06. 자동 구성 기반 애플리케이션

메타 애노테이션과 합성 애노테이션

> spring boot가 어떻게 application개발을 빠르고 편리하게 해주는가의 중요한 축이 되는 자동 구성에 대해서 보자

@AutoConfiguration

이것이 어떤식으로 만들어져있고 어떻게 동작하는지 보자 (사실 Spring에 있는 기능을 boot가 쓰기 편하게 해준거임)

메타 애노테이션

메타 애너테이션을 활용하면 뭐가 좋을까?

사실 componentScanner가 바라볼때는 기능적으로 똑같음

다만 **다른 이름을 부여함으로써 우리가 코드를 읽을때** 이 클래스는 스프링 빈으로 등록되는구나 뿐만아니라 웹MVC의 컨트롤러 역할을 하는구나 등의 **정보를 알 수 있음**

또 애너테이션 자체가 달라지기 때문에 부가적인 효과를 기대할수있다.

예를 들어, @Controller 가 붙어있으면 dispatcher servlet이 이거는 웹 controller로 사용되어지는구나 그리고 애너테이션을 이용한 맵핑 정보가 들어있겠구나 판단을 하고 Request mapping이나 GetMapping을 찾아볼것이다

또 애너테이션을 새롭게 만들면 메타 애너테이션엔 없었던 새로운 element를 추가할 수 있다. 이 **애너테이션의 기능을 확장해서 활용**되도록 만들 수 있다.

상속과 혼동하지 말자!

애너테이션 자체에는 상속이라는 개념이 없다.

애너테이션에는 retention과 target정보가 꼭 있어야된다고 했는데 target이 중요한 이유는 애너테이션이 적용될 수 있는 위치 중에서 애너테이션 타입이라는 위치에 사용할 수 있어야지만 메타 애너테이션이 될 수 있다.

spring의 메타 애너테이션을 어떻게 활용되는지 보자!

HelloServiceTest.java

```
package tobyspring.helloboot;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


// 4.    ? -> ,     ANNOTATION_TYPE  ->
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@UnitTest //   @UnitTest
@interface FastUnitTest{

}

// 2.
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})
@Test //   @Test
@interface UnitTest{

}

public class HelloServiceTest {
    //@Test // -> 1. JUnit test   ->      ?   ? ->
    @UnitTest // 3.
    void simpleHelloService(){
        SimpleHelloService helloService = new SimpleHelloService();
        String ret = helloService.sayHello("Test");

        Assertions.assertThat(ret).isEqualTo("Hello Test");
    }

    @Test
    void helloDecorator(){
        HelloDecorator helloDecorator = new HelloDecorator(name -> name);
        String ret = helloDecorator.sayHello("Test");
        Assertions.assertThat(ret).isEqualTo("*Tets*");
    }
}
```

spring과 spring boot가 이 메타 애너테이션을 어떻게 활용되는지 보자!

composed annotation(합성 애너테이션)

메타 애너테이션을 하나 이상 적용해서 만드는 경우 composed annotation라고 부름

합성 애너테이션을 사용하면 class나 method에 부여하는 애너테이션이 가지고 있는 모든 메타 애너테이션들이 다 거기에 적용되어있는 것과 동일한 효과를 냄

애너테이션을 많이 쓰면 지저분해보여 -> 잘보니까 어떤 애너테이션들이 반복적으로 같이 자주 쓰여 -> composed annotation으로 만들어서 간략하게 사용 가능

예시

controller로 API 기능을 많이 개발하는데 이땐 항상 @ResposnseBody가 붙어 동시에 스캔이 되어야 하니까 @Component 도 필요함

그래서 @RestController라는 애너테이션이 생김

-> @ResponseBody와 @Controller를 동시에 메타 애너테이션으로써 가지고 있는 애너테이션임

-> 필요한걸 다 갖추었다~

합성 애노테이션의 적용

합성 애너테이션을 적용해서
HellobootApplication에 붙어있는 @Configuration, @ComponentScan을 원래 스프링부트처럼 하나로 줄여보자!

