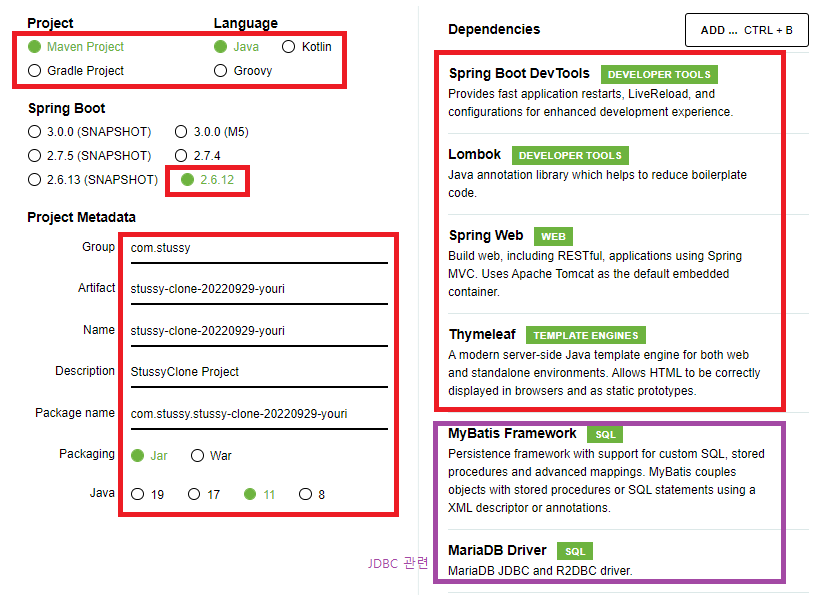
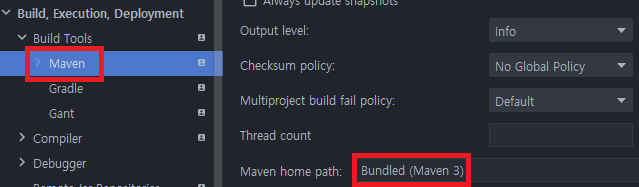
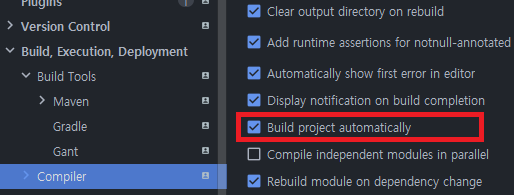
## **인텔리제이**

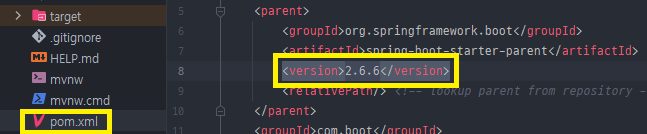
| 프로젝트: <https://start.spring.io/>  https://start.spring.io/#!type=maven-project&language=java&platformVersion=2.6.12&packaging=jar&jvmVersion=11&groupId=com.stussy&artifactId=stussy-clone-20220929-youri&name=stussy-clone-20220929-youri&description=StussyClone%20Project&packageName=com.stussy.stussy-clone-20220929-youri&dependencies=devtools,lombok,web,thymeleaf,mybatis,mariadb |
| --- |



