

# [Spring] Annotation 정리

## Annotation 이란?

Annotation(@)은 사전적 의미로는 주석이라는 뜻이다.

자바에서 Annotation 은 코드 사이에 주석처럼 쓰이며 특별한 의미, 기능을 수행하도록 하는 기술이다.

즉, 프로그램에게 추가적인 정보를 제공해주는 메타데이터라고 볼 수 있다.

*meta data : 데이터를 위한 데이터)*

다음은 어노테이션의 용도를 나타낸 것이다.

컴파일러에게 코드 작성 문법 에러를 체크하도록 정보를 제공한다.

소프트웨어 개발 툴이 빌드나 배치시 코드를 자동으로 생성할 수 있도록 정보를 제공한다.

실행시(런타임시)특정 기능을 실행하도록 정보를 제공한다.

기본적으로 어노테이션을 사용하는 순서는 다음과 같다.

1. 어노테이션을 정의한다.
2. 클래스에 어노테이션을 배치한다.
3. 코드가 실행되는 중에 Reflection 을 이용하여 추가 정보를 획득하여 기능을 실시한다.

Reflection 이란?

Reflection 이란 프로그램이 실행 중에 자신의 구조와 동작을 검사하고, 조사하고, 수정하는 것이다.

Reflection 은 프로그래머가 데이터를 보여주고, 다른 포맷의 데이터를 처리하고, 통신을 위해 **serialization(직렬화)**를 수행하고, **bundling** 을 하기 위해 일반 소프트웨어 라이브러리를 만들도록 도와준다.

Java 와 같은 객체 지향 프로그래밍 언어에서 Reflection 을 사용하면 컴파일 타임에 인터페이스, 필드, 메소드의 이름을 알지 못해도 실행 중에 클래스, 인터페이스, 필드 및 메소드에 접근할 수 있다.

또한 새로운 객체의 인스턴스화 및 메소드 호출을 허용한다.

Java 와 같은 객체 지향 프로그래밍 언어에서 Reflection 을 사용하여 멤버 접근 가능성 규칙을 무시할 수 있다.

*[EX] reflection 을 사용하면 서드 파티 라이브러리의 클래스에서 private 필드의 값을 변경할 수 있다.*

Spring 에서 BeanFactory 라는 Container 에서 객체가 호출되면 객체의 인스턴스를 생성하게 되는데 이 때 필요하게 된다. 즉, 프레임워크에서 유연성있는 동작을 위해 쓰인다.

Annotation 자체는 아무런 동작을 가지지 않는 단순한 표식일 뿐이지만, Reflection 을 이용하면 Annotation 의 적용 여부와 엘리먼트 값을 읽고 처리할 수 있다.

Class 에 적용된 Annotation 정보를 읽으려면 `java.lang.Class` 를 이용하고 필드, 생성자, 메소드에 적용된 어노테이션 정보를 읽으려면 Class 의 메소드를 통해 `java.lang.reflect` 패키지의 배열을 얻어야 한다.

`Class.forName()`, `getName()`, `getModifier()`, `getFields()` `getPackage()` 등등 여러 메소드로 정보를 얻을 수 있다.

Reflection 을 이용하면 Annotation 지정만으로도 원하는 클래스를 주입할 수 있다.

```
// Without reflection
Foo foo = new Foo();
foo.hello();
```

```
// With reflection
Object foo = Class.forName("complete.classpath.and.Foo").newInstance();
// Alternatively: Object foo = Foo.class.newInstance();
Method m = foo.getClass().getDeclaredMethod("hello", **new** Class<?>[0]);
m.invoke(foo);
```

## Annotation 종류

@ComponentScan

@Component 와 @Service, @Repository, @Controller, @Configuration 이 붙은 클래스 Bean 들을 찾아서 Context 에 bean 등록을 해주는 Annotation 이다.

*@Component Annotation 이 있는 클래스에 대하여 bean 인스턴스를 생성*

ApplicationContext.xml 에 <bean id="jeongpro" class="jeongpro" /> 과 같이 xml 에 bean 을 직접등록하는 방법도 있고 위와 같이 Annotation 을 붙여서 하는 방법도 있다.

base-package 를 넣으면 해당 패키지 아래에 있는 컴포넌트들을 찾고 그 과정을 spring-context-버전(4.3.11.RELEASE).jar 에서 처리한다.