MySpringBootAnnotation.java

```
package tobyspring.helloboot;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME) // runtime  default CLASS -> class compile
@Target(ElementType.TYPE) // TYPE: class, interface, enum
@Configuration // meta annotation
@ComponentScan // meta annotation
public @interface MySpringBootAnnotation {

}
```

합성 애너테이션으로 적용

HellobootApplication.java

```
package tobyspring.helloboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.server.WebServer;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;


@MySpringBootAnnotation // 1.
public class HellobootApplication {
        // 2. factory method  , spring boot
        //
        //      ->       ! -> config.java
//        @Bean
//        public ServletWebServerFactory servletWebServerFactory(){
//                return new TomcatServletWebServerFactory();
//        }
//        @Bean
//        public DispatcherServlet dispatcherServlet(){
//                return new DispatcherServlet();
//        }

        public static void main(String[] args) {
                SpringApplication.run(HellobootApplication.class, args);
        }


}
```

HellobootApplication에서 factory method를 빼고 싶음, spring boot 처럼

Config.java

```
package tobyspring.helloboot;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.DispatcherServlet;

//@Component // class      @Bean    ? ->
@Configuration // -> meta annotation @Component
public class Config {
    @Bean
        public ServletWebServerFactory servletWebServerFactory(){
                return new TomcatServletWebServerFactory();
        }
        @Bean
        public DispatcherServlet dispatcherServlet(){
                return new DispatcherServlet();
        }
}
```

빈 오브젝트의 역할과 구분

목표: 스프링 컨테이너에 여러 종류의 빈이 등록이 되는데 조금씩 성격이 다름, 구성 정보를 작성하는 방법도 달라질 수 있다. 그래서 그 접근방법을 우리가 잘 이해를 하면 스프링 부트가 어떤 종류의 빈에 어떤 스타일의 구성 정보를 사용하는지 파악하기 좋다

어떤 식으로 구분을 해서 구성 정보를 어떤 전략으로 작성을 할 것인가 생각해보자

Spring container에 올라가는 bean들을 구분하는 방법

스프링 컨테이너가 생성하고 관리하는 빈들은

- 애플리케이션 빈
- 컨테이너 인프라스트럭처 빈

으로 구분한다.

- **애플리케이션 빈**
  - 개발자가 어떤 빈을 사용하겠다라고 명시적으로 구성정보를 제공한 것을 말함
  - 제공된 configuration metadata를 이용해서 spring container가 빈으로 등록해주는 것들을 application bean이라고 함

- **컨테이너 인프라스트럭처 빈**
  - 스프링 컨테이너 자신이거나 스프링 컨테이너가 계속 기능을 확장하면서 추가해온 것들 빈으로 등록시켜서 사용하는 것들
  - 개발을 할 때 이런 빈들을 등록해달라고 요청하진 않지만 컨테이너가 스스로 빈으로 등록해서 동작시키는 방식
  - 사실 애플리케이션 로직에서 이걸 참조하거나 사용하진 않음 (원한다면 활용은 가능)

애플리케이션 빈 내에서 구분

개발자가 구성해야하는 애플리케이션 빈도 또 구분할 수 있다

- **애플리케이션 로직 빈**
  - 애플리케이션의 기능
  - 비지니스 로직, 도메인 로직

- **애플리케이션 인프라스트럭처 빈**
  - 기술 관련
  - 대부분 직접 작성하지 않음
  - 이미 만들어져 있는 것을 이 애플리케이션에서 이거이거 사용하겠다라고 명시적으로 빈 구성정보를 작성해줘야 나중에 애플리케이션의 기능이 정상적으로 동작함
  - 애플리케이션 로직 빈이 애플리케이션 인프라스트럭처 빈을 직접 의존해서 사용하기도 하고
  - 스프링이 동작하는 메커니즘에 의해 이런 빈들이 적절한 타이밍에 자기 기능을 수행하기도 함
  - 어쨌건 구성 정보를 제공해야 등록이 됨

controller나 decorator, service는 분명히 로직 빈인데

TomcatServletWebServerFactory나 DispatcherServlet는 우리가 직접 빈으로 등록했고 없으면 안돌아가는 상황인디 얘는 뭘까?

초기 spring에서는 tomcat을 외장으로 따로 설치해서 TomcatServletWebServerFactory는 필요 없었고 MVC를 안쓰면 DispatcherServlet도 필요 없었음

하지만 spring boot application에서는 이 두가지가 꼭 빈으로 등록이 되어야하고 애플리케이션이 정상적으로 동작하는데 필요함

그럼 'container가 뜰때 필요한 컨테이너 인프라스트럭처 빈 아녀?' 라고 생각할 수 있지마는 우리가 명시적으로 선언을 해줘야지만 빈으로 등록이 되고 이것을 사용되어지는 애플리케이션이 됨

따라서 이 두가지를 애플리케이션 인프라 스트럭처 빈으로 취급하는게 맞다고 생각함

스프링 부트 개발자들이 분류한 방식은?

