✨ SpringBoot BEAN 구성원리

- 작성자 : 윤요섭 사원

🏘 강의 소개

> 스프링 부트의 핵심 기능을 코드로 직접 구현하면서 스프링 부트의 동작 원리와 스프링 부트에 적용된 스프링 프레임워크의 활용법을 익히게 되는 강의

🗀 목차

4. 자동 구성 기반 애플리케이션

Spring Boot의 어노테이션 활용에 앞서서 스프링의 기본적인 어노테이션은 아래와 같다.

스프링 어노테이션

3) 스프링 어노테이션

컨트롤러, 비즈니스 로직이나 DAO 설정 등에 사용하는 스프링 어노테이션이다.

종류	설명
@Service	컨트롤러와 데이터베이스를 처리하는 클래스에 비즈니스 로직이나 트랜잭션을 처리하는 클래스를 두게 되는데 이 역할을 담당하는 클래스를 서비스 클래스로 설정한다. @Service어노테이션을 선언한 클래스는 자동 검색을 통해 빈으로 자동 설정한다. org.springframework.stereotype.Service
@Reponsitory	DAO의 역할을 담당하여 데이터 베이스와 연동해서 데이터 검색이나 입력, 수정하는 클래스를 빈으로 설정하기 위해서 사용한다. org.springframework.stereotype.Reponsitory
@Component	<context:component-csan>태그로 클래스를 빈으로 자동 설정한다. org.springframework.stereotype.Component
@Autowired	의존 관계 자동 설정. 생성자, 필드, 메서드에 적용 가능. 즉 생성자, 메서드, 필드에 붙여 스프링에서 자동 주입을 명시한다. org.springframework.beans.Factory.annotation.Autowired
@Transactional	트랜잭션 처리시 자동으로 트랜잭션을 제어하는 기능을 제공한다. org.springframework.transaction.annotation.Transactional
@Scope	빈의 범위를 싱글톤이 아닌 request, session, prototype 등으로 설정한다. org.springframework.context.annotation.Scope

4-1. 메타 어노테이션

어노테이션에 적용한 어노테이션을 메타 어노테이션이라고 한다.

4-1-1. 메타 어노테이션 살펴보기

SimpleHelloService.java

```
@Service
// 사용자 정의 클래스
public class SimpleHelloService implements HelloService{
    ...
}
```

✔ **@Service 어노테이션 구성 보기**

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

```
        annotation = Component.class
    )
    String value() default "";
}
```

여기서, @Services는 @Component 어노테이션의 메타 어노테이션임을 알 수 있다.

HelloController.java

```
@RestController
// 사용자 정의 클래스
public class HelloController {
    ...
}
```

✔ **@RestController 어노테이션 구성 보기**

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Controller
@ResponseBody
public @interface RestController {
    @AliasFor(
        annotation = Controller.class
    )
    String value() default "";
}
```

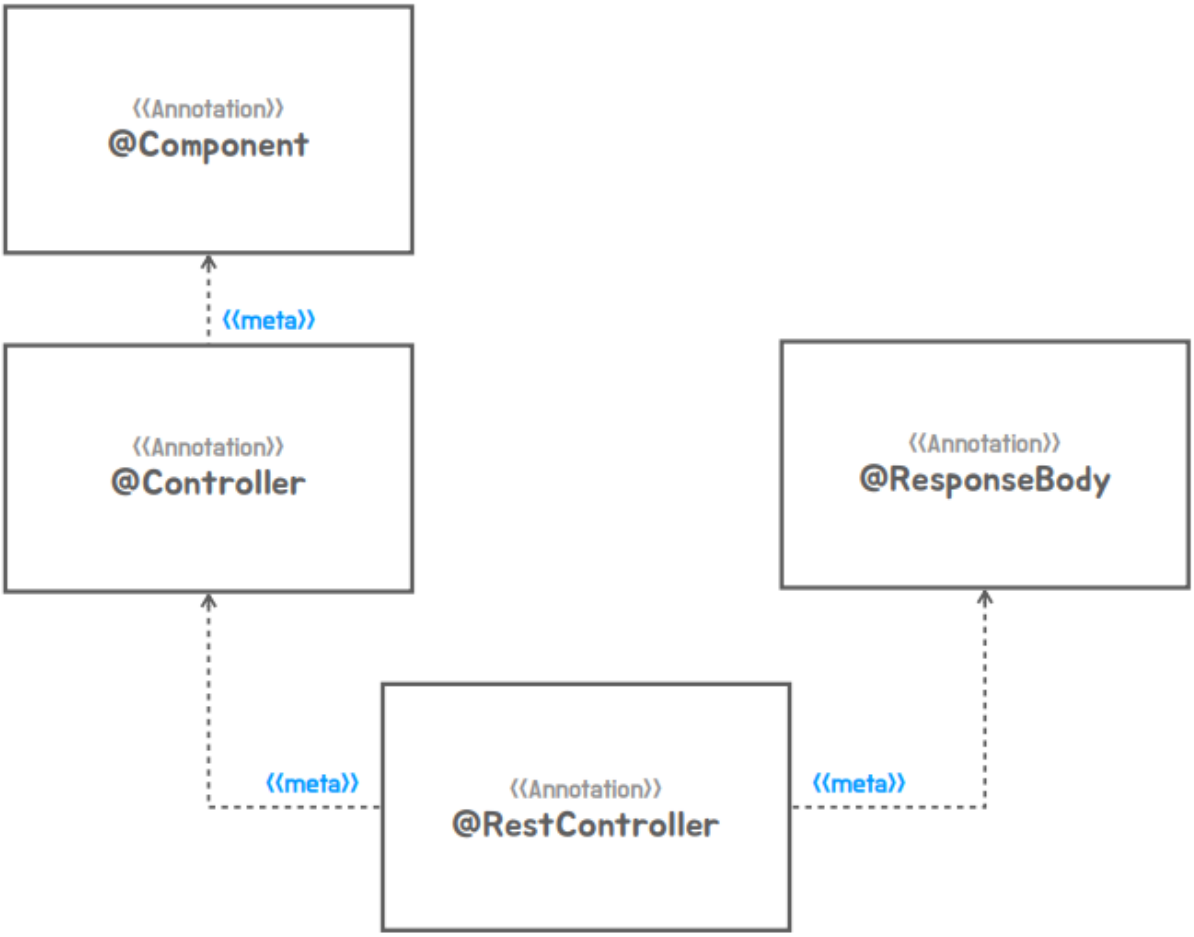
HelloController 클래스 위에 있는 @RestController는 @Controller와 @ResponseBody의 메타 어노테이션이다.

✔ **@Controller 구성 보기**

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Controller {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";
}
```

RestController @interface위의 Controller는 @Component의 메타 어노테이션이다.