Spring 에서 @Component 로 다 쓰지 않고 @Repository, @Service, @Controller 등을 사용하는 이유는, 예를들어 @Repository 는 DAO 의 메소드에서 발생할 수 있는 unchecked exception 들을 스프링의 DataAccessException 으로 처리할 수 있기 때문이다.

또한 가독성에서도 해당 애노테이션을 갖는 클래스가 무엇을 하는지 단 번에 알 수 있다.

자동으로 등록되는 Bean 의 이름은 클래스의 첫문자가 소문자로 바뀐 이름이 자동적용된다.

*HomeController -> homeController*

@Component

@Component 은 개발자가 직접 작성한 Class 를 Bean 으로 등록하기 위한 Annotation 이다.

```
@Component
public class Student {
    public Student() {
        System.out.println("hi");
    }
}
```

```
@Component(value="mystudent")
public class Student {
    public Student() {
        System.out.println("hi");
    }
}
```

Component 에 대한 추가 정보가 없다면 Class 의 이름을 camelCase 로 변경한 것이 Bean id 로 사용된다.

하지만 @Bean 과 다르게 @Component 는 name 이 아닌 value 를 이용해 Bean 의 이름을 지정한다.

@Bean

@Bean 은 개발자가 직접 제어가 불가능한 외부 라이브러리등을 Bean 으로 만들려할 때 사용되는 Annotation 이다.

```
@Configuration
public class ApplicationConfig {
    @Bean
    public ArrayList<String> array(){
        return new ArrayList<String>();
    }
}
```

ArrayList 같은 라이브러리등을 Bean 으로 등록하기 위해서는 별도로 해당 라이브러리 객체를 반환하는 Method 를 만들고 @Bean Annotation 을 사용하면 된다.

위의 경우 @Bean 에 아무런 값을 지정하지 않았으므로 Method 이름을 camelCase 로 변경한 것이 Bean id 로 등록된다.

*method 이름이 arrayList()인 경우 arrayList 가 Bean id*

```
@Configuration
public class ApplicationConfig {
    @Bean(name="myarray")
    public ArrayList<String> array(){
        return new ArrayList<String>();
    }
}
```

위와 같이 **@Bean** 에 **name** 이라는 값을 이용하면 자신이 원하는 id 로 Bean 을 등록할 수 있다.

@Autowired

속성(field), setter method, constructor(생성자)에서 사용하며 **Type** 에 따라 알아서 **Bean** 을 주입 해준다.

무조건적인 객체에 대한 의존성을 주입시킨다.

이 Annotation 을 사용할 시, 스프링이 자동적으로 값을 할당한다.

**Controller** 클래스에서 **DAO** 나 **Service** 에 관한 객체들을 주입 시킬 때 많이 사용한다.

필드, 생성자, 입력 파라미터가 여러 개인 메소드(@Qualifier 는 메소드의 파라미터)에 적용 가능하다.

**Type** 을 먼저 확인한 후 못 찾으면 **Name** 에 따라 주입한다.

*Name 으로 강제하는 방법: @Qualifier 을 같이 명시*

```

public class Boy {
    private String name;
    private int age;

    // getters and setters
}

public class College {

    @Autowired
    private Boy student;

    public void setStudent(Boy boy) {
        this.stuendt = aboy;
    }
}

```

## annotation이용 (@Autowired)

- > 바로 주입 (Container에 Boy와 일치하는 타입)
- > singleton이므로 bean이 하나만 만들어지므로

### Bean을 주입받는 방식 (3가지)

1. @Autowired
2. setter
3. 생성자 (@AllArgsConstructor 사용) -> 권장방식

@Inject

@Autowired 어노테이션과 비슷한 역할을 한다.

