으로 우리가 가장 많이 사용할 기술.

일단 `@ComponentScan`을 이해하기 전에 `@Component`가 무엇인지부터 확인해보자.

`@Component`

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Indexed
public @interface Component {

    The value may indicate a suggestion for a logical component name, to be turned into a Spring bean in
    case of an autodetected component.
    Returns: the suggested component name, if any (or empty String otherwise)
    String value() default "";

}
```

Spring은 초기에 Bean 을 xml 파일에 설정하거나, @Bean 애너테이션을 사용하여 등록해왔다.

(훨씬 이전에는 우리가 사용했던 원시적인 방법으로 싱글톤 패턴을 직접 선언하고 직접 getInstance()를 호출하는 방식이었음. 우리가 싱글톤 패턴을 만들어서 사용해왔던 모든 Dao, Service, Controller를 Bean이라고 생각하면 됨.)

그러다가 Spring 2.5 에서부터는 Bean 설정을 **클래스에 @Component 만 선언**해두어도 `@ComponentScan`이 지원되도록 했다.

@Component 계열의 애너테이션

1. `@Contoller`

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {

    The value may indicate a suggestion for a logical component name, to be turned into a Spring bean in
    case of an autodetected component.
    Returns: the suggested component name, if any (or empty String otherwise)
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

2. `@Repository`

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Repository {

    The value may indicate a suggestion for a logical component name, to be turned into a Spring bean in case of an autodetected component.
    Returns: the suggested component name, if any (or empty String otherwise)
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

3. `@Service`

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Service {

    The value may indicate a suggestion for a logical component name, to be turned into a Spring bean in case of an autodetected component.
    Returns: the suggested component name, if any (or empty String otherwise)
    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

4. `@Configuration`

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {

    /**
     * Explicitly specify the name of the Spring bean definition associated with the
     * {@code @Configuration} class. If left unspecified (the common case), a bean
     * name will be automatically generated.
     *
     * <p>The custom name applies only if the {@code @Configuration} class is picked
     * up via component scanning or supplied directly to an
     * {@link AnnotationConfigApplicationContext}. If the {@code @Configuration} class
     * is registered as a traditional XML bean definition, the name/id of the bean
     * element will take precedence.
     *
     * @return the explicit component name, if any (or empty String otherwise)
     * @see AnnotationBeanNameGenerator
     */
    @AliasFor(annotation = Component.class)
    String value() default "";

    /**
     * Specify whether {@code @Bean} methods should get proxied in order to enforce
     * bean lifecycle behavior, e.g. to return shared singleton bean instances even
     * in case of direct {@code @Bean} method calls in user code. This feature
     * requires method interception, implemented through a runtime-generated CGLIB
     * subclass which comes with limitations such as the configuration class and
     * its methods not being allowed to declare {@code final}.
     *
     * <p>The default is {@code true}, allowing for 'inter-bean references' via direct
     * method calls within the configuration class as well as for external calls to
     * this configuration's {@code @Bean} methods, e.g. from another configuration
     * class. If this is not needed since each of this particular configuration's
     * {@code @Bean} methods is self-contained and designed as a plain factory method
     * for container use, switch this flag to {@code false} in order to avoid
     * CGLIB subclass processing.
     *
     * <p>Turning off bean method interception effectively processes {@code @Bean}
     * methods individually like when declared on non-{@code @Configuration} classes,
     * a.k.a. "@Bean Lite Mode" (see {@code @Bean}'s javadoc). It is therefore
     * behaviorally equivalent to removing the {@code @Configuration} stereotype.
     *
     * @since 5.2
     */
    boolean proxyBeanMethods() default true;

}
```

활용 예시

```java
package com.example.user.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {
}
```

```java
package com.example.user.service;

import org.springframework.stereotype.Service;

@Service
public class MyService {
}
```

이렇게 애너테이션만 지정해줘도 applicationContext를 사용할 수 있는 환경에 바로 getBean()을 수행할 수 있다.

참고) main() 에서 `applicationContext`를 받아오는 방법

```
package com.example.user;

import com.example.user.service.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;

@SpringBootApplication
public class MyApplication implements CommandLineRunner {

    @Autowired
    ApplicationContext applicationContext;

    public static void main(String[] args) {
        SpringApplication application = new
SpringApplication(MyApplication.class);
        application.setWebApplicationType(WebApplicationType.NONE);
        application.run(args);
    }

    @Override
    public void run(String... args) throws Exception {

        System.out.println(applicationContext.getDisplayName());
        System.out.println(applicationContext.getId());

        MyService myService = applicationContext.getBean(MyService.class);
        System.out.println(myService);
    }
}
```

사실 이렇게 `ApplicationContext`를 `@Autowired`할 바엔 `MyService` 빈을 `@Autowired`하는 것이 낫다.