### **- [ 기본세팅 ]**



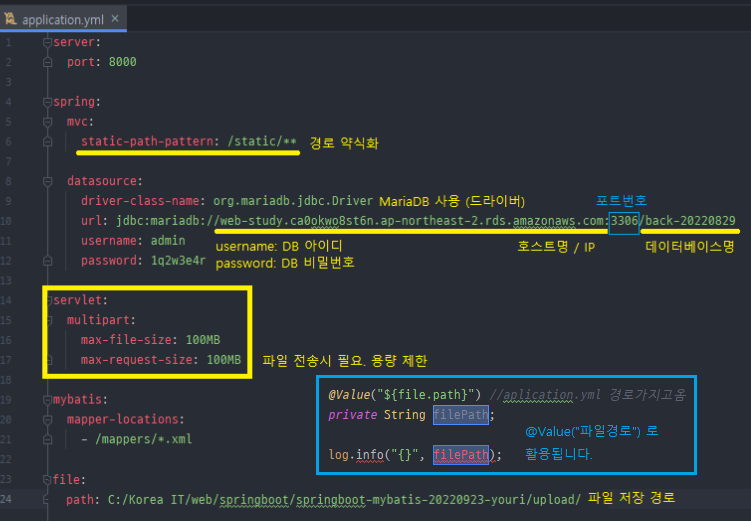




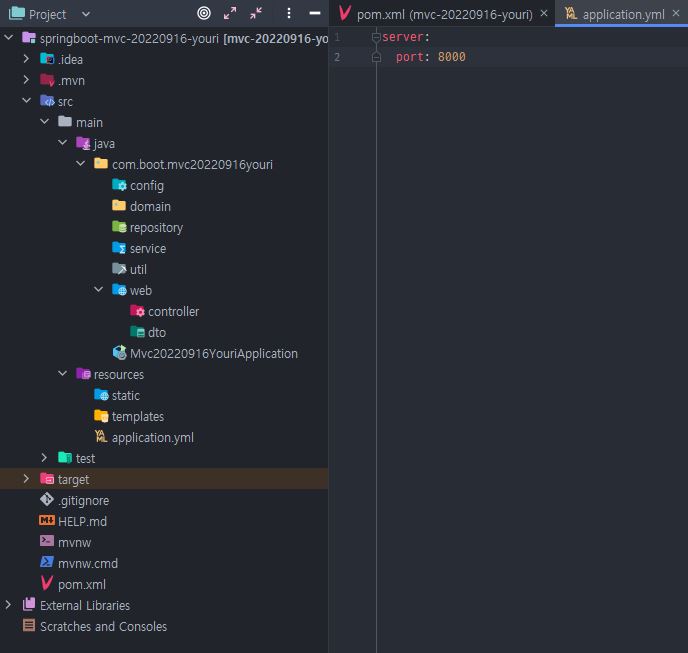
#### 

view 관련만 잡아준다. 기본 값이라 생략이 가능하다.

* prefix: resource 경로 앞부분
* suffix: resource 경로 뒷부분. 보통 확장자명

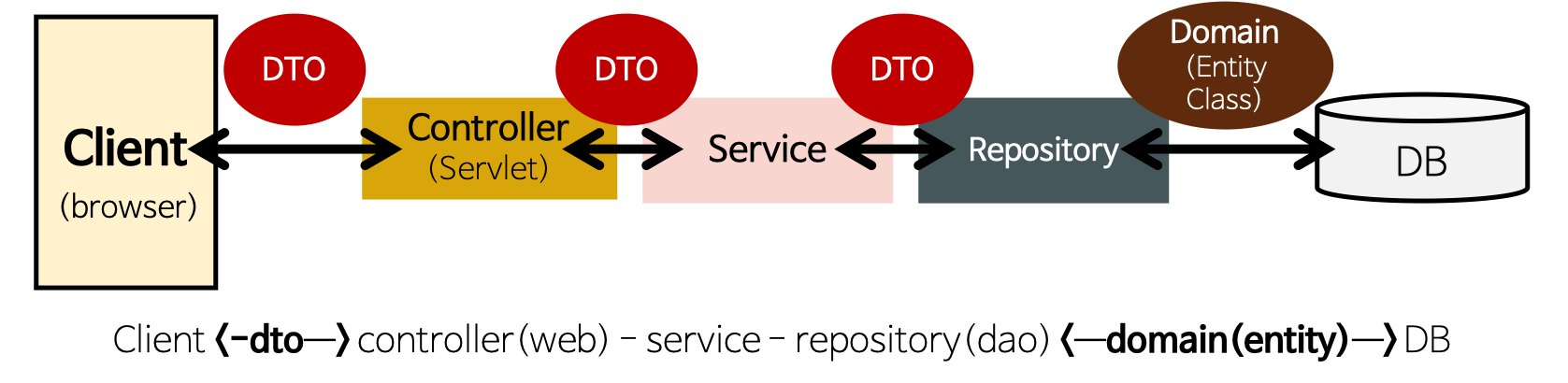


#### **폴더**



#### 

#### **설계**



VO(Value Object) 클래스

* 값을 담는 객체
* 불변성 가져야.​ 상태 변경 행위 일어나면 안됨
* Read-only
* 객체의 필드 값이 같으면, 같은 객체라고 판단
* equals()와 hashcode() 를 오버라이딩 하는 것이 핵심 역할

DTO(Data Transfer Object) 클래스

* 데이터 전송 객체
* 비동기 처리 등 값을 주고 받을 때 사용
* 필요한 속성들만 JSON 형식으로 보내야 할 경우, 데이터 가공을 위해 DTO 만듬
* **로직을 갖고 있지 않은** 순수한 데이터
* setter 대신, 생성자로 값을 할당
* 서비스 단에는 dto가 노출되지 않아야​​ 한다.
* Request와 Response용 DTO는 View를 위한 클래스
* 자주 변경이 필요한 클래스
* Presentation Model
* toEntity() 메서드를 통해서 DTO에서 필요한 부분을 이용하여 Entity로 만든다.
* 또한 Controller Layer에서 Response DTO 형태로 Client에 전달한다.