@Controller

**Spring 의 Controller** 를 의미한다. Spring MVC 에서 Controller 클래스에 쓰인다.

@RestController

**Spring 에서 Controller 중 View 로 응답하지 않는, Controller** 를 의미한다.

**method 의 반환 결과를 JSON 형태로 반환한다.**

**이 Annotation 이 적혀있는 Controller 의 method 는 HttpServletResponse 로 바로 응답이 가능하다.**

**@ResponseBody** 역할을 자동적으로 해주는 Annotation 이다.

*@Controller + @ResponseBody 를 사용하면 @ResponseBody 를 모든 메소드에서 적용한다.*

@Controller 와 @RestController 의 차이

- @Controller

**API 와 view 를 동시에 사용하는 경우에 사용한다.**

**대신 API 서비스로 사용하는 경우는 @ResponseBody 를 사용하여 객체를 반환한다.**

**view(화면) return 이 주목적이다.**

- @RestController

**view 가 필요없는 API 만 지원하는 서비스에서 사용한다.**

*Spring 4.0.1 부터 제공*

**@RequestMapping 메서드가 기본적으로 @ResponseBody 의미를 가정한다.**

**data(json, xml 등) return 이 주목적이다.**

**즉, @RestController = @Controller + @ResponseBody 이다.**

@Service

**Service Class** 에서 쓰인다.

**비즈니스 로직을 수행하는 Class 라는 것을 나타내는 용도이다.**

@Repository

**DAO class** 에서 쓰인다.

**DataBase** 에 접근하는 **method** 를 가지고 있는 **Class** 에서 쓰인다.

@EnableAutoConfiguration

**Spring Application Context** 를 만들 때 자동으로 설정하는 기능을 켜다.

**classpath** 의 내용에 기반해서 자동으로 생성해준다.

만약 **tomcat-embed-core.jar** 가 존재하면 톰캣 서버가 **setting** 된다.

@Configuration

**@Configuration** 을 클래스에 적용하고 **@Bean** 을 해당 **Class** 의 **method** 에 적용하면 **@Autowired** 로 **Bean** 을 부를 수 있다.

@Required

**setter method** 에 적용해주면 **Bean** 생성시 필수 프로퍼티 임을 알린다.

**Required Annotation** 을 사용하여 **optional** 하지 않은, 꼭 필요한 속성들을 정의한다.

영향을 받는 **bean property** 를 구성할 시에는 **XML** 설정 파일에 반드시 **property** 를 채워야 한다.

*엄격한 체크, 그렇지 않으면 BeanInitializationException 예외를 발생*

```
<!-- Definition for student bean -->
<bean id = "student" class = "com.tutorialspoint.Student">
    <property name = "name" value = "Zara" />
    <property name = "age" value = "11"/>
</bean>
```

@Qualifier("id123")

**@Autowired** 와 같이 쓰이며, 같은 타입의 **Bean** 객체가 있을 때 해당 아이디를 적어 원하는 **Bean** 이 주입될 수 있도록 하는 **Annotation** 이다.

*같은 타입이 존재하는 경우 ex) 동물 = 원숭이, 닭, 개, 돼지*



같은 타입의 Bean 이 두 개 이상이 존재하는 경우에 Spring 이 어떤 Bean 을 주입해야 할지 알 수 없어서 Spring Container 를 초기화하는 과정에서 예외를 발생시킨다.


이 경우 @Qualifier 을 @Autowired 와 함께 사용하여 정확히 어떤 bean 을 사용할지 지정하여 특정 의존 객체를 주입할 수 있도록 한다.

- 예시

- **Solution to @Autowired for type ambiguity**

```
public class Boy {  
    private String name;  
    private int age;  
  
    // getters and setters ...  
}
```

```
public class College {  
  
    @Autowired  
    @Qualifier(value="tony")  
    private Boy student;  
  
    // getters ...  
}
```



Qualifier  
value

xml 설정에서 bean 의 한정자 값(qualifier value)을 설정한다.

@Autowired 어노테이션이 적용된 주입 대상에 @Qualifier 어노테이션을 설정한다.

**@Autowired** 와 마찬가지로 **Bean** 객체를 주입해주는데 차이점은 **Autowired** 는 타입으로, **Resource** 는 이름으로 연결해준다.

`javax.annotation.Resource`

표준 자바(JSR-250 표준) Annotation 으로, Spring Framework 2.5.\* 부터 지원 가능한 Annotation 이다.

**Annotation** 사용으로 인해 특정 **Framework** 에 종속적인 어플리케이션을 구성하지 않기 위해서는 **@Resource** 를 사용할 것을 권장한다.

**@Resource** 를 사용하기 위해서는 class path 내에 jsr250-api.jar 파일을 추가해야 한다.

**필드, 입력 파라미터가 한 개인 bean property setter method** 에 적용 가능하다.

`@PostConstruct, @PreConstruct`

의존하는 객체를 생성한 이후 초기화 작업을 위해 객체 생성 전/후에(pre/post) 실행해야 할 method 앞에 붙인다.