이를 그림으로 표현하면 아래와 같다.



✔ **Composed annotation(합성 애노테이션)**

이렇게 메타 애노테이션을 하나 이상 적용해서 만든 애노테이션을 합성 애노테이션이라고 한다.

클래스나 메서드에 부여하는 어노테이션이 가지고 있는 모든 메타 어노테이션이 적용되어 있는 것과 같다.

😲 **기존에 있는 어노테이션만 사용해도 될 것 같은데, 굳이 메타 어노테이션을 사용하는 이유는?**

```
1.  개발자가 코드를 읽을 때, 해당 클래스가 스프링 빈으로 등록되는 지 파악함과 동시에 어노테이션의 기능과 역할에 대한 추가적인 정보를 얻을 수 있다

2. 어노테이션에 있어서 다른 부가적인 기능 및 효과를 기대할 수 있다.
```

4-1-2. 메타 어노테이션 활용 예제

- HelloServiceTest.java

```
package tobyspring.helloboot;

import org.junit.jupiter.api.Test;
...
```

```
import static org.assertj.core.api.Assertions.assertThat;

// 사용자 정의 클래스
public class HelloServiceTest {
    @Test
    void simpleHelloService(){
        System.out.println("==== simpleHelloService ====");
        SimpleHelloService helloService = new SimpleHelloService();

        String ret = helloService.sayHello("Test");
        assertThat(ret).isEqualTo("Simple HelloTest");
    }

    @Test
    void complexHelloService(){
        System.out.println("==== complexHelloService ====");
        ComplexHelloService helloService = new ComplexHelloService();

        String ret = helloService.sayHello("Test");
        assertThat(ret).isEqualTo("Complex HelloTest");
    }
}
```

- 출력 결과(테스트 이상 없음)

```
==== simpleHelloService ====
==== complexHelloService ====
```

Test 어노테이션을 붙여줌으로써 Test 기능임을 파악할 수 있으나, 어떤 테스트인지 알기 위해, 메타 어노테이션을 만든다고 가정하자.

이 때, 임의로 UnitTest라는 메타 어노테이션을 만들어서 실행할 수 있다.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Test
@interface UnitTest{}
/*
1. @Retention과 @Target은 반드시 붙여줘야 하는 필수 옵션이다.
2. @Target({ElementType.ANNOTATION_TYPE, ElementType.METHOD})으로 작성할 경우, 다른 어노테이션이
   @UnitTest 어노테이션을 활용할 수 있다.
3. @Test 어노테이션이 없으면, @UnitTest가 @Test의 메타 어노테이션인지 알 수 없으므로, 넣어주어야 한다.
*/

// 사용자 정의 클래스
public class HelloServiceTest {
    @UnitTest
```

```
    void simpleHelloService(){
        ...
    }

    @Test
    void complexHelloService(){
        ...
    }
}
```

4-2. 자동 구성 정보 파일 분리 및 어노테이션 적용

앞서, 테스트 로직을 통해 메타 어노테이션을 활용하는 예시를 작성했다면, 이번에는 메타 애노테이션을 통해 자동 구성 정보를 분리해보도록 한다.

4-2-1. 메타 애노테이션 생성, Config 클래스 생성하기

- RwkSpringBootAnnotation 애노테이션 생성하기

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration
@ComponentScan
// 사용자 정의 어노테이션
public @interface RwkSpringBootAnnotation {
}
```

@Configuration 어노테이션과 @ComponentScan 어노테이션의 메타 어노테이션을 생성하였다.

@Configuration 어노테이션의 경우 @Bean 어노테이션이 필수인데, 이를 Config 클래스에 분리시키도록 한다.

이 때, Config.java는 별도로 config 패키지 안에서 생성한다.

- 현재 패키지 구조

```
java
    ㄴ tobyspring
        ㄴ config
            ㄴ Config.java
        ㄴ helloboot
            ㄴ HelloBootApplication
            ㄴ @RwkSpringBootAnnotation
...
```

- Config.java

```
...
// 사용자 정의 클래스
public class Config {
    @Bean
    public ServletWebServerFactory servletWebServerFactory(){
        System.out.println("======== ServletWebServerFactory register =========");
        return new TomcatServletWebServerFactory();
    }

    @Bean
    public DispatcherServlet dispatcherServlet(){
        System.out.println("======== DispatcherServlet register =========");
        return new DispatcherServlet();
    }
}
```

- 출력 결과

구성과 관련된 Config.java와 같은 클래스들을 config라는 패키지에 따로 분리한 뒤, 실행시키면 다음과 같은 에러가 난다.

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

이유는 HelloBootApplication에 있는 @RwkSpringBootApplication의 @ComponentScan 범위를 벗어났기 때문이다.

이럴 경우, @Import 어노테이션을 통해 구성정보에 필요한 Class를 불러올 수 있다.

4-2-2. @Import로 컴포넌트를 구성정보에 추가하기

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration
@ComponentScan
@Import(Config.class)
// 사용자 정의 어노테이션
public @interface RwkSpringBootAnnotation {
}
```

- 단, Import에 들어갈 class는 @Component 어노테이션이 있는 class거나, @Component의 메타 어노테이션이 있는 class여야 한다.

- 출력 결과

```
...
======== ServletWebServerFactory register =========
...
======== DispatcherServlet register =========
...
```

@Import 어노테이션으로 구성정보를 import 한 뒤, 다시 정상적으로 실행되는 것을 확인 할 수 있다.

😱 Import해야할 클래스들의 종류가 많아질 경우, 하나하나 @Import 안에 추가하기 불편해질텐데 이럴 경우에는 어떻게 해야 하나요?

스프링에서 하나의 클래스의 경우, @Import(클래스이름.class)로 등록하면 되지만, 실제로 스프링에서 Import 해야할 클래스가 많아질 경우, 스프링에 있는 ImportSelector 인터페이스를 활용하여 여러 개를 import 할 수 있다.

```
public interface ImportSelector {
    String[] selectImports(AnnotationMetadata importingClassMetadata);

    @Nullable
    default Predicate<String> getExclusionFilter() {
        return null;
    }
}
```

- ImportSelector를 상속한 DeferredImportSelector 인터페이스

```
public interface DeferredImportSelector extends ImportSelector
```

4-2-3. DeferredImportSelector를 구현하여 selectImports 메서드 작성하기

1. RwkAutoConfiguration

- Path From Source Root: tobyspring.config.RwkAutoConfiguration.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration
// 사용자 정의 어노테이션
public @interface RwkAutoConfiguration {
}
```

먼저, @Configuration의 메타 어노테이션을 생성한다.