Entity 클래스

* **DB 테이블의 컬럼** 만을 필드로 가지는 클래스
* 상속 받거나 구현체이면 안됨
* setter 금지

DOMAIN 클래스

* 최대한 Entity 클래스의 Getter 를 사용하지 않도록, 필요한 로직 구현​
* 구현한 로직은 주로 Service Layer에서 사용
* Presntation Logic은 들어가지 않도록 해야함
* 필드 입력에 대한 유효성 검증 등 항상 완전한 상태로 모델이 만들어지도록 해야함
* final 키워드 사용
* DB에 저장하기 위해 유저가 정의한 클래스?

Repository

* 실제로 DB에 접근하는 객체이다.
* Service와 DB를 연결하는 고리의 역할을 한다.
* SQL를 사용(개발자가 직접 코딩)하여 DB에 접근한 후 적절한 CRUD API를 제공한다.
* JPA 대부분의 기본적인 CRUD method를 제공하고 있다.

### **- [ 단축키 ]**

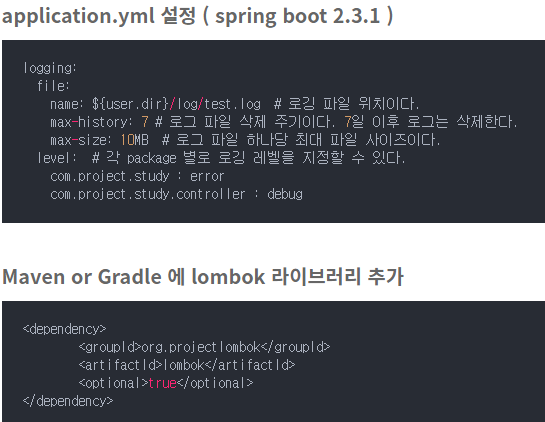
| **단축키** | **뜻** | |
| --- | --- | --- |
| sout + space | System.out.println(); | |
| psvm | public static void main | |
| shift + f10 | 실행 | |
| ctrl + shift + o | 업데이트 | |
| ctrl + alt + s | 플러그인 | |
| ctrl + i | 자동오버라이드 (implements) | |
| ctrl + o | 자동오버라이드 | |
| ctrl + b | 구현중인 곳으로 이동 | |
| alt + insert | generate | |
| ctrl + p | 자료형과 어떤 정보인지 | |
| ctrl + shift + i | 해당값 정보 | |
| ctrl + d | line 복사 | |
| ctrl + y | line 삭제 | |
| ctrl + /  ctrl + shift + / | 주석 | |
| ctrl + alt + l | 라인정리 | |
| shift + f6 | 이름바꾸기 | |

### **- [ SLF4J ]**

<https://inseok9068.github.io/java/java-lombok-slf4j/>

Simple Logging Facade For Java). Slf4j는 로깅 프레임워크가 아니다. 단지 복잡한 로깅 프레임워크들을 쉽게 사용할 수 있도록 도와주는 퍼사드(facade)에 불과하다. **퍼사드는 GoF 디자인 패턴 중 하나로서 복잡한 서브 시스템을 쉽게 사용할 수 있도록 간단하고 통일된 인터페이스를 제공**한다.

인터페이스를 사용하여 로깅을 구현하게 되면 추후에 필요로 의해 로깅 라이브러리를 변경할 때 코드의 변경 없이 가능하다.



| **Log 레벨** | **뜻** | |
| --- | --- | --- |
| Trace | 추적 레벨은 Debug보다 좀 더 상세한 정보를 나타낸다 | |
| Debug | 프로그램을 디버깅하기 위한 정보 지정 | |
| Info | 상태 변경과 같은 정보성 메세지 | |
| Warn | 처리 가능한 문제, 향후 시스템 에러의 원인이 될 수 있는 경고성 메시지 | |
| Error | 요청을 처리하는 중 문제가 발생한 경우 | |

클래스 변수로 선언하지 않고 console에 간단히 로그를 찍을 수 있다.

| import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  public class **Slf4jSample** {  private static final **Logger** log= LoggerFactory.getLogger(Slf4jSample.class);  public static void main(String[] args) {  log.info(" **Log 테스트**");  }  }  **Console >**  20200616 15:03:00.675 [main] INFO b.Slf4jSample - **Log 테스트** |
| --- |
| **@Slf4j**  public class Slf4jSample {  public static void main(String[] args) {  String value1 = " A ";  String value2 = " B ";  log.info(" **Log 테스트**"); // @Slf4j 선언하여 사용할 경우 변수명이 log로 고정된다  log.info("1번의 값은 : "+ value1 + " 2번의 값은 : "+ value2);  log.info(" 1번의 값은 : {} 2번의 값은 : {} ", value1 , value2);  }  }  **Console >**  20200616 15:36:09.144 [main] INFO b.Slf4jSample - **Log 테스트**  20200616 15:36:09.148 [main] INFO b.Slf4jSample - **1번의 값은 : A 2번의 값은 : B**  20200616 15:36:09.149 [main] INFO b.Slf4jSample - **1번의 값은 : A 2번의 값은 : B** |

## 

## **정규표현식**

정규표현식 혹은 정규식은 특정한 규칙을 가진 문자열의 집합을 표현하는데 사용되는 언어다. 정규 표현식은 텍스트 편집기나 스크립트 언어에서 문자열의 검색과 치환을 위해 지원하고 있다. 특히 Perl 같은 스크립트 언어는 언어 자체에서 강력한 정규표현식을 구현하고 있어, 문자열, 텍스트 처리에 특화된 스크립트를 작성할 수 있다.

