웹 프로그래밍 - backend

기본 Annotation

1. @RestController

view가 필요없는 Rest API를 사용할 때 해당 클래스가 Controller로 사용됨을 Spring Framework에게 알리는 어노테이션

```
@RestController
public class HomeController {}
```

2. @Service

해당 클래스가 Service로 사용됨을 Spring Framework에게 알리는 어노테이션

```
@Service
public class UserServiceImpl implements UserService {
    ...
}
```

3. @Repository

해당 클래스가 Repository로 사용됨을 Spring Framework에게 알리는 어노테이션

```
@Repository
public class UserRepository extends JpaRepository<User, Long> {
    ...
}
```

위 3개의 어노테이션은 Spring이 Bean을 등록하는데 도와준다.

4. @Autowired

해당 필드의 자료형에 따라 알아서 Bean을 주입하는 어노테이션

```
@Autowired
private UserService userService;
```

5. @Qualifier

같은 자료형의 Bean이 두 개 이상 존재하는 경우 어떤 Bean을 사용할지 지정해주는 어노테이션 (같은 interface를 상속받는 클래스가 두 개 이상일 때 사용)

```
// UserService.java
public interface UserService {}

// StudentService.java
@Service
public interface StudentServiceImpl implements UserService {}

// TeacherService.java
@Service
public interface TeacherServiceImpl implements UserService {}

@Autowired
@Qualifier(value="TeacherServiceImpl")
private UserService userService; // 내용은 TeacherServiceImpl로 됨
```

컨트롤러 Annotation

1. @RequestParam

HTTP 요청에 대해 매칭되는 URL 파라미터 값이 자동으로 들어간다.

일반적으로 GET에서 사용된다.

ex) `GET http://localhost:8080/home?num1=1`

```
@GetMapping("/home")
public String home(@RequestParam("num1") int num1) {
    num1++;
    return "" + num1;
} // 2반환됨
```

2. @PathVariable

HTTP 요청에 대해 매치되는 URL의 각 구문자들의 값이 자동으로 들어간다

ex) `GET http://localhost:8080/post/1`

```
@GetMapping("/post/{id}")
public Post getPost(@PathVariable("id") int id) {
    return this.postService.getPostById(id);
}
```

3. @RequestBody

HTTP **POST,PUT**요청에 대해 Request Body에 있는 request message에서 값을 얻어와 매칭한다

ex)

```
POST http://localhost:8080/post/add

{
    "title": "post title",
    "content": "this is content"
}
```

```
@PostMapping("/post/add")
public Post addPost(@RequestBoody Post post) {
    return this.postService.addPost(post);
}
```

4. @RequestMapping

`@GetMapping` , `@PostMapping` , `@PutMapping` , `@DeleteMapping` 도 포함된다.

해당 http요청에 다음 메소드나 클래스를 매핑합니다.

`@RequestMapping` 은 method을 설정하여 특정 요청을 설정할 수 있습니다.

> `@RequestMapping(value = "/post", method = "GET")`
>
> /post의 GET요청에 대해 수행

```
@RestController
@RequestMapping("/home") // 클래스의 root url을 /home으로 설정합니다.
public class HomeController {
    @GetMapping("/hello")
    public String hello() { // GET /home/hello 에 대해 실행
        return "Hello World"
    }
    @PutMapping("/addHello") // PUT /home/addHello 에 대해 실행
    public String addHello() {

    }
    @RequestMapping("/")
    public String all() { // /home 에 대한 모든 요청에 대해 실행
        return "home";
    }
}
```

JPA 어노테이션

###

1. @Entity

실제 DB테이블과 매칭 될 클래스임을 JPA에게 알립니다.

클래스 명을 snake_case로 바꾸어 db에 생성합니다.

```
@Entity
public class StudentUser {} // 실제 DB에는 student_user로 생성
```

2. @Table

엔티티 클래스에 매핑할 테이블 정보를 알려줍니다.