`@PreDestroy`

객체를 제거하기 전(pre)에 해야할 작업을 수행하기 위해 사용한다.

`@PropertySource`

해당 프로퍼티 파일을 Environment 로 로딩하게 해준다.

클래스에 `@PropertySource("classpath:/settings.properties")`라고 적고 클래스 내부에 **@Resource** 를 Environment 타입의 멤버 변수앞에 적으면 매핑된다.

`@ConfigurationProperties`

yaml 파일 읽는다.

default 로 classpath:application.properties 파일이 조회된다.

속성 클래스를 따로 만들어두고 그 위에 (prefix="mail")을 써서 프로퍼티의 접두사를 사용할 수 있다.

```
mail.host = mailserver@mail.com
```

```
mail.port = 9000
```

```
mail.defaultRecipients[0] = admin@mail.com
```

```
mail.defaultRecipients[1] = customer@mail.com
```

```
@Lazy
```

지연로딩을 지원한다.

@Component 나 @Bean Annotation 과 같이 쓰는데 **Class 가 로드될 때 스프링에서 바로 bean 등록을 마치는 것이 아니라 실제로 사용될 때 로딩이 이뤄지게 하는 방법이다.**

```
@Value
```

properties 에서 값을 가져와 적용할 때 사용한다.

```
@Value("${value.from.file}")
```

private String valueFromFile; 이라고 구성되어 있으면 value.from.file 의 값을 가져와서 해당 변수에 주입해준다.

spEL 을 이용해서 조금 더 고급스럽게 쓸 수 있다.

```
@Value("#{systemProperties['priority'] ?: 'some default'}")
```

```
@SpringBootApplication
```

**@Configuration, @EnableAutoConfiguration, @ComponentScan 3 가지를 하나의 애노테이션으로 합친 것이다.**

```
@RequestMapping
```

**요청 URL 을 어떤 method 가 처리할지 mapping 해주는 Annotation 이다.**

Controller 나 Controller 의 method 에 적용한다.

요청을 받는 형식인 GET, POST, PATCH, PUT, DELETE 를 정의하기도 한다.

요청 받는 형식을 정의하지 않는다면, 자동적으로 GET 으로 설정된다.

```
@RequestMapping("/list"), @RequestMapping("/home, /about");
```

```
@RequestMapping("/admin", method=RequestMethod.GET)
```

```
@Controller
```

```
// 1) Class Level
```

//모든 메서드에 적용되는 경우 “/home”로 들어오는 모든 요청에 대한 처리를 해당 클래스에서 한다는 것을 의미

```
@RequestMapping("/home")
```

```
public class HomeController {
```

```
    /* an HTTP GET for /home */
```

```
    @RequestMapping(method = RequestMethod.GET)
```

```
    public String getAllEmployees(Model model) {
```

```
        ...
```

```
    }
```

```
    /*
```

```
2) Handler Level
```

요청 url 에 대해 해당 메서드에서 처리해야 되는 경우

“/home/employees” POST 요청에 대한 처리를 addEmployee()에서 한다는 것을 의미한다.

value: 해당 url 로 요청이 들어오면 이 메서드가 수행된다.

method: 요청 method 를 명시한다. 없으면 모든 http method 형식에 대해 수행된다.

```
*/
```

```
/* an HTTP POST for /home/employees */
```

```
@RequestMapping(value = "/employees", method = RequestMethod.POST)
```

```
public String addEmployee(Employee employee) {
```

```
    ...
```

```
}
```

```
}
```

**@RequestMapping 에 대한 모든 매핑 정보는 Spring 에서 제공하는 HandlerMapping Class 가 가지고 있다.**

```
@CookieValue
```

**쿠키 값을 parameter 로 전달 받을 수 있는 방법이다.**

해당 쿠키가 존재하지 않으면 500 에러를 발생시킨다.