우리가 주로 사용할 패턴

```
package com.example.user.controller;

import com.example.user.repository.MyRepository;
import com.example.user.service.MyService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
```

```java
@Controller
public class MyController {

    @Autowired
    MyService myService;

    @Autowired
    MyRepository myRepository;

}
```

`@Autowired`는 밑에서 보자.

그렇다면 `@ComponentScan`은 무엇인가?

- `@Component`로 지정된 클래스를 웹 서버의 초기화 시점에 싱글톤 객체로 생성하는 애너테이션이다.
- 우리는 `@ComponentScan`을 쓰지 않았다. 그런데 어떻게 빈들이 자동으로 생성되었을까?

그것은 메인메서드가 있는 Application 클래스의 `@SpringBootApplication` 가 `@ComponentScan`을 가지고 있기 때문이다.

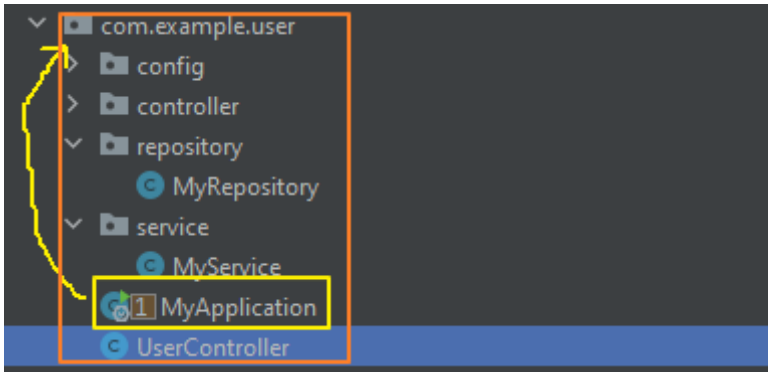
`@SpringBootApplication` 선언부

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

@ComponentScan 의 주의사항

`@CompoentScan`은 `@CompoentScan`이 선언된 클래스가 위치한 패키지부터 컴포넌트 스캔을 수행한다.

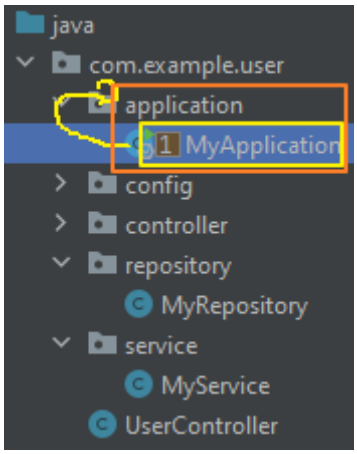
예를 들어



MyApplication에 `@CompoentScan`이 선언되어있다고 하자.

그러면 MyApplication이 위치한 com.example.user 패키지의 모든 하위 컴포넌트들이 생성된다.

그렇다면 MyApplication이 더 깊은 패키지에 위치한다면 어떻게 될까?



MyApplication이 더 깊은 패키지인 `com.example.user.application` 패키지에 속해있다면 바깥에 있던 빈들은 생성되지 않는다. (`com.example.user.application` 하위의 빈들만 생성되기 때문이다.)

이런 경우 실행해도 예외가 발생한다.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

A component required a bean of type 'com.example.user.service.MyService' that could not be found.
```

그렇기때문에 Applicaton (`main()`이 있는) 클래스는 루트 패키지에 두고 안 건드리는 것이 낫다.

참고) 원하는 패키지를 ComponentScan을 수행하는 방법

방법1. basePackages 프로퍼티 사용

```
@SpringBootApplication
@ComponentScan(basePackages = "com.example")
public class MyApplication implements CommandLineRunner {
```

이러면 com.example 하위의 모든 빈들이 생성된다. 이 경우 `@SpringBootApplication`이 어느 곳에 있던지 상관없이 원하는 위치부터 컴포넌트 스캔을 수행할 수 있다.

방법2. basePackageClasses 프로퍼티 사용

```
@SpringBootApplication
@ComponentScan(basePackageClasses = {MyService.class, MyRepository.class})
public class MyApplication implements CommandLineRunner {
```

이러면 basePagesClasses 로 지정된 클래스들의 소속패키지부터 컴포넌트 스캔을 수행한다.

기타 방법.
https://atoz-develop.tistory.com/entry/Spring-Component-Scan%EA%B3%BC-Function%EC%9D%84-%EC%82%AC%EC%9A%A9%ED%95%9C-%EB%B9%88-%EB%93%B1%EB%A1%9D-%EB%B0%A9%EB%B2%95

참고) 원하는 패지키를 ComponentScan에서 제외하는 방법

https://www.baeldung.com/spring-component-scanning 의 **3.2 항목** 참고