생략하면 `@Entity` 설정대로 테이블을 생성합니다.

```
@Table(name = "student_user")
@Entity
public class Student { // 실제 DB 테이블의 명은 student_user가 됩니다.
    ...
}
```

3. @Id

해당 테이블의 `Primary Key` 컬럼를 나타냅니다

```
@Id
private Long id; // id bigint not null primary key
```

4. @GeneratedValue

해당 컬럼를 auto_increment를 설정합니다.

기본적으로 mysql의 모든 테이블의 auto_increment값을 공유합니다.

> 즉 다른 테이블에 id가 1로 생성되어 insert되면 현재 테이블이 insert될 때 id가 2로 insert 됩니다.

그러므로 `strategy = GenerationType.IDENTITY` 를 설정하면 해당 테이블은 독자적으로 auto_increment값을 관리합니다.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id; // id bigint not null auto_increment primary key
```

5. @Column

컬럼의 부가적인 정보를 설정할 수 있습니다.

`@Column` 어노테이션을 생략하면 필드명을 사용해서 컬럼명을 생성합니다.

```
@Column(name = "post_title")
private String title; // post_title varchar(255)

@Column(nullable = false)
private String writer_id; // writer_id varchar(255) not null

@Column(name = "post_content", length = 80)
private String content; // post_content varchar(80)
```

6. @CreationTimeStamp

데이터를 생성하는 시간을 저장하는 컬럼을 생성합니다.

```
@CreationTimeStamp
private LocalDateTime created;
```

7. @UpdateTimeStamp

데이터를 수정하는 시간을 저장하는 컬럼을 생성합니다.

```
@UpdateTimeStamp
private LocalDateTime modified;
```

6번과 7번은 JPA에서 관리하는 컬럼이기 때문에 mysql에서 직접 쿼리문을 실행할 때는 생기거나 업데이트되지 않습니다.

LomBok Annotation

1. @NoArgsConstructor

기본생성자를 자동으로 추가해줍니다.

```
@NoArgsConstructor
public class User {
    public String name;
    public User(String name) {
        this.name = name;
    }
}
// 사용
User user = new User("사람1"); // 가능
User user2 = new User(); // 가능 (Lombok이 기본 생성자를 생성해줌)
```

2. @Data

클래스의 getter, setter, equals, hashcode, 모든 프로퍼티를 설정하는 생성자를 생성해 줍니다.

```
@Data
public class Post {
    private String title;
    private int commentCount;
}
// 사용
Post p = new Post("안녕", 2); // 가능
p.setTitle("수정됨"); // 가능
```

Jackson Annotation

1. @JsonProperty

클래스가 JSON으로 변환하거나 JSON에서 db 데이터로 변환할 때 해당 필드의 행동을 설정합니다.

- value: json의 key값의 이름을 설정합니다.
- access: 접근 가능한 권한을 설정합니다.
    - WRITE_ONLY: json으로 변환하지 않고 데이터를 db에 insert할 때만 반영합니다. ex) 비밀번호
    - READ_ONLY: json으로 변환은 하지만 데이터를 db에 insert할 때 반영하지 않습니다. ex) 데이터베이스에 필요없는 데이터
    - READ_WRITE(default): 둘 다 허용합니다.

```
@JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
@Column(nullable = false)
private String password;
```

Spring AOP Annotation

키워드:

- AOP: Aspect-Obiented Programming - 관점 지향 프로그래밍
- Advice: 언제 코드를 실행하는지 정하는 것 (Spring에서는 `@Before, @After`와 같은 어노테이션)

1. @Aspect

해당 클래스를 AOP 클래스로 사용한다고 알립니다.

```
@Aspect
public class TestAOP {
    ...
}
```

2. @PointCut

타켓 메소드를 설정합니다.

execution([리턴 타입][타갯이 되는 메소드][(인자 타입)]

> ex) `"execution(* com.kr.hs.dgsw.board.BoardService.get*(..))"`
>
> BoardService 내에 있는 모든 get으로 시작하는 메소드를 타겟으로

> 앞에 execution은 지정자라고 부르고 여러개가 존재한다. 안배웠으니 생략.