- 좌측
    - 애플리케이션 로직을 담고 있는 빈들은 **사용자 구성정보**를 이용해서 등록하는 빈이다라고 설명
    - ComponentScan을 통해서 자동으로 코드와 애너테이션에서 구성정보를 읽어오는 방식으로 등록함
- 우측
    - TomcatServletWebServerFactory나 DispatcherServlet 것들, DB 관련 data source나 transaction manager 등 이런 빈들은 **자동 구성정보**로 구성정보가 만들어지는 빈이다.
    - 자동구성정보라는 메커니즘을 통해서 등록이 되는 방식(AutoConfiguration)

자동 구성정보가 적용되는 방식

애플리케이션에서 사용이 될 수 있는 인프라스트럭처 빈들을 담은 Configuration class들을 만들어 놓는다 (기능으로 구분)

- servlet container를 생성하기 위한 factory bean들을 담고 있는 구성정보
- spring web을 사용하기위해 필요한 DispatcherServlet 빈을 생성하는 configuration class

를 따로 구성해둠

그리고 spring boot가 이 애플리케이션 필요에 따라서 이 중에서 필요한 configuration들을 골라서 필요한 방식으로 구성해서 자동으로 적용해준다 -> **자동 구성 방식의 기본적인 동작 원리**

스프링 부트의 자동 구성이라는 것이 어떤 원리로 적용이 되는지 code를 만들어 가면서 살펴보자

Config.java

```
package tobyspring.config;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.DispatcherServlet;

@Configuration
public class Config {
        /*
        Tomcat
         spring boot style application

        1. component scan   -> component scan    (tobyspring.config )
        2.         -> component scan
        3. application   HellobootApplication @MySpringApplication  ->  config  ??? -> @Import
(@MysSpringApplication)
        4. component scan   ??
        5.      config  ->    ? -> .            -> Config
        6.  import
        7. import  config        -> autoConfig package
        8. autoConfig package config   @MySpringBootApplication import       ->
        9.
            @MySpringBootApplication -> config package  -> helloboot package

         */
//  @Bean
//        public ServletWebServerFactory servletWebServerFactory(){
//                return new TomcatServletWebServerFactory();
//        }
//        @Bean
//        public DispatcherServlet dispatcherServlet(){
//                return new DispatcherServlet();
//        }
}
```

MySpringBootApplication.java

```
package tobyspring.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import tobyspring.config.EnableMyAutoConfiguration;
import tobyspring.config.autoConfig.DispatcherServletConfig;
import tobyspring.config.autoConfig.TomcatWebServerConfig;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration
@ComponentScan
// component annotation ,
// Import           ->
//@Import({DispatcherServletConfig.class, TomcatWebServerConfig.class}) -> 8.      -> @EnableMyAutoConfiguration

@EnableMyAutoConfiguration
public @interface MySpringBootApplication {

}
```

DispatcherServletConfig

```
package tobyspring.config.autoConfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.DispatcherServlet;

@Configuration
public class DispatcherServletConfig {
    @Bean
    public DispatcherServlet dispatcherServlet(){
        return new DispatcherServlet();
    }
}
```

TomcatWebServerConfig.java

```
package tobyspring.config.autoConfig;

import org.springframework.boot.web.embedded.tomcat.TomcatServletWebServerFactory;
import org.springframework.boot.web.servlet.server.ServletWebServerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class TomcatWebServerConfig {
    @Bean
    public ServletWebServerFactory servletWebServerFactory(){
        return new TomcatServletWebServerFactory();
    }
}
```

EnableMyAutoConfiguration.java

```
package tobyspring.config;

import org.springframework.context.annotation.Import;
import tobyspring.config.autoConfig.DispatcherServletConfig;
import tobyspring.config.autoConfig.TomcatWebServerConfig;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import({DispatcherServletConfig.class, TomcatWebServerConfig.class}) //@Import
public @interface EnableMyAutoConfiguration {
}
```

지금까지 짜여진 구조를 보자

첫번째 작업을 했을때 상태

@EnableMyAutoConfiguration을 추가하고 @MySpringBootApplication은 이것만 메타 애너테이션으로 붙여 놓은거임

그리고 구체적으로 어떤 configuration 클래스를 import할 것인가는 @EnableMyAutoConfiguration이라는 애너테이션안으로 넘김