속성으로 required 가 있는데 default 는 true 다.

false 를 적용하면 해당 쿠키 값이 없을 때 null 로 받고 에러를 발생시키지 않는다.

```
// 쿠키의 key 가 auth 에 해당하는 값을 가져옴
```

```
public String view(@CookieValue(value="auth")String auth){...};
```

```
@CrossOrigin
```

**CORS 보안상의 문제로 브라우저에서 리소스를 현재 origin 에서 다른 곳으로의 AJAX 요청을 방지하는 것이다.**

@RequestMapping 이 있는 곳에 사용하면 해당 요청은 타 도메인에서 온 ajax 요청을 처리해준다.

```
//기본 도메인이 http://jeong-pro.tistory.com 인 곳에서 온 ajax 요청만 받아주겠다.
```

```
@CrossOrigin(origins = "http://jeong-pro.tistory.com", maxAge = 3600)
```

```
@ModelAttribute
```

**view 에서 전달해주는 parameter 를 Class(VO/DTO)의 멤버 변수로 binding 해주는 Annotation 이다.**

binding 기준은 <input name="id" /> 처럼 어떤 태그의 name 값이 해당 Class 의 멤버 변수명과 일치해야하고 setmethod 명도 일치해야한다.

```
class Person{
```

```
String id;
```

```
public void setId(String id){ this.id = id;}
```

```
public String getId(){ return this.id }
```

```
}
```

```
@Controller
```

```
@RequestMapping("/person/*")
```

```
public class PersonController{
```

```
    @RequestMapping(value = "/info", method=RequestMethod.GET)
```

```
    //view 에서 myMEM 으로 던져준 데이터에 담긴 id 변수를 Person 타입의
```

```
person 이라는 객체명으로 바인딩.
```

```
    public void show(@ModelAttribute("myMEM") Person person, Model model)
```

```
    { model.addAttribute(service.read(person.getId())); }
```

}

@GetMapping

**@RequestMapping(Method=RequestMethod.GET)과 같다.**

@PostMapping, @PutMapping, @PatchMapping, @DeleteMapping 등 도 있다.

@SessionAttributes

Session 에 data 를 넣을 때 쓰는 Annotation 이다.

**@SessionAttributes("name")**이라고 하면 Model 에 key 값이 "name"으로 있는 값은 자동으로 세션에도 저장되게 한다.

@Valid

**유효성 검증이 필요한 객체임을 지정한다.**

@InitBinder

**@Valid** 애노테이션으로 유효성 검증이 필요하다고 한 객체를 가져오기전에 수행해야할 method 를 지정한다.

@ModelAttribute

**Request 에 설정되어 있는 속성 값을 가져올 수 있다.**

@RequestBody

**요청이 온 데이터(JSON 이나 XML 형식)를 바로 Class 나 model 로 매핑하기 위한 Annotation 이다.**

POST 나 PUT, PATCH 로 요청을 받을때에, 요청에서 넘어온 body 값들을 자바 타입으로 파싱해준다.

**HTTP POST 요청에 대해 request body 에 있는 request message 에서 값을 얻어와 매핑한다.**

**RequestData 를 바로 Model 이나 클래스로 매핑한다.**

이런데 JSON 이나 XML 같은 데이터를 적절한 messageConverter 로 읽을 때 사용하거나 POJO 형태의 데이터 전체로 받는 경우에 사용한다.

```

@RequestMapping(value = "/book", method = RequestMethod.POST)
public ResponseEntity<?> someMethod(@RequestBody Book book) {
    // we can use the variable named book which has Book model type.
    try {
        service.insertBook(book);
    } catch(Exception e) {
        e.printStackTrace();
    }

    // return some response here
}

@RequestHeader

```

Request 의 header 값을 가져올 수 있다. 메소드의 파라미터에 사용한다.

```

//ko-KR,ko;q=0.8,en-US;q=0.6
@RequestHeader(value="Accept-Language")String acceptLanguage 로 사용

@RequestParam

```

@PathVariable 과 비슷하다.

**request 의 parameter 에서 가져오는 것이다. method 의 파라미터에 사용된다.**  
**?moviename=thepurge 와 같은 쿼리 파라미터를 파싱해준다.**

HTTP GET 요청에 대해 매칭되는 request parameter 값이 자동으로 들어간다.  
url 뒤에 붙는 parameter 값을 가져올 때 사용한다.

```

http://localhost:8080/home?index=1&page=2

```

```

@GetMapping("/home")
public String show(@RequestParam("page") int pageNum {
}

```

위의 경우 GET /home?index=1&page=2 와 같이 uri 가 전달될 때 page parameter 를 받아온다.