자바 역시 프로그래머가 정규표현식을 사용할 수 있도록 여러가지 유틸을 제공하고 있다.

### **- [ Pattern 클래스 ]**

java.util.regex.Pattern 클래스는 matches() 메소드를 지원한다. 이 메소드를 이용해서 특정 문자열이 주어진 정규식에 매칭되는지를 테스트해볼 수 있다.

| String pattern = "^[0-9]\*$"; // 숫자만 등장하는지  String str = "123321";  boolean result = **Pattern**.matches(pattern, str);  System.out.println(result); // true |
| --- |

matches() 메소드의 첫 번째 인자는 정규 표현식이고, 두 번째 인자는 정규표현식이 매칭되는지 확인하고자하는 문자열이다. 만약 문자열이 정규표현식에 매칭되면 true가 리턴되고 그렇지 않으면 false가 리턴된다.

| **메소드** | **뜻** | |
| --- | --- | --- |
| compile(String regex); | 주어진 정규표현식으로부터 패턴 생성 | |
| matcher(CharSequence input); | 대상 문자열이 패턴과 일치할 경우 true 반환 | |
| asPredicate(); | 문자열을 일치시키는데 사용할 수 있는 Predicate 작성 | |
| pattern(); | 컴파일된 정규표현식을 String 형태로 변환 | |
| split(CharSequence input); | 문자열을 주어진 인자값 CharSequence 패턴에 따라 분리 | |

### **- [ Matcher 클래스 ]**

Matcher 클래스는 문자열의 패턴을 해석하고 주어진 패턴과 일치하는지 체크할 때 주로 사용한다. Pattern 클래스와 마찬가지로 Matcher 클래스는 생성자가 없다. Matcher 객체는 Pattern 객체의 matcher() 메소드를 호출해서 얻는다.

| **Pattern** pattern = **Pattern**.compile("^[0-9]\*$");  String str = "04234234";  **Matcher** matcher = pattern.matcher(str);  System.out.println(matcher.find()); |
| --- |

pattern 객체의 matcher() 메소드를 호출해서 얻어온 Matcher 클래스를 이용한 코드다. Matcher 클래스에는 다양한 메소드가 정의되어 있다.

| **메소드** | **뜻** | |
| --- | --- | --- |
| matches() | 대상 문자열과 패턴이 일치하는 경우 true 반환 | |
| find() | 대상 문자열과 패턴이 일치하는 경우 true 반환하고 그 위치로 이동 | |
| find(int start) | start 인자로 받은 위치부터 매칭 체크 | |
| start() | 매칭되는 문자열의 시작 위치를 반환 | |
| start(int group) | 지정된 그룹이 매칭되는 시작위치 반환 | |
| end() | 매칭되는 문자열의 끝 바로 다음 위치를 반환 | |
| end(int group) | 지정된 그룹이 매칭되는 끝 바로 다음 문자 위치를 반환 | |
| group() | 매칭된 부분 반환 | |
| group(int group) | 매칭된 부분 중 group번째 그루핑 매칭부분을 반환 | |
| groupCount() | 패턴내 그룹핑한 전체 갯수를 반환 | |

### **- [ 자주 사용하는 정규식 ]**

| ^ | 문자열의 시작 |
| --- | --- |
| $ | 문자열의 끝 |
| . | 임의의 한 문자 |
| \* | 문자가 0번 이상 발생 |
| + | 문자가 1번 이상 발생 |
| ? | 문자가 0번 혹은 1번 발생 |
| [ ] | 문자의 집합 범위를 나타냄. 앞에 ^가 나타나면 not을 의미  [0-9] : 숫자 (0부터 9) [a-z] : 알파벳 (a부터 z) |
| { } | 횟수 또는 범위를 의미 |
| ( ) | 소괄호 안의 문자를 하나의 문자로 인식 |
| | | or 조건 |
| \ | 확장 문자의 시작 |
| \b | 단어의 경계 |
| \B | 단어가 아닌 것의 경계 |
| \A | 입력의 시작부분 |
| \G | 이전 매치의 끝 |
| \Z | 입력의 끝이지만 종결자가 있는 경우 |
| \z | 입력의 끝 |
| \s | 공백문자 |
| \S | 공백문자가 아닌 나머지 문자 |
| \w | 알파벳이나 숫자 |
| \W | 알파벳이나 숫자를 제외한 문자 |
| \d | [0-9]와 동일 |
| \D | 숫자를 제외한 모든 문자 |