@MySpringBootApplication는 단순화 됐제

또 auto configuration이라는 것을 사용하겠구나 정도만 @MySpringBootApplication안에 도출을 해둔것

동적인 자동 구성 정보 등록

두 개의 configuration 파일 import 하는 것을 좀 더 동적으로 처리해보자

코드에서 하드코딩으로 직접 import하는게 아니라 외부 설정파일을 이용해서 동적으로 추가하는 방식을 적용해보자

지난 시간엔,,

메인이 되는 @MySpringBootApplication에서 출발해서 메타 애너테이션으로 @EnableMyAutoConfiguration이라는 것을 추가함 . 이걸 달면 auto configuration이 적용될거다 라는 의미의 애너테이션임

이 애너테이션에서 하는 일은 사실 기존에 만들어 놨던 config class들을 import 해둔 것이다

게다가 import 문의 클래스 이름이 지금은 하드코딩되어 있음

모든 spring boot application에서 이 두가지 config가 항상 다 쓰이는 것은 아님

그래서 어떤 config가 있다고할 때 이것을 동적으로 가져올 수 있는 메카니즘을 도입해보자

어떻게 하면 config class로 만들어 둔 구성정보를 동적으로 추가할 수 있을까

즉, @EnableMyAutoConfiguration을 고치지 않고도 추가하는 방법이 뭐가 있을까

동적으로 config 가져오기

@import 만으로는 안됨 -> import selector (인터페이스)를 사용해야함

selectImports()라는 메소드가 있는데 애너테이션 메타 데이터를 전달받고 import할 config의 이름들을 String[]로써 반환함

그 string으로된 이름에 해당하는 config들을 구성 정보로 container가 사용함

하지마는 ImportSelector interface 바로 구현하는게 아니라 요거를 한번 더 확장한 defudeImportSelector를 구현할거임

이것은 다른 config이 붙은 클래스에 구성 정보 생성 작업이 모든 끝난 다음에 ImportSelector가 동작하도록 지연시켜줌

결과적으로 ImportSelector를 구현한 클래스를 사용하면 configuration class들을 프로그램에서 동적으로 결정해서 가져올 수 있다는 뜻

동적이라는 말은 어떤 config를 가져올지 읽어와서 참고해서 코드에 의해 동적으로 결정할 수 있다는 것

MyAutoConfigImortSelector.java

```
package tobyspring.config;

import org.springframework.context.annotation.DeferredImportSelector;
import org.springframework.core.type.AnnotationMetadata;

public class MyAutoConfigImportSelector implements DeferredImportSelector {
    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {

        // 1. Import  class  package  String array
        // 2.     class
        // 3.  selectImports     ? -> @Import()     -> @EnableMyAutoConfiguration
            //     config class     (    )
        return new String[]{
                "tobyspring.config.autoConfig.DispatcherServletConfig",
                "tobyspring.config.autoConfig.TomcatWebServerConfig",
        };
    }
}
```

EnableMyAutoConfiguration.java

```
package tobyspring.config;

import org.springframework.context.annotation.Import;
import tobyspring.config.autoConfig.DispatcherServletConfig;
import tobyspring.config.autoConfig.TomcatWebServerConfig;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
//@Import({DispatcherServletConfig.class, TomcatWebServerConfig.class})
@Import(MyAutoConfigImportSelector.class) // MyAutoConfigImportSelector      , string config class
public @interface EnableMyAutoConfiguration {
}
```

이렇게,,

코드(ImportSelector)에 의해서 사용할 config 정보를 return하게 하면 이 정보를 다양한 방식으로 가져올 수 있음

이젠 외부 파일을 참조해서 auto config를 만들어보자

자동 구성 정보 파일 분리

일단 보다 유연한 설정을 위해 소스코드에 등록해놨던 정보를 외부 설정 파일로 빼는 작업을 해보자

규격화된 방식으로 작성된 외부 설정 파일을 읽어오는 코드를 만들어 보자

스프링 부트의 자동 구성 정보 생성에 사용될 것들이기 때문에 애너테이션을 만들어 보자

MyAutoConfiguration.java

```
package tobyspring.config;

import org.springframework.context.annotation.Configuration;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration  // 1.
public @interface MyAutoConfiguration {
    /*
    2.        ->       config class
    3. MyAutoConfigImportSelector
     */
}
```