@RequestParam 어노테이션의 괄호 안의 문자열이 전달 인자 이름(실제 값을 표시)이다.

```

@RequestMapping(value = "/search/movie", method = RequestMethod.GET)
public ResponseEntity<?> someMethod(@RequestParam String moviename) {
    // request URI would be like '/search/movie?moviename=thepurge'
}

```

```
try {
    List<Movie> movies = service.searchByMoviename(moviename);
} catch(Exception e) {
    e.printStackTrace();
}
// return some response here
}

@RequestPart
```

**Request 로 온 MultipartFile 을 바인딩해준다.**

```
@RequestPart("file") MultipartFile file

@ResponseBody
```

**HttpMessageConverter** 를 이용하여 JSON 혹은 xml 로 요청에 응답할수 있게 해주는 **Annotation** 이다.

**view** 가 아닌 **JSON** 형식의 값을 응답할 때 사용하는 **Annotation** 으로 문자열을 리턴하면 그 값을 **http response header** 가 아닌 **response body** 에 들어간다.

이미 **RestController Annotation** 이 붙어 있다면, 쓸 필요가 없다.

하나 그렇지 않은 단순 컨트롤러라면, **HttpResponse** 로 응답 할 수 있게 해준다.

만약 객체를 **return** 하는 경우 **JACKSON** 라이브러리에 의해 문자열로 변환되어 전송된다.

**context** 에 설정된 **viewResolver** 를 무시한다고 보면된다.

```
@PathVariable
```

method parameter 앞에 사용하면서 해당 URL 에서 {특정값}을 변수로 받아 올 수 있다.

```
@RequestMapping(value = "/some/path/{id}", method = RequestMethod.GET)
public ResponseEntity<?> someMethod(@PathVariable int id) {
}
```

**HTTP** 요청에 대해 매핑되는 **request parameter** 값이 자동으로 **Binding** 된다.

**uri** 에서 각 구분자에 들어오는 값을 처리해야 할 때 사용한다.



REST API 에서 값을 호출할 때 주로 많이 사용한다.

```
http://localhost:8080/index/1
```

```
@PostMapping("/index/{idx}")
@ResponseBody
public boolean deletePost(@PathVariable("idx") int postNum) {
    return postService.deletePost(postNum);
}
```

@RequestParam 와 @PathVariable 동시 사용 예제

```
@GetMapping("/user/{userId}/invoices")
public List<Invoice> listUsersInvoices(@PathVariable("userId") int user,
                                       @RequestParam(value = "date", required =
false) Date dateOrNull) {
}
```

위의 경우 GET /user/{userId}/invoices?date=190101 와 같이 uri 가 전달될 때  
구분자 {userId}는 @PathVariable("userId")로,  
뒤에 이어붙은 parameter 는 @RequestParam("date")로 받아온다.

```
@ExceptionHandler(ExceptionClassName.class)
```

해당 클래스의 예외를 캐치하여 처리한다.

```
@ControllerAdvice
```

Class 위에 ControllerAdvice 를 붙이고 어떤 예외를 잡아낼 것인지는 각 메소드  
상단에 **@ExceptionHandler(예외클래스명.class)**를 붙여서 기술한다.

```
@RestControllerAdvice
```