2. tobyspinrg.config.RwkAutoConfiguration.imports

- Path From Source Root: META-INF/spring/tobyspinrg.config.RwkAutoConfiguration.imports

```
tobyspring.config.autoconfig.DispatcherServletConfig
tobyspring.config.autoconfig.TomcatWebServerConfig
```

imports 파일을 만든 뒤, Import할 Class의 name space를 작성한다.

3. AutoConfigImportSelector

- Path From Source Root: tobyspring.config.AutoConfigImportSelector.java

```
// 사용자 정의 클래스
public class AutoConfigImportSelector implements DeferredImportSelector{

    private final ClassLoader classLoader;

    public AutoConfigImportSelector(ClassLoader classLoader){
        this.classLoader=classLoader;
    }

    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        // ImportCandidates::load -> imports 파일에 있는 클래스들을 클래스 로더로 읽
어서 Iterable에 저장
        Iterable<String> candidates
=ImportCandidates.load(RwkAutoConfiguration.class, classLoader);
        // new 키워드로 인스턴스
        return StreamSupport.stream(candidates.spliterator(),
false).toArray(String[]::new);
    }
}
```

ImportSelector를 상속한 DeferredImportSelector 인터페이스를 구현한 클래스를 만들어 클래스들을 읽은 뒤, 인스턴스 한다.

4. RwkSpringBootAnnotation

- Path From Source Root: tobyspring.helloboot.RwkSpringBootAnnotation.java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Configuration
@ComponentScan
@Import(AutoConfigImportSelector.class)
// 사용자 정의 어노테이션
public @interface RwkSpringBootAnnotation {
}
```

이제, helloboot 패키지 안에 있는 RwkSpringBootAnnotation 어노테이션에 @Import 안에 AutoConfigImportSelector.class를 import 해준다.

- 출력 결과

```
...
======== ServletWebServerFactory register =========
...
======== DispatcherServlet register =========
...
```

4.3. @Configuration과 proxyBeanMethods

- 01.SpringBoot WEB 동작원리에서 싱글톤 레지스트리에 대해 언급한 내용 다시 보기

> 스프링 애플리케이션에서는 싱글톤 패턴과 유사하게 애플리케이션이 동작하는 동안 딱 **하나의 오브젝트만**을 만들고 사용되게 만들어 준다. 이런 면에서 스프링 컨테이너는 **싱글톤 레지스트리**라고 한다.

😲 스프링은 왜 싱글톤 레지스트리를 고집할까? 그리고 어떻게 싱글톤 레지스트리를 유지할 수 있는 것일까?

1. 스프링은 한 개의 오브젝트만 사용하게 되는데, 만약 하나의 빈을 두 개 이상의 다른 빈에서 의존하고 있다면 FactoryMethod를 호출할 때마다, 새로운 빈이 만들어지는 문제가 생긴다.
2. 스프링은 이 문제를 해결하기 위해 @Configuration 어노테이션이 붙은 클래스는 기본적으로 Proxy를 만들어서 이를 해결해준다

> ✔ Proxy 기능 구현해보기

```java
// 사용자 정의 클래스
public class ConfigurationTest{
    // Proxy 예시
    static class MyConfigProxy extends MyConfig {
        private Common common;

        @Override
        Common common() {
        if (this.common == null) this.common = super.common();
             return this.common;
        }
    }

    @Configuration
    static class MyConfig{
        @Bean
        Common common(){
            return new Common();
        }
        // ... 이하 생략
    }
```

```
    // ... 이하 생략
}
```

MyConfigProxy라는 클래스가 없다면 아래와 같은 코드가 있을 때, new Common() 코드가 여러번 호출 될 경우, 객체의 값은 다르게 나오게 된다.

하지만 MyConfigProxy를 통해, new Common() 코드가 딱 한 번만 실행하게 만들어줄 수 있다.

@Configuration 어노테이션이 붙은 클래스는 기본적으로 proxyBeanMethods가 true로 설정되어 있어, proxyBeanMethods를 통해 싱글톤 레지스트리를 고수한다.

그러나, proxyBeanMethods를 false로 바꾼다면, @Configuration이 붙은 클래스를 빈으로 등록 할 때, Proxy를 만들지 않고, 자바 코드 그대로 동작하게 만들어줄 수 있다.

4.3.3. @Configuration(proxyBeanMethods = false)로 실제 활용 예시

- @EnableScheduling

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Import({SchedulingConfiguration.class})
@Documented
public @interface EnableScheduling {
}
```

- SchedulingConfiguration.class

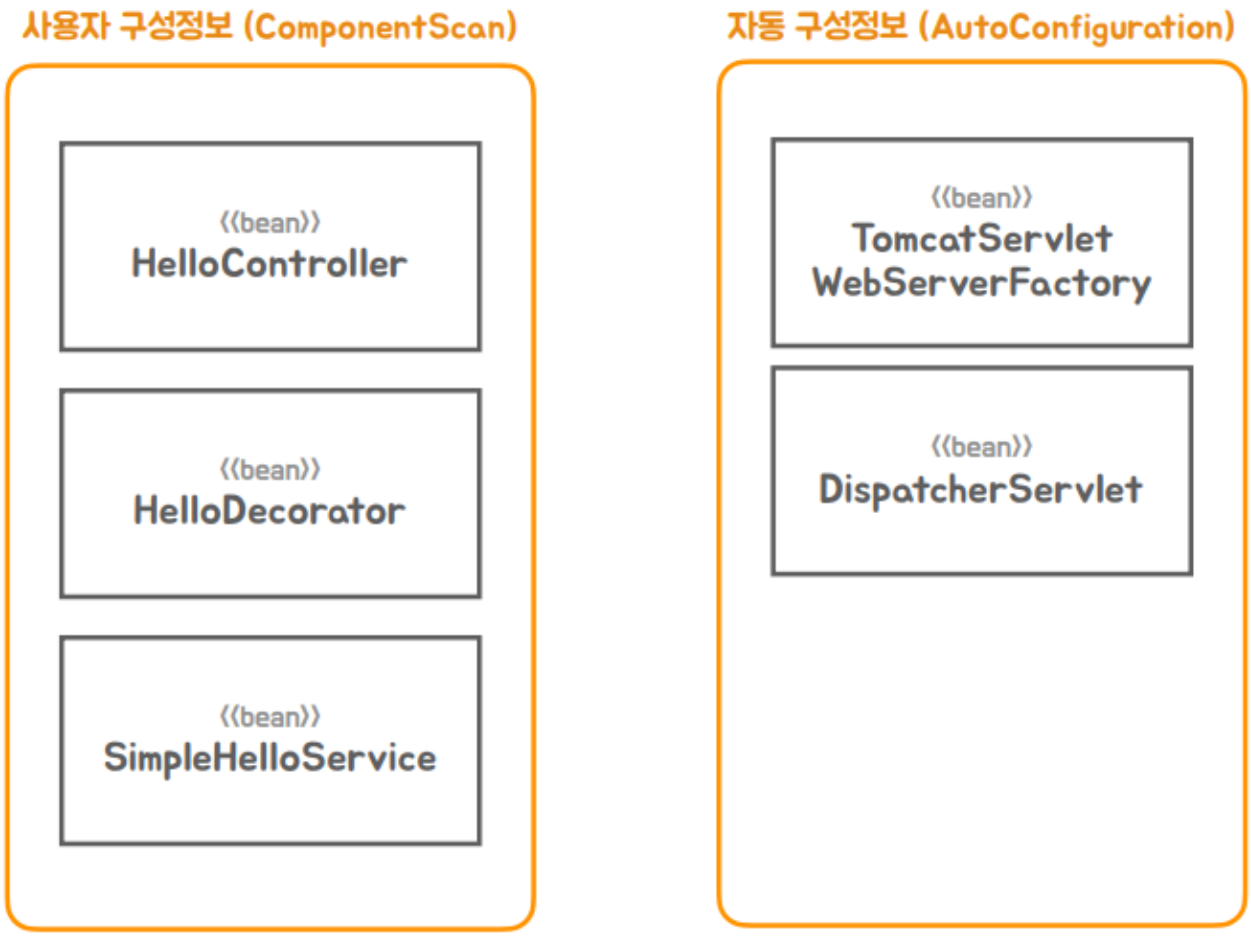
```
@Configuration(
    proxyBeanMethods = false
)
@Role(2)
public class SchedulingConfiguration {
    public SchedulingConfiguration() {
    }
    @Bean(
        name =
{"org.springframework.context.annotation.internalScheduledAnnotationProcessor"}
    )
    @Role(2)
    public ScheduledAnnotationBeanPostProcessor scheduledAnnotationProcessor() {
        return new ScheduledAnnotationBeanPostProcessor();
    }
}
```