```
@Aspect
public class TestAOP {
    @PointCut("execution(* kr.hs.dgsw.board.BoardService.*(..))")
    private void allPointcut() {}
    // BoardService의 모든 메소드에 대한 지점 생성
}
```

아래 `@Before, @After, @Around, @AfterRturning, @AfterThrowing`과 인자값 설명은 같습니다.

> 다만 `@PointCut`은 지점을 선택해 주는 역할을 합니다. (특정 행동을 수행하는 것이 아니다.)

```
@PointCut(...)
private void allPointCut() {}

@Before("allPointCut()")
public void beforePointcut() {
    ...
}
```

> 위 코드와 같이 다른 어드바이스들이 해당 pointcut메소드를 타겟으로 설정함으로써 쉽게 사용하게 만들 수 있다.

아래 어드바이스들의 지점도 execution... 과 같이 사용할 수 있지만 간결함을 위해 모두 allPointcut()으로 사용하겠습니다.

3. @Before

타겟 메소드가 호출되기 전에 해당 메소드를 수행합니다.

> "해당"이란 어노테이션 다음에 오는 메소드를 뜻합니다.

```
@Before("allPointcut()")
public void beforePointcut(JoinPoint joinpoint) {
    System.out.println("before execute")
}
```

4. @After

타겟 메소드가 호출된 후 결과(성공, 예외)와 상관없이 완료되면 해당 메소드를 수행합니다.

```
@After("allPointcut()")
public void afterPointcut(JoinPoint joinpoint) {
    ...
}
```

5. @AfterReturning

타겟 메소드가 성공적으로 결과값을 반환 후에 해당 메소드를 수행합니다.

인자값으로 returning을 지정할 수 있다.

```
@AfterReturning(pointcut = "allPointcut()", returning="res")
public void afterReturnPointcut(Joinpoint jp, Object res) {
    System.out.println("return Value is " + res);
}
```

6. @AfterThrowing

타겟 메소드가 수행 중 예외를 던지게 되면 해당 메소드를 수행합니다.

```
@AfterThrowing(pointcut = "allPointcut()", throwing="ex")
public void afterThrowingPointcut(Joinpoint jp, Exception ex) {
    System.out.println("exception throwed", ex);
}
```

7. @Around

타겟 메소드가 실행 전후에 해당 메소드를 수행합니다.

`ProcessingJoinPoint`가 인자값으로 오는데, 해당 메소드에서 `proceed()`를 실행하여 해당 메소드 내에서 코드를 실행할 수 있습니다.

```
@Around("allPointcut()")
public void roundPointCut(ProceedingJoinPoint point) {
    long time1 = System.currentTimeMillis();
    Object retVal = point.proceed();
    System.out.println("return Value is " + resVal);
    long processTime = System.currentTimeMillis() - time1;
    System.out.println("time " + processTime);
    return retVal;
} // 해당 메소드를 실행하는데 걸린 시간을 측정하는 코드입니다.
```

Spring AOP xml 설정

Annotation을 사용할 때

```
<aop:aspectj-autoproxy />
```

Annotation을 사용하지 않을 때

```xml
<aop:config>
  <aop:pointcut
    id="allPointcut"
    expression="execution(* kr.hs.dgsw.biz..*Impl.*(..))"
  />
  <aop:pointcut
    id="getPointcut"
    expression="execution(* kr.hs.dgsw.biz.*.*Impl.get*(..))"
  />
  <aop:aspect ref="LogAdvice">
    <aop:before pointcut-ref="allPointcut" method="printLog"/>
    <aop:after  pointcut-ref="allPointcut" method="AfterprintLog"/>
  </aop:aspect>
  <aop:aspect ref="AroundAdvice">
    <aop:around pointcut-ref="allPointcut" method="arroundLog"/>
  </aop:aspect>
  <aop:aspect ref="AfterReturningAdvice">
    <aop:after-returning
      pointcut-ref="allPointcut"
      method="afterLog"
      returning="returnObj"/>
  </aop:aspect>

  <aop:aspect ref="AfterThrowingAdvice">
    <aop:after-throwing
      pointcut-ref="allPointcut"
      method="exceptionLog"
      throwing="exceptionObj"/>
  </aop:aspect>
</aop:config>
```