| **정규표현식** | **뜻** | |
| --- | --- | --- |
| ^[0-9]\*$ | 숫자 | |
| ^[a-zA-Z]\*$ | 영문자 | |
| ^[가-힣]\*$ | 한글 | |
| \\w+@\\w+\\.\\w+(\\.\\w+)? | 이메일 주소 | |
| ^\d{2,3}-\d{3,4}-\d{4}$ | 전화번호 | |
| ^01(?:0|1|[6-9])-(?:\d{3}|\d{4})-\d{4}$ | 핸드폰 번호 | |
| \d{6} \- [1-4]\d{6} | 주민등록 번호 | |
| ^\d{3}-\d{2}$ | 우편번호 | |

## **@Valid**

java.validation의 @Valid 어노테이션 사용법. Spring Boot 라이브러리에서 기본적으로 탑재된 기능이라 따로 dependency해 줄 필요가 없다. (Spring Boot Version은 2.2.2.RELEASE)

| **어노테이션** | **제약조건** | |
| --- | --- | --- |
| @NotNull | Null 불가 | |
| @Null | Null만 입력 가능 | |
| @NotEmpty | Null, 빈 문자열 불가 | |
| @NotBlank | Null, 빈 문자열, 스페이스만 있는 문자열 불가 | |
| @Size(min=,max=) | 문자열, 배열등의 크기가 만족하는가? | |
| @Pattern(regexp=) | 정규식을 만족하는가? 아래 비밀번호 예문! | |
| @Pattern(regexp = "^( ?=.\*[a-zA-Z] )( ?=.\*\\d )( ?=.\*[~!@#$%^&\*\_] )[a-zA-Z\\d-!@#$%^&\*\_]{8,16}$")  // () or () or () 중에 포함되어야 하며 [] 만 사용가능하며 {} 글자수제한 | | |
| @Max(숫자) | 지정 값 이하인가? | |
| @Min(숫자) | 지정 값 이상인가 | |
| @Future | 현재 보다 미래인가? | |
| @Past | 현재 보다 과거인가? | |
| @Positive | 양수만 가능 | |
| @PositiveOrZero | 양수와 0만 가능 | |
| @Negative | 음수만 가능 | |
| @NegativeOrZero | 음수와 0만 가능 | |
| @Email | 이메일 형식만 가능 | |
| @Digits(integer=, fraction = ) | 대상 수가 지정된 정수와 소수 자릿수 보다 작은가? | |
| @DecimalMax(value=) | 지정된 값(실수) 이하인가? | |
| @DecimalMin(value=) | 지정된 값(실수) 이상인가? | |
| @AssertFalse | false 인가? | |
| @AssertTrue |  | |

### **- [ BindingResult ]**

**BindingResult**가 스프링이 제공하는 검증 오류 처리의 핵심이다. 이유는 BindingResult는 스프링이 제공하는 **검증 오류를 보관하는 객체**이기 때문이다.

**데이터 유효성 검사를 실패**하면 **ConstraintViolationException을 발생**시킨다. 데이터가 유효하지 않은 속성이 있으면 그에 대한 에러 정보를 BindingResult에 담는다.

정상적인 동작에서는 BindingResult에 담은 오류 정보를 가지고 컨트롤러를 호출한다. 하지만 BindingResult가 없다면 4xx 오류가 발생하면서 컨트롤러가 호출되지 않고 오류 페이지로 이동한다.

* @ModelAttribute의 객체에 타입 오류 등으로 바인딩이 실패하는 경우 스프링이 FieldError을 생성해서 BindingResult에 넣어준다.
* new FieldError를 만들어서 개발자가 직접 넣어준다.
* Validator를 사용(@Valid, @Validated)

보통은 @ModelAttribute를 검증하기 때문에 BindingResult를 @ModelAttribute 뒤에 두고 사용한다.

#### **BindingResult 메소드**

BindingResult 의 첫 번째 사용 방식으로 메소드를 사용하는 방식이 있다.

| **메소드** | **뜻** | |
| --- | --- | --- |
| boolean hasErrors() | 에러의 유무를 판단한다. | |
| boolean hasGlobalErrors() | 글로벌 에러의 유무를 판단한다. | |
| void addError(ObjectError error) | field, type, value 등의 에러를 출력할 수 있다. | |
| void rejectValue() | field, errorCode, defaultMessage 등을 받아서 reject 됐을 때 데이터를 남길 수 있다. | |

| **전달 파라미터** | **값** | |
| --- | --- | --- |
| objectName | 오류가 발생한 객체의 이름 | |
| field | 오류 필드 | |
| rejectedValue | 사용자가 입력한 값(거절된 값) | |
| bindingFailure | 타입 오류 같은 바인딩 실패인지, 검증 실패인지 구분 값 | |
| codes | 메시지 코드 | |
| arguments | 메시지에서 사용하는 인자 | |
| defaultMessage | 기본 오류 메시지 | |

#### **Validator**

두 번째 방법으로 Validator를 사용하는 방식도 존재한다.