SchedulingConfiguration 소스 코드를 보면, Bean을 하나 생성하는 코드가 있는데, 해당 Bean은 생성하는 동안에, 다른 오브젝트를 의존하고 있지 않다.

이런 경우, 매번 Proxy를 만들어서 적용할 필요가 없다고 판단하여 proxyBeanMethods를 false로 설정한 케이스이다.

4.4. @Conditional과 Condition

4-4-1. 스프링 컨테이너가 생성하고 관리하는 빈



스프링은 사용자 구성정보와 자동 구성정보를 통해 필요한 구성정보를 읽어온다.

빈 오브젝트와 역할과 구분에서 보았듯이, 스프링은 사용자 구성정보(ComponentScan)과 자동 구성정보(AutoConfiguration)을 통해, 필요한 구성정보를 읽어온다.

실제로 스프링 컨테이너는 이 외에도 더 많은 빈을 생성하고 관리한다.

- 스프링 컨테이너가 생성하고 관리하는 빈

```
1. 애플리케이션 로직 빈

애플리케이션의 기능(비즈니스 로직, 도메인 로직)을 담고 있는 빈

2. 애플리케이션 인프라스트럭처 빈

기술과 관련된 것으로 직접 작성하지 않고, 사용하겠다고 작성한 빈 구성정보
```

```
3. 컨테이너 인프라스트럭처 빈

스프링 컨테이너 자신 혹은 기능을 확장하면서 추가해온 것들을 빈으로 등록시켜서 사용
```

자동 구성정보를 통해 필요한 구성정보를 읽어오는 과정을 알아보기 위해 @AutoConfiguration을 보도록 한다.

4-4-2. @AutoConfiguration

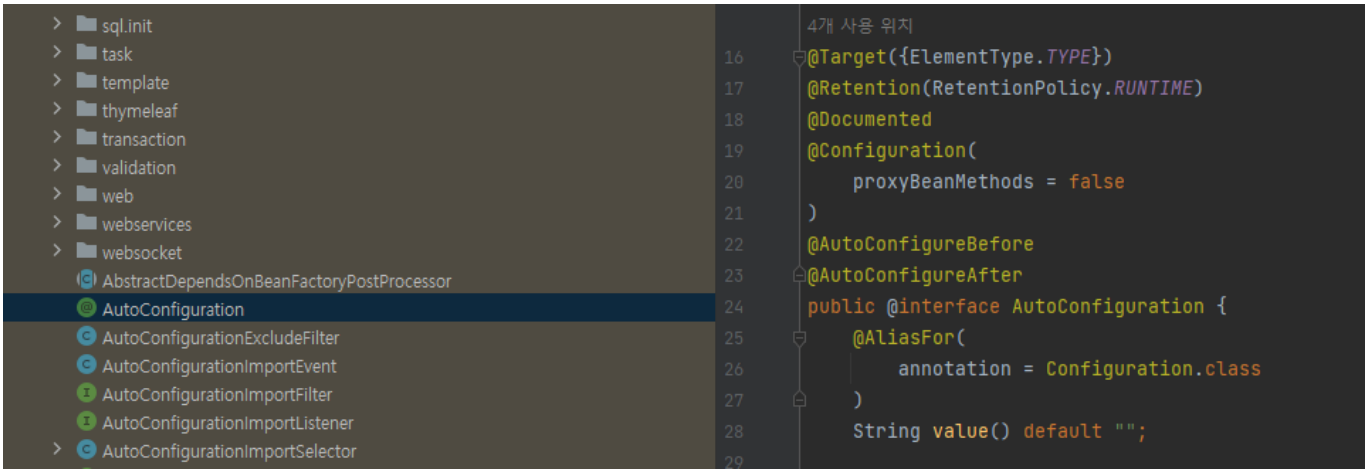
@AutoConfiguration 어노테이션을 보면, proxyBeanMethods가 false로 세팅되어있는 것을 볼 수 있다.