MyAutoConfigImportSelector.java

```
package tobyspring.config;

import org.springframework.beans.factory.BeanClassLoaderAware;
import org.springframework.boot.context.annotation.ImportCandidates;
import org.springframework.context.annotation.DeferredImportSelector;
import org.springframework.core.type.AnnotationMetadata;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

public class MyAutoConfigImportSelector implements DeferredImportSelector {//, BeanClassLoaderAware {
    // 6. class loader       ( )
    private final ClassLoader classLoader;
    public MyAutoConfigImportSelector(ClassLoader classLoader){
        this.classLoader = classLoader;
    }

    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        /*
        3.       config
             ? :   config     application
                         .

         */
//        return new String[]{
//                "tobyspring.config.autoConfig.DispatcherServletConfig",
//                "tobyspring.config.autoConfig.TomcatWebServerConfig",
//        };

        // 4. annotation class ,   . && classLoader
        //               classloader : application
        //                       spring container bean        class loader
        // 7. ImportCandidates.load() import candidate   (String iterable )
        //         configuration  class
        ImportCandidates importCandidates = ImportCandidates.load(MyAutoConfiguration.class, classLoader);

        // 8. Sring[]  iterable  iterate      list
        List<String> autoConfigs = new ArrayList<>();
//        for (String candidate : importCandidates) {
//            autoConfigs.add(candidate);
//        }

//        importCandidates.forEach(candidate -> autoConfigs.add(candidate));
        importCandidates.forEach(autoConfigs::add);

        // 9. toArray()       collection  ,
        //
        //
        return autoConfigs.toArray(new String[0]);

        //return autoConfigs.stream().toArray(String[]::new); -> java 8
        //return Arrays.copyOf(autoConfigs.toArray(), autoConfigs.size(), String[].class); ->
        //return StreamSupport.stream(importCandidates.spliterator(), false).toArray(String[]::new); -> list
    }

    // 5. BeanClassLoaderAware
//    @Override
//    public void setBeanClassLoader(ClassLoader classLoader) {
//
//    }
}

// 9.      ?! -> load()    -> resources.META-INF.spring.tobyspring.config.MyAutoConfiguration.imports
//                                                 : The names of the import candidates are stored in files
```

```
named META-INF/spring/full-qualified-annotation-name.imports on the classpath.

// 10. ImportCandidates.load(MyAutoConfiguration.class, classLoader);    (MyAutoConfiguration + .imports)
//        string array    ->
```

tobyspring.config.MyAutoConfiguration.imports

```
tobyspring.config.autoConfig.TomcatWebServerConfig
tobyspring.config.autoConfig.DispatcherServletConfig
```

이번엔,,,

- 외부에서 config 파일을 읽어오는 구조를 살펴보고
- @MyAutoConfiguration를 기존에 만든 config 파일에 적용해보자

자동 구성 애노테이션 적용

만든 애너테이션(@MyAutoConfiguration)을 import file에 의해서 로딩되는 config class에다가 사용을 해줘야함

MyAutoConfiguration.java

```
package tobyspring.config;

import org.springframework.context.annotation.Configuration;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
// 1. @Configuration      @Configuration      ?
// 4.  @Configuration element
//      proxyBeanMethods = false -> @MyAutoConfiguration    config    @Configuration
//                                  proxyBeanMethods false  Configuration   ->   ??
@Configuration(proxyBeanMethods = false)
public @interface MyAutoConfiguration {

}
// 3. ...MyAutoConfiguration.imports   config  @MyAutoConfiguration   -> config class


// 5. @Configuration
//        config class
```

DispatcherServletConfig

```
package tobyspring.config.autoConfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.DispatcherServlet;
import tobyspring.config.MyAutoConfiguration;

//@Configuration
@MyAutoConfiguration // 2.   @Configuration   config file
public class DispatcherServletConfig {
    @Bean
    public DispatcherServlet dispatcherServlet(){
        return new DispatcherServlet();
    }
}
```

지금까지 작업한 구조를 살펴보자

@EnableMyAutoConfiguration에서 config 파일을 직접 import하는 대신에 ImportSelector를 적용했음

MyAutoConfigImportSelector에서 @MyAutoConfiguration과 같은 이름을 가진 .imports 파일에서 config class의 목록을 가져오고 이것을 리턴하는 방식으로 해서 config class들이 구성정보로 import 되도록 만들었음

이렇게 만든 @MyAutoConfiguration는 기존 config class에 @Configuration를 대체해서 적용하는 것까지 하였음

이게 자동 구성 정보를 가져오는 거의 최종적인 모습임

물론 @MyAutoConfiguration을 붙인 config class들을 더 확장할거임 (중요)