@ControllerAdvice + @ResponseBody 다.

```
@ResponseStatus
```

사용자에게 원하는 response code 와 reason 을 return 해주는 Annotation 이다.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "my page URL
changed..") => 예외처리 함수 앞에 사용한다.
```

@EnableEurekaServer

Eureka 서버로 만들어준다.

@EnableDiscoveryClient

Eureka 서버에서 관리될 수 있는 클라이언트 임을 알려주기위한 Annotation 이다.

@Transactional

**데이터베이스 트랜잭션을 설정하고 싶은 method 에 Annotation 을 적용하면 method 내부에서 일어나는 데이터베이스 로직이 전부 성공하게되거나 데이터베이스 접근중 하나라도 실패하면 다시 롤백할 수 있게 해주는 Annotation 이다.**

@Transaction(readOnly=true, rollbackFor=Exception.class)에서 readOnly 는 읽기전용임을 알리고 rollbackFor 는 해당 Exception 이 생기면 롤백하라는 뜻이다.

@Transaction(noRollbackFor=Exception.class)는 해당 Exception 이 나타나도 롤백하지 말라는 뜻이다.

@Transaction(timeout = 10)은 10 초안에 해당 로직을 수행하지 못하면 롤백하라는 뜻이다.

메소드 내에서 Exception 이 발생하면 해당 메소드에서 이루어진 모든 DB 작업을 초기화한다.

*save 메소드를 통해서 10 개를 등록해야 하는데 5 번째에서 Exception 이 발생하면 앞에 저장된 4 개 까지 모두 롤백*

정확히 얘기하면, 이미 넣은걸 롤백시키는건 아니며, 모든 처리가 정상적으로 됐을때만 DB 에 커밋하며 그렇지 않은 경우엔 커밋하지 않는 것이다.

비지니스 로직과 트랜잭션 관리는 대부분 Service 에서 관리한다.

따라서 일반적으로 DB 데이터를 등록/수정/삭제 하는 Service 메소드는 @Transactional 를 필수적으로 가져간다.

@Cacheable

method 앞에 지정하면 해당 method 를 최초로 호출하면 캐시에 적재하고 다음부터는 동일한 method 호출이 있을 때 캐시에서 결과를 가져와서 return 하므로 method 호출 횟수를 줄여주는 Annotation 이다.

주의할 점은 입력이 같으면 항상 출력이 같은 method(=순수 함수)에 적용해야한다.

그런데 또 항상 같은 값만 뱉어주는 메서드에 적용하려면 조금 아쉬울 수 있다.

따라서 메서드 호출에 사용되는 자원이 많고 자주 변경되지 않을 때 사용하고 나중에 수정되면 캐시를 없애는 방법을 선택할 수 있다.

```
@Cacheable(value="cacheKey"), @Cacheable(key="cacheKey")
```

@CachePut

캐시를 업데이트하기 위해서 method 를 항상 실행하게 강제하는 Annotation 이다.

해당 Annotation 이 있으면 method 호출을 항상한다. 그러므로 @Cacheable 과 상충되어 같이 사용하면 안된다.

@CacheEvict

캐시에서 데이터를 제거하는 트리거로 동작하는 method 에 붙이는 Annotation 이다.

캐시된 데이터는 언제가는 지워져야한다. 그러지 않으면 결과값이 변경이 일어났는데도 기존의 데이터(캐시된 데이터)를 불러와서 오류가 발생할 수 있다.

물론 캐시 설정에서 캐시 만료시간을 줄 수도 있다.