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration(
    proxyBeanMethods = false
)
@AutoConfigureBefore
@AutoConfigureAfter
public @interface AutoConfiguration {
    // 생략...
}
```

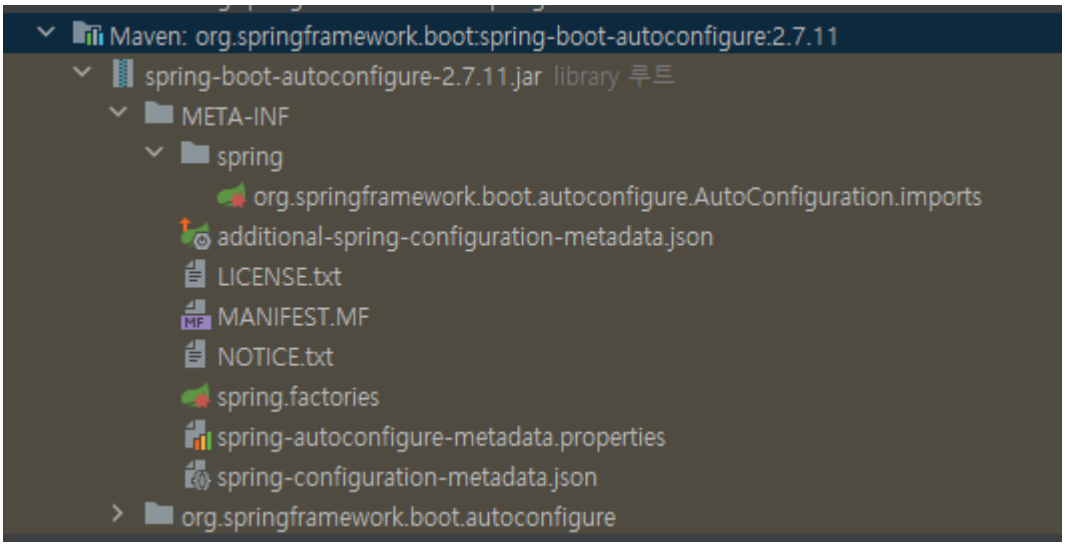
@AutoConfiguration 어노테이션이 어떻게 구성되어 있는지, spring-boot-autoconfigure-2.7.11.jar 내부를 보면 알 수 있다.

4.4.3. 메이븐 외부 라이브러리 spring-boot-autoconfigure-2.7.11.jar 내부 뜯어보기

1. org.springframework.boot.autoconfigure - @AutoConfiguration



2. META-INF/spring - org.springframework.boot.autoconfigure.AutoConfiguration.imports



- org.springframework.boot.autoconfigure.AutoConfiguration.imports 살펴보기

```
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration
```

org.springframework.boot.autoconfigure.AutoConfiguration.imports 안에는 총 144개의 클래스들이 Autoconfiguration이 Spring Boot에 기본적으로 등록 된 것을 볼 수 있다.

> 😨 **Application 시작할 때마다 144개의 Configuration class가 다 로딩이 되면서 안에 있는 빈을 다 등록할까?**

그렇게 된다면 매우 비효율적인 방식일 것이다. 스프링은 조건에 따라 컨테이너에 빈으로 등록을 결정한다.

이 때, @Conditional 어노테이션으로 모든 조건을 만족하는 경우에만 컨테이너에 빈으로 등록되도록 할 수 있다.

4.4.4. @Conditionial을 통해 조건에 따라 Tomcat과 Jetty 서버 띄우기

✔ 서블릿 컨테이너 기술 구현 라이브러리

1. Tomcat은 자바의 서블릿 컨테이너를 기술을 구현한 구현 라이브러리 중 하나이다.
2. 스프링부트는 Tomcat, Jetty, Undertow 세 가지 서블릿 컨테이너를 standalone으로 동작하는 스프링 부트 어플리케이션의 임베디드 서블릿 컨테이너로 사용할 수 있도록 해준다.

@Conditional 조건으로 Tomcat 서버를 쓸지, 다른 종류 서블릿 컨테이너(Jetty, Undertow) 중 Jetty 쓸지 결정하는 로직을 작성해보도록 한다.

- pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

dependency에 jetty를 추가해준다.

- JettyWebServerConfig

```
@RwkAutoConfiguration
// 사용자 정의 클래스
public class JettyWebServerConfig {
    @Bean("jettyWebServerFactory")
    public ServletWebServerFactory servletWebServerFactory(){
        return new JettyServletWebServerFactory();
    }
}
```

기존에 있던 TomcatWebServerConfig 클래스의 경우 @Bean을 @Bean("tomcatWebServerFactory")으로 바꾸어 준다.

- tobyspring.config.RwkAutoConfiguration.imports

```
tobyspring.config.autoconfig.DispatcherServletConfig
tobyspring.config.autoconfig.TomcatWebServerConfig
tobyspring.config.autoconfig.JettyWebServerConfig
```

마지막으로, tobyspring.config.RwkAutoConfiguration.imports에 tobyspring.config.autoconfig.JettyWebServerConfig를 추가해준다.

😲 스프링부트를 실행시키면 Jetty 서버가 뜰까? Tomcat 서버가 뜰까?

```
...
Unable to start ServletWebServerApplicationContext due to multiple
ServletWebServerFactory beans : tomcatWebServerFactory,jettyWebServerFactory
...
```

서버를 실행하면, multiple ServletWebServerFactory beans이 존재하여 선택할 수 없어서 에러가 난다.

✔ @Conditional과 Condition 인터페이스를 구현하여 Tomcat과 Jetty중 선택하여 빈 등록하기

- TomcatWebServerConfig

```

```

```
@Configuration
@Conditional(TomcatWebServerConfig.TomcatCondition.class)
// 사용자 정의 클래스
public class TomcatWebServerConfig {
    @Bean("tomcatWebServerFactory")
    public ServletWebServerFactory servletWebServerFactory(){
        return new TomcatServletWebServerFactory();
    }
    // 사용자 정의 클래스
    static class TomcatCondition implements Condition {
        @Override
        public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
            return false;
        }
    }
}
```

- JettyWebServerConfig

```
@RwkAutoConfiguration
@Conditional(JettyWebServerConfig.JettyCondition.class)
// 사용자 정의 클래스
public class JettyWebServerConfig {
    @Bean("jettyWebServerFactory")
    public ServletWebServerFactory servletWebServerFactory(){
        return new JettyServletWebServerFactory();
    }

    // 사용자 정의 클래스
    static class JettyCondition implements Condition {
        @Override
        public boolean matches(ConditionContext context, AnnotatedTypeMetadata
metadata) {
            return true;
        }
    }
}
```

@Conditional과 Condition을 구현하여 Jetty 서버만 띄우고, Tomcat 서버는 띄우지 않게 한 다음 실행하면 정상적으로 작동한다.

- HelloBootApplication 실행 결과

```
org.eclipse.jetty.server.session         : DefaultSessionIdManager workerName=node0
org.eclipse.jetty.server.session         : No SessionScavenger set, using defaults
org.eclipse.jetty.server.session         : node0 Scavenging every 600000ms
o.e.jetty.server.handler.ContextHandler  : Started o.s.b.w.e.j.JettyEmbeddedWebAppContext@2f17e30d{/,[file:///C:
org.eclipse.jetty.server.Server          : Started @1461ms
o.e.j.s.handler.ContextHandler.ROOT      : Initializing Spring DispatcherServlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
o.s.web.servlet.DispatcherServlet        : Completed initialization in 108 ms
o.e.jetty.server.AbstractConnector       : Started ServerConnector@6bca7e0d{HTTP/1.1, (http/1.1)}{0.0.0.0:8080}
o.s.b.web.embedded.jetty.JettyWebServer  : Jetty started on port(s) 8080 (http/1.1) with context path '/'
t.helloboot.HelloBootApplication         : Started HelloBootApplication in 1.24 seconds (JVM running for 1.602)
```

이번에는 반대로 TomcatCondition클래스의 matches 메서드의 리턴값을 true로 지정하고, JettyCondition 클래스의 matches 메서드의 리턴값을 false로 지정하자, Tomcat 서버만 정상적으로 작동한다.