@Valid 혹은 @Validated 애노테이션을 이용해서 **데이터 유효성 검증**을 할 수 있다. @Valid는 자바 표준 검증 애노테이션이고, @Validated는 스프링 전용 검증 애노테이션이다. @Valid, @Validated는 거의 동일한 역할을 한다..

정확히 말하면, @Validated는 group 기능이 추가된 @Valid이다.

하지만 @Validated의 group 기능은 잘 쓰지는 않는다. 이유는 데이터를 한곳에서 가지고 오지 않고, 다른 데이터를 함께 사용하기 때문이다. 그리고 **Bean Validation이라는 데이터 유효성 검사 프레임워크를 사용**할 수도 있다.

다양한 제약 조건을 도메인 모델에 애노테이션을 사용해서 정의할 수 있다.

이 제약 조건을 유효성 검사가 필요한 객체에 직접 정의하는 방법으로 기존 유효성 검사 로직의 문제를 해결한다. 주의할 점은 데이터 유효성 감사를 진행할 때 검사가 중복으로 실행되지 않도록 해야 한다. 같은 데이터 유효성 검사가 여러 번 실행될 경우 애플리케이션 성능이 저하될 수 있기 때문이다.

| **제약조건 어노테이션** | **뜻** | |
| --- | --- | --- |
| @Length | 길이를 설정할 수 있다. | |
| @NotBlank | 빈 문자열을 막을 수 있다 | |
| @NotNull | 널을 방지할 수 있다. | |
| @NotEmpty | 해당 값이 Null이 아니고, 빈 스트링("")이 아닌지 검증함 | |
| @Range | 범위를 설정할 수 있다. | |
| @Max / @Min | 최대/최소를 설정할 수 있다. | |
| @Email | 이메일 형태의 데이터를 확인한다.(@ 포함 여부 확인) | |

#### **타임리프**

마지막으로, 타임리프에서도 검증 오류를 쉽게 표현하는 기능을 제공한다.

| **코드** | **뜻** | |
| --- | --- | --- |
| $fields | BindingResult가 제공하는 검증 오류에 접근할 수 있다.  th:field="\*{itemName}"  -> id, name, value를 알아서 만들어준다. | |
| th:errors | 해당 필드에 오류가 있는 경우에 태그를 출력한다. th:if의 편의 버전  th:errors="\*{itemName}"  -> 해당 필드에 오류가 있는 경우에 태그를 출력한다. | |
| th:errorclass | th:field에서 지정한 필드에 오류가 있으면 class 정보를 추가한다.  th:errorclass="field-error"  -> 오류가 나면 해당 클래스를 출력하겠다. | |

### **- [ 예문1 ]**

[post] /member요청 시, member를 저장하는 코드이다.

**@Valid** 어노테이션을 통해 **Member가 유효한 객체**인지 **검사**한다. Member 객체가 유효하지 않으면 **bindingResult.hasErrors()** 메소드에서 **true 값이 반환**된다. Member가 유효하지 않다면, **ResponseEntity에 BAD\_REQUEST와 bindingResult.getAllErrors() 값을 넣고 생성하여 반환**한다.

| // Member.java  public class **Member** {  **@NotNull**(message = "id는 필수 값입니다.")  **@Size**(min = 5, max = 10)  private String id;  **@Max**(value = 25, message = "25세 이하만 가능합니다.")  **@Min**(value = 18, message = "18살 이상만 가능합니다.")  private int age;  **@Pattern**(regexp = "[a-zA-z0-9]+@[a-zA-z]+[.]+[a-zA-z.]+")  private String email;    ..setter/getter 생략..  }  // TestController.java  @Controller  @RequestMapping(value = "/member")  public class TestController {    @PostMapping  @ResponseBody  public **ResponseEntity** saveMember(**@Valid** Member member, BindingResult bindingResult) {    if (bindingResult.hasErrors()) {  return ResponseEntity.status(**HttpStatus.BAD\_REQUEST**).body(**bindingResult.getAllErrors()**);  }    /\*  save Memeber  \*/    return ResponseEntity.ok(member);  }  } |
| --- |

### **- [ 예문2 ]**

| public class Item {  private Long id;  **@NotBlank**  private String itemName;  **@NotNull @Range**(min = 1000, max = 1000000)  private Integer price;  **@NotNull @Max**(value = 9999)  private Integer quantity;  }  @PostMapping("/add")  public String add(**@Validat** **@ModelAttribute** Item item, **BindingResult** bindingResult) {    if (**bindingResult**.hasErrors()) {  return "error"; //보통 되돌리기  }  return "success"; 성공 페이지로 이동 혹은 redirect로 원하는 페이지로 이동    }  <form action="item.html" th:action th:object="${item}" method="post">  <div>  <label for="itemName" >상품명</label>  <input type="text" id="itemName" th:field="\*{itemName}"  th:errorclass="field-error" placeholder="이름을 입력하세요">    <div class="field-error" th:errors="\*{itemName}">상품명 오류</div>  </div>  </form> |
| --- |