하지만 기본적으로 동적으로 구성 정보를 사용할 config class 정보를 외부 설정파일에서 읽어와서 적용하는 기본틀은 지금까지 작성한 이 방식을 이용할 것임

다음에는 ,,,

@MyAutoConfiguration이 붙은 클래스에 특별한 동작방식 (proxy bean method)라는 elements의 설정값을 따라서 달라지는 부분을 볼 것임

@Configuration과 proxyBeanMethods

@Configuration의 동작 방식을 이해해보자

간단한 학습 테스트 실습

학습 테스트 : 우리가 가져다 쓰는 남의 코드의 동작방식을 정확하게 이해할 수 있도록 테스트 코드로 간단하게 샘플을 만드는 것. 즉, 테스트 코드를 학습 목적으로 만들어서 사용하는 것

config class에 대한 테스트

일단 패키지 생성 (test.java.tobyspring.study)

ConfigurationTest.java

```
package tobyspring.study;

import org.assertj.core.api.Assertions;
import org.junit.jupiter.api.Test;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

public class ConfigurationTest {
    /**
     * configuration default
     * configuration class :
     * @Bean        bean object
     *
     */
    @Test
    void configuration(){
        // 1.
        /* Bean1 <-- Common (Bean1  Common   )
           Bean2 <-- Common
             Bean1 Bean2   Common    ->

        */

        //3.
        //Assertions.assertThat(new Common()).isSameAs(new Common()); //isSameAs() ->
        //MyConfig myConfig = new MyConfig();
        //Bean1 bean1 = myConfig.bean1();
        //Bean2 bean2 = myConfig.bean2();
        //Assertions.assertThat(bean1.common).isSameAs(bean2.common); //

        /*
        4.  MyConfig class spring container   common
            MyConfig  Bean1, Bean2 Common     common  common
         */
        AnnotationConfigApplicationContext annotationConfigApplicationContext = new
AnnotationConfigApplicationContext();
        annotationConfigApplicationContext.register(MyConfig.class); // @Configuration
        annotationConfigApplicationContext.refresh(); //

        Bean1 bean1 = annotationConfigApplicationContext.getBean(Bean1.class);
        Bean2 bean2 = annotationConfigApplicationContext.getBean(Bean2.class);
        //  MyConfig  Bean1, Bean2 Common     common  common
        Assertions.assertThat(bean1.common).isSameAs(bean2.common);
        /*
        5. @Configuration   spring container
            @Configuration proxy bean method true(default)
           MyConfig  bean   bean    proxy object
            bean
         */
    }

    // 8.    @Configuration             .
    void proxyCommonMethod(){
        // 9.  decorator
        //
        MyConfigProxy myConfigProxy = new MyConfigProxy();
        Bean1 bean1 = myConfigProxy.bean1();
        Bean2 bean2 = myConfigProxy.bean2();

        // 10.     common
        //    spring container  spring container      proxy  spring container
        //           common method  common object           .
        Assertions.assertThat(bean1.common).isSameAs(bean2.common) ;

        // 11. Configuration class proxybean method true     spring container    class(MyConfigProxy )
        //     @Configuration annotation      ->
    }

    // 6.   proxy
    static class MyConfigProxy extends MyConfig{
```

```
        private Common common;
        @Override
        Common common() {
            //Common common = super.common();
            //
            // common  null  ( )   ()
            if (this.common == null) this.common = super.common();

            return this.common;
        }
        // 7. MyConfig
    }

    @Configuration
    static class MyConfig{
        @Bean
        Common common(){
            return new Common();
        }

        @Bean
        Bean1 bean1(){
            return new Bean1(common()); // common
        }

        @Bean
        Bean2 bean2(){
            return new Bean2(common());
        }
        /*
        2.    bean1(), bean2()  common   ? -> 2(  2 common)
        3.
         */
    }
    static class Bean1 {
        private final Common common;

        Bean1(Common common){
            this.common = common;
        }

    }

    static class Bean2 {
        private final Common common;

        Bean2(Common common){
            this.common = common;
        }

    }

    static class Common{

    }


}
```

MyAutoConfiguration.java

```
package tobyspring.config;

import org.springframework.context.annotation.Configuration;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
// 12. proxyBeanMethods = false    MyAutoConfiguration   config proxy
//     @Configuration              ,
@Configuration(proxyBeanMethods = false)
public @interface MyAutoConfiguration {

}
```