```
o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
o.apache.catalina.core.StandardService   : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.74]
o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 588 ms

o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
t.helloboot.HelloBootApplication         : Started HelloBootApplication in 1.007 seconds (JVM running for 1.338)
```

✔ **스프링 부트의 @Conditional은 Class Conditions, Bean Conditions 등 다양한 조건으로 matches하여 체크 할 수 있다.**

스프링 부트의 @Conditional

💡 스프링 프레임워크의 @Profile도 @Conditional 애노테이션이다.

```
@Conditional(ProfileCondition.class)
public @interface Profile {
```

스프링 부트는 다음과 같은 종류의 @Conditional 애노테이션과 Condition을 제공한다. 스프링 부트의 자동 구성은 이 @Conditional을 이용한다.

Class Conditions

- @ConditionalOnClass
- @ConditionalOnMissingClass

지정한 클래스의 프로젝트내 존재를 확인해서 포함 여부를 결정한다.

주로 @Configuration 클래스 레벨에서 사용하지만 @Bean 메소드에도 적용 가능하다. 단, 클래스 레벨의 검증 없이 @Bean 메소드에만 적용하면 불필요하게 @Configuration 클래스가 빈으로 등록되기 때문에, 클래스 레벨 사용을 우선해야 한다.

Bean Conditions

- @ConditionalOnBean
- @ConditionalOnMissingBean

빈의 존재 여부를 기준으로 포함여부를 결정한다. 빈의 타입 또는 이름을 지정할 수 있다. 지정된 빈 정보가 없으면 메소드의 리턴 타입을 기준으로 빈의 존재여부를 체크한다.

컨테이너에 등록된 빈 정보를 기준으로 체크하기 때문에 자동 구성 사이에 적용하려면 @Configuration 클래스의 적용 순서가 중요하다. 개발자가 직접 정의한 커스톰 빈 구성 정보가 자동 구성 정보 처리보다 우선하기 때문에 이 관계에 적용하는 것은 안전하다. 반대로 커스톰 빈 구성 정보에 적용하는 건 피해야 한다.

Property Conditions

@ConditionalOnProperty는 스프링의 환경 프로퍼티 정보를 이용한다. 지정된 프로퍼티가 존재하고 값이 false가 아니면 포함 대상이 된다. 특정 값을 가진 경우를 확인하거나 프로퍼티가 존재하지 않을 때 조건을 만족하게 할 수도 있다.

프로퍼티의 존재를 확인해서 빈 오브젝트를 추가하고, 해당 빈 오브젝트에서 프로퍼티 값을 이용해서 세밀하게 빈 구성을 할 수도 있다.

Resource Conditions

@ConditionalOnResource는 지정된 리소스(파일)의 존재를 확인하는 조건이다.

Web Application Conditions

- @ConditionalOnWebApplication
- @ConditionalOnNotWebApplication

웹 애플리케이션 여부를 확인한다. 모든 스프링 부트 프로젝트가 웹 기술을 사용해야 하는 것은 아니다.

SpEL Expression Conditions

@ConditionalOnExpression은 스프링 SpEL(스프링 표현식)의 처리 결과를 기준으로 판단한다. 매우 상세한 조건 설정이 가능하다.

5. 외부 설정을 이용한 자동구성

스프링 부트 프로젝트를 보면 application.properties, common.properties 등의 외부 환경 설정 파일들을 흔히 보게 된다.

Spring Boot는 자동 구성에 따른 Environment 프로퍼티를 적용할 수 있다.

5-1. 자동 구성에 Environment property 적용

5.1.1. ApplicationRunner

- ApplicationRunner

```
package org.springframework.boot;

@FunctionalInterface
public interface ApplicationRunner {
    void run(ApplicationArguments args) throws Exception;
}
```

1. ApplicationRunner 함수형 인터페이스를 구현하는 빈을 등록하면, 모든 스프링 부트 초기화 작업이 끝난 다음에, ApplicationRunner를 구현한 오브젝트를 run메서드를 통해 실행해준다.
2. Application을 실행할 때, 인자값으로 받은 정보도 함께 받을 수 있다.
3. ApplicationRunner를 구현한 클래스가 빈으로 등록되기 때문에, 그 자체로 DI를 받을 수 있다.

5.1.2. ApplicationRunner로 Environment 값 받아오기

- HelloBootApplication

```
@RwkSpringBootAnnotation
// 사용자 정의 클래스
public class HelloBootApplication{
    @Bean
    ApplicationRunner applicationRunner(Environment env){
        return args -> {
            String name = env.getProperty("my.name");
            System.out.println("my.name : "+name);
        };
    }