```
@CacheEvict(value="cacheKey"), @CacheEvict(value="cacheKey", allEntries=true)
```

*allEntries 는 전체 캐시를 지울지 여부를 선택하는 것이다.*

@CacheConfig

클래스 레벨에서 공통의 캐시설정을 공유하는 기능이다.

@Scheduled

Linux 의 crontab 처럼 정해진 시간에 실행해야하는 경우에 사용한다.

@Scheduled(cron = "0 0 07 \* \* ?")

"초 분 시 일 월 요일 년(선택)에 해당 메서드 호출"

---

## Lombok Annotation

@NoArgsConstructor

기본생성자를 자동으로 추가한다.

@NoArgsConstructor(access = AccessLevel.PROTECTED)

기본생성자의 접근 권한을 protected 로 제한한다.

생성자로 *protected Posts() {}*와 같은 효과

**Entity Class** 를 프로젝트 코드상에서 기본생성자로 생성하는 것은 금지하고, JPA 에서 **Entity** 클래스를 생성하는것은 허용하기 위해 추가한다.

@AllArgsConstructor

모든 필드 값을 파라미터로 받는 생성자를 추가한다.

@RequiredArgsConstructor

**final** 이나 **@NonNull** 인 필드 값만 파라미터로 받는 생성자를 추가한다.

*final: 값이 할당되면 더 이상 변경할 수 없다.*

@Getter

Class 내 모든 필드의 Getter method 를 자동 생성한다.

@Setter

Class 내 모든 필드의 Setter method 를 자동 생성한다.

Controller 에서 @RequestBody 로 외부에서 데이터를 받는 경우엔 기본생성자 + set method 를 통해서만 값이 할당된다.

그래서 이때만 setter 를 허용한다.

*Entity Class* 에는 *Setter* 를 설정하면 안된다.

*차라리 DTO 클래스를 생성해서 DTO 타입으로 받도록 하자*

@ToString

Class 내 모든 필드의 toString method 를 자동 생성한다.

@ToString(exclude = "password")

특정 필드를 toString() 결과에서 제외한다.

클래스명(필드 1 이름=필드 1 값, 필드 2 이름=필드 2 값, ...) 식으로 출력된다.

@EqualsAndHashCode

equals 와 hashCode method 를 오버라이딩 해주는 Annotation 이다.

@EqualsAndHashCode(callSuper = true)

**callSuper 속성을 통해 equals 와 hashCode 메소드 자동 생성 시 부모 클래스의 필드까지 감안할지 안 할지에 대해서 설정할 수 있다.**

즉, callSuper = true 로 설정하면 부모 클래스 필드 값들도 동일한지 체크하며,  
callSuper = false 로 설정(기본값)하면 자신 클래스의 필드 값들만 고려한다.

@Builder

어느 필드에 어떤 값을 채워야 할지 명확하게 정하여 생성 시점에 값을 채워준다.

Constructor와 Builder의 차이

생성 시점에 값을 채워주는 역할은 똑같다.

하지만 Builder 를 사용하면 **어느 필드에 어떤 값을 채워야 할지 명확하게 인지할 수 있다.**

해당 Class 의 Builder 패턴 Class 를 생성 후 생성자 상단에 선언 시 생성자에 포함된 필드만 빌더에 포함된다.

@Data

@Getter @Setter @EqualsAndHashCode @AllArgsConstructor 을 포함한 **Lombok** 에서 제공하는 필드와 관련된 모든 코드를 생성한다.

실제로 사용하지 않는것이 좋다.

전체적인 모든 기능 허용으로 위험 존재

---

## JPA Annotation

JPA 를 사용하면 DB 데이터에 작업할 경우 실제 쿼리를 사용하지 않고 Entity 클래스의 수정을 통해 작업한다.

@Entity

실제 DB 의 테이블과 매칭될 Class 임을 명시한다.

즉, 테이블과 링크될 클래스임을 나타낸다.

Entity Class

가장 Core 한 클래스로 클래스 이름을 언더스코어 네이밍(\_)으로 테이블 이름을 매칭한다.

SalesManage 스.java -> sales\_manager table

Controller에서 쓸 DTO 클래스란??

**Request 와 Response 용 DTO 는 view 를 위한 클래스로, 자주 변경이 필요한 클래스이다.**

**Entity 클래스와 DTO 클래스를 분리하는 이유는 View Layer 와 DB Layer 를 철저하게 역할 분리하기 위해서다.**

테이블과 매핑되는 Entity 클래스가 변경되면 여러 클래스에 영향을 끼치게 되는 반면 View 와 통신하는 DTO 클래스(Request/ Response 클래스)는 자주 변경되므로 분리해야 한다.

@Table

Entity Class 에 매핑할 테이블 정보를 알려준다.

@Table(name = "USER")

**Annotation 을 생략하면 Class 이름을 테이블 이름 정보로 매핑한다.**

@Id

해당 테이블의 **PK 필드를 나타낸다.**

@GeneratedValue

**PK의 생성 규칙을 나타낸다.**

*가능한 Entity의 PK는 Long 타입의 Auto\_increment를 추천  
스프링 부트 2.0에선 옵션을 추가하셔야만 auto\_increment가 된다.*

기본값은 AUTO로, MySQL의 auto\_increment와 같이 자동 증가하는 정수형 값이 된다.

@Column

**테이블의 컬럼을 나타내며, 굳이 선언하지 않더라도 해당 Class의 필드는 모두 컬럼이 된다.**

*@Column을 생략하면 필드명을 사용해서 컬럼명과 매핑*

@Column(name = "username")

**@Column을 사용하는 이유는, 기본값 외에 추가로 변경이 필요한 옵션이 있을 경우 사용한다.**

*문자열의 경우 VARCHAR(255)가 기본값인데, 사이즈를 500으로 늘리고 싶거나(ex: title),*

*타입을 TEXT로 변경하고 싶거나(ex: content) 등의 경우에 사용*