### **- [ Validation Groups ]**

**우선순위를 적용**해서 하나의 검증이 실패하면 그 메시지만 리턴하고 검증을 통과하면 그 다음 항목을 검증하는 식으로 진행할 수 있도록 돕는다

| public class UrlDto {  public static class **Request**{  @Data  public static class **Save**{  **@NotBlank**(message = "Link 필드는 필수입니다.")  **@Pattern**(regexp="^((http(s?))\\:\\/\\/)([0-9a-zA-Z\\-]+\\.)+[a-zA-Z]{2,6}(\\:[0-9]+)?(\\/\\S\*)?$",  message = "Link 형식이 유효하지 않습니다.")  private String url;  }  }  RequestBody    ResponseBody: |
| --- |

**1순위로 NotBlank를 검사**하고, **2순위로 정규표현식을 검사**해서 그에 맞는 메시지만을 리턴하게 만들어야 한다. 수행한 방법은 아래순서와 같습니다.

#### **인터페이스 생성**

먼저 인터페이스를 생성해줍니다. 따로 만들면 파일이 늘어나니, 클래스 하나를 만들고 그안에 정의하는 방식을 선택했습니다.

| public class **ValidationGroups** {  public interface NotEmptyGroup {};  public interface PatternCheckGroup {};  } |
| --- |

#### **@GroupSequence 사용**

@GroupSequence를 사용하여 그룹별 인터페이스를 정의해줍니다. 왼쪽부터 유효성 검사를 체크해서 오류가 없으면 다음 유효성 검사를 하는 순서로 진행이 됩니다

**Default.class** -> **NotEmptyGroup.class** -> **PatternCheckGroup.class**

| import foo.study.url.dto.ValidationGroups.NotEmptyGroup;  import foo.study.url.dto.ValidationGroups.PatternCheckGroup;  import javax.validation.GroupSequence;  import javax.validation.groups.Default;  **@GroupSequence**({Default.class, NotEmptyGroup.class, PatternCheckGroup.class, })  public interface **ValidationSequence** {  } |
| --- |

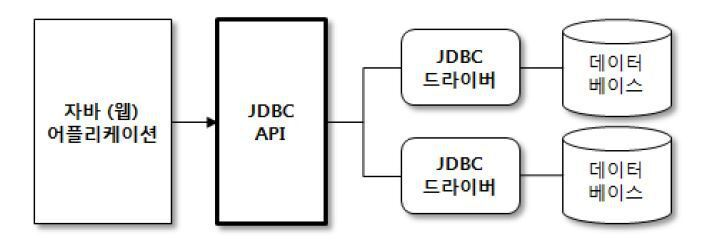
#### **groups = "인터페이스명"을 추가**

Validaton 어노테이션에서 각각 groups = "인터페이스명"을 추가해줍니다.

| public class UrlDto {  public static class **Request**{  @Data  public static class Save{  **@NotBlank**(message = "Link 필드는 필수입니다.",  **groups =** ValidationGroups.NotEmptyGroup.class)  **@Pattern**(regexp="^((http(s?))\\:\\/\\/)([0-9a-zA-Z\\-]+\\.)+[a-zA-Z]{2,6}(\\:[0-9]+)?(\\/\\S\*)?$",  message = "Link 형식이 유효하지 않습니다.",  **groups** = ValidationGroups.PatternCheckGroup.class)  private String url;  } |
| --- |

## **JDBC**

Java로 각 관계형 데이터베이스들에 접속하고 SQL문을 수행하는 처리 동작을 수행할 때 사용되는 표준 SQL Interface API이다. JDBC를 사용하면 개발자는 DBMS마다의 복잡하고 고유한 동작들을 모두 알 필요가 없어진다. 단지 JDBC가 추상화 인터페이스를 제공하면 개발자는 JDBC 드라이버를 통해 각각의 DBMS에 대응만 하면 DB에 접근할 수 있게 된다.