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

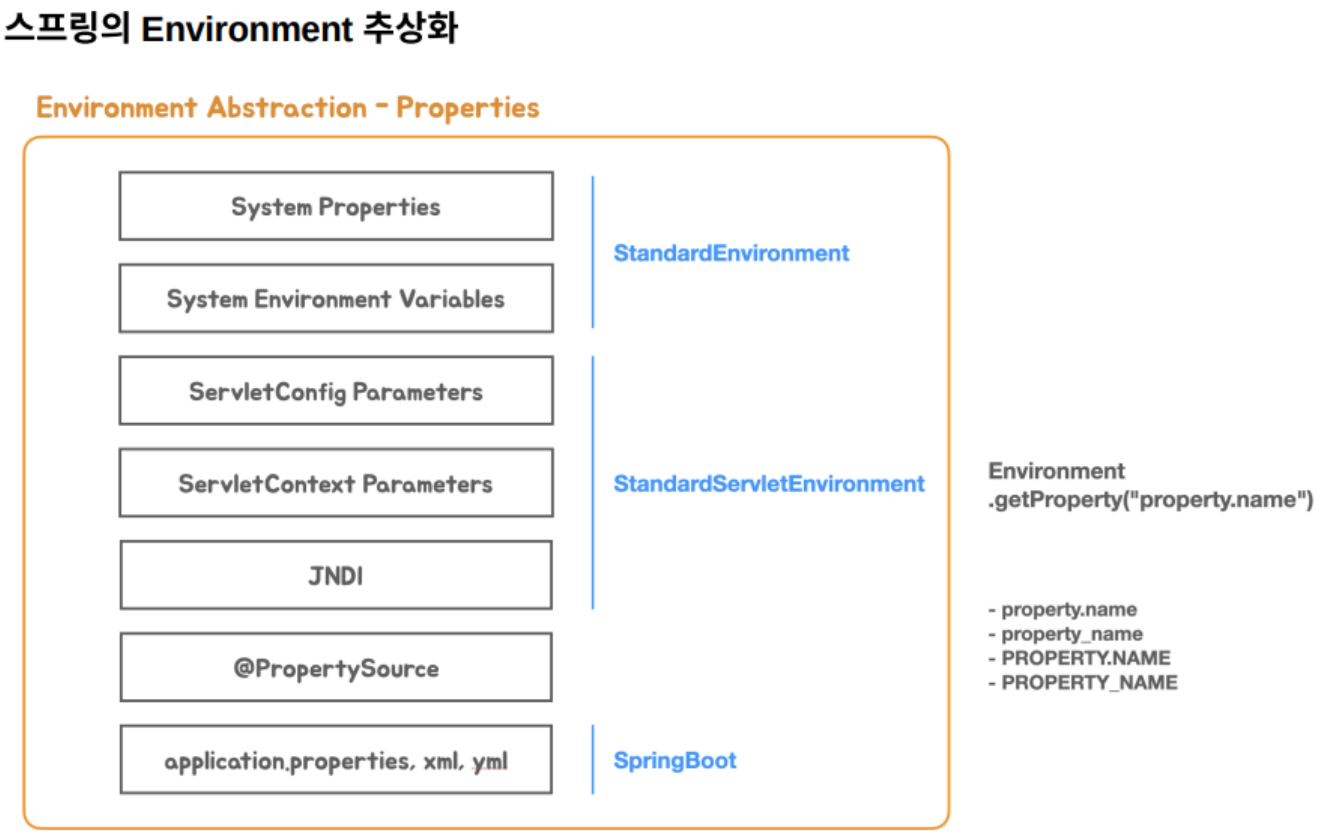
- application.properties

```
my.name=ApplicationProperties
```

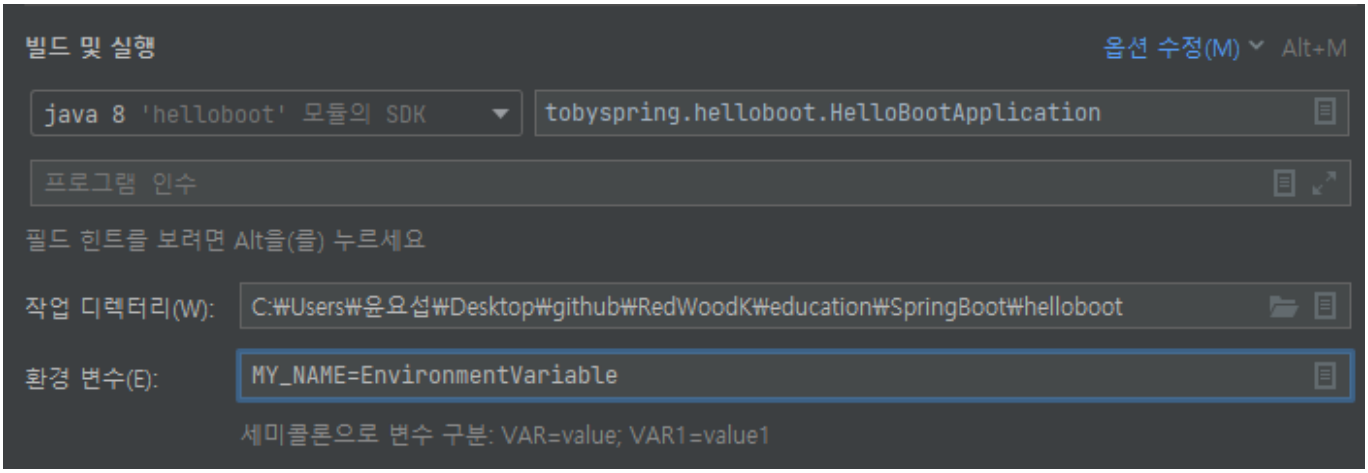
- 출력 결과

```
my.name : ApplicationProperties
```

5.1.2. Environment 우선 순위 적용



✔ **우선 순위에 있는 의 환경 변수를 세팅해보기**

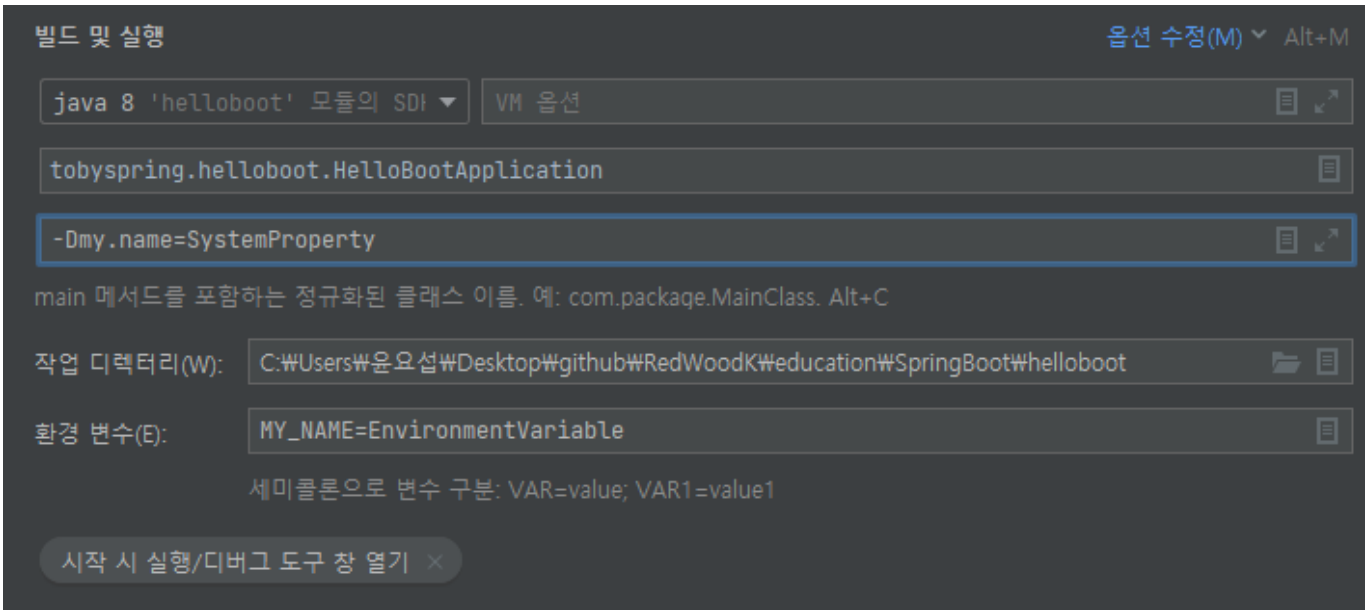


- 출력 결과

```
my.name : EnvironmentVariable
```

ApplicationProperties이 아닌 EnvironmentVariable가 출력 되는 것을 확인 할 수 있다.

✔ **우선 순위에 있는 System Environment Variables의 환경 변수를 세팅해보기**



- 출력 결과

```
my.name : EnvironmentVariable
```

ApplicationProperties이 아닌 EnvironmentVariable가 출력 되는 것을 확인 할 수 있다.

5.1.3. Tomcat실행시, Environment contextPath 받은 뒤, 테스트

외부 설정 정보를 가져와서 Tomcat의 contextPath에 적용하는 테스트를 진행해보도록 하자.

- application.properties

```
contextPath=/hydrak-bh
```

- TomcatWebServerConfig

```
@Configuration
@ConditionalRwkOnClass("org.apache.catalina.startup.Tomcat")
// 사용자 정의 클래스
public class TomcatWebServerConfig {
    /*
    *  @ConditionalOnMissingBean
    * ServletWebServerFactory Bean이 사용자가 등록했는지 체크하고
    * 없을 경우, 빈으로 등록
    * */
    @Bean("tomcatWebServerFactory")
    @ConditionalOnMissingBean
    public ServletWebServerFactory servletWebServerFactory(Environment env){
        TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();
        factory.setContextPath(env.getProperty("contextPath"));
```

```
            return factory;
        }
    }
```

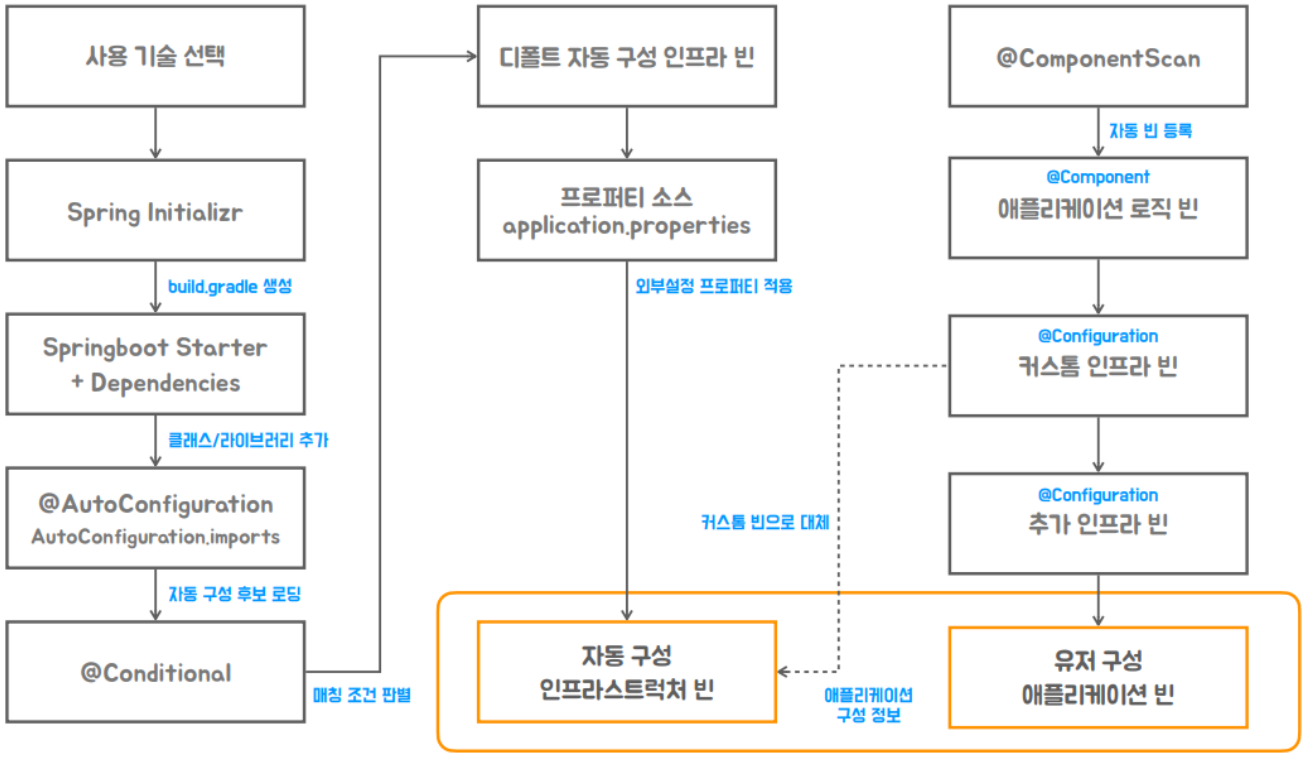
- httpie 테스트

```
C:\Users\윤요섭>http -v GET ":8080/hydrak-bh/hello?name=HydarK-BH"
GET /hydrak-bh/hello?name=HydarK-BH HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/3.2.1



HTTP/1.1 200
Connection: keep-alive
Content-Length: 24
Content-Type: text/plain;charset=ISO-8859-1
Date: Mon, 08 May 2023 16:17:23 GMT
Keep-Alive: timeout=60

Simple Hello : HydarK-BH
```

application.properties의 contextPath를 가져와서 적용까지 잘 된 것을 확인 할 수 있다.

스프링부트 자세히 살펴보기

스프링 부트의 동작 방식을 이해하고, 자신이 사용하는 기술과 관련된 자동 구성과 프로퍼티 등을 분석하고, 어떻게 활용할 수 있는지 파악하기 위해서 **02.SpringBoot BEAN구성원리**를 통해 다양한 실습을 통해 SpringBoot WEB 동작 방식과 BEAN의 구성 원리에 대해 알아볼 수 있었다.



실제로 스프링 부트의 동작 방식을 보면, Spring Boot가 시작하면서 자동 구성 목록을 import하여, Condition 조건에 따라 매핑조건을 판별하고 .properties 소스에서 외부설정 프로퍼티 적용까지 앞에서 한 번씩 실습했던 코드들이 스프링부트에 동작 방식안에 있음을 알 수 있다.

이번 교육자료를 통해 스프링 부트(Spring boot)가 어떻게 **스프링을 기반**으로 실무 환경에 사용가능한 수준의 **독립실행형 애플리케이션**을 복잡한 고민 없이 빠르게 작성할 수 있게 도와줄 수 있는지 자세히 살펴볼 수 있었다.

- 끝 -