JDBC 드라이버에 로딩하기 위해 각 DB의 드라이버 클래스 네임, 연결하기 위해 URL 포맷들을 알고 있어야 한다.

| **DB** | **Driver Class** | |
| --- | --- | --- |
| **MariaDB** | org.mariadb.jdbc.Driver  - MySQL 과 같은 브랜치로부터 릴리즈되며, 대개의 경우 MySQL과 동일하게 동작한다. MySQL의 모든 명령어, 인터페이스, 라이브러리와 API 가 MariaDB 에도 존재한다. 또한 MariaDB 로 데이터베이스를 변환할 필요도 없다. 즉, MariaDB 는 사실상 MySQL 의 완벽한 대체재 라고 말할 수 있다. | |
| **MySQL** | com.mysql.cj.jdbc.Driver  - 과거 무료였으나 오라클로 인수되며 상업용은 유료로 전환. 이후 대체재로 MariaDB가 나옴 | |
| Oracle | oracle.jdbc.driver.OracleDriver | |
| SQL Server | com.microsoft.sqlserver.jdbc.SQLServerDriver | |
| Tibero | com.tmax.tibero.jdbc.TbDriver | |
| Altibase | Altibase.jdbc.driver.AltibaseDriver | |
| Postgresql | postgresql.driver | |
| Sybase | com.sybase.jdbc2.jdbc.SybDriver | |

### **- [ MyBatis ]**

JDBC 라이브러리이다. (JDBC에 종속. 설정에 프로퍼티로 jdbc설정)

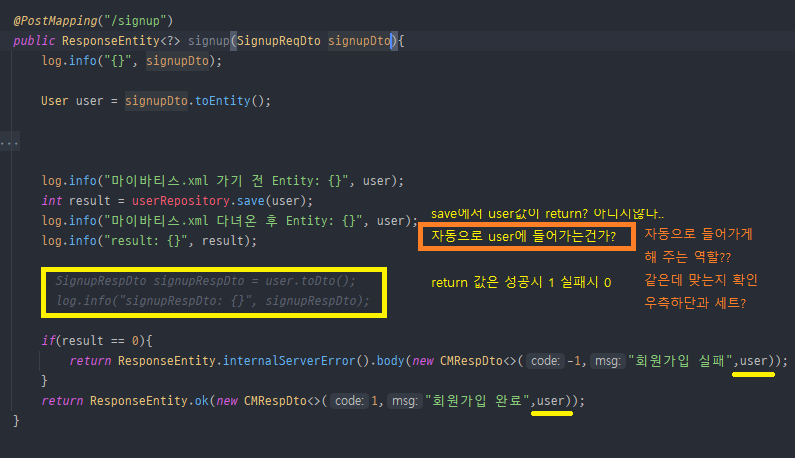
Java에는 관계형 데이터베이스와 함께 작동하는 JDBC(Java DataBase Connectivity) API가 있다. 그러나 JDBC는 매우 낮은 수준의 API이며 데이터베이스 작업을 수행하기 위해 많은 코드를 작성해야 한다.

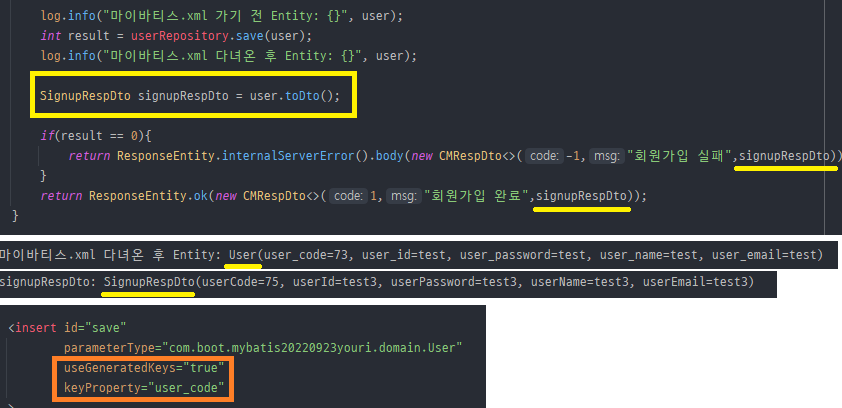
MyBatis는 이러한 모든 공통 작업을 추상화하여 개발자가 실행해야 하는 SQL 문을 준비하고 입력 데이터를 Java 개체로 전달하는 것과 같은 정말 중요한 측면에 집중할 수 있도록 한다. Java 및 SQL에 대한 지식에 의존하기 때문에 배우고 사용하기가 매우 간단하다.

| **mybatis-config.xml 설정 파일 생성 [위치 : Java Resources/src/main/resources/]** [참고링크](https://study-ce.tistory.com/35#toc-%3C%ED%99%9C%EC%9A%A9%ED%95%98%EA%B8%B0%3E) |
| --- |
| 해당 파일에 아래의 코드 추가 // xml 유효성 체크를 위해 필요 but 스프링에서는 필요X  // 스프링에서는 config는 자동으로 잡아줌.  <!DOCTYPE configuration PUBLIC  "-//mybatis.org/DTD Config 3.0/EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> |

| **@Mapper 파일 생성 [위치 :** com.boot.mybatis20220923youri.repository.**UserRepository]** |
| --- |
| **@Mapper** //news.xml과 연결되어있다. 이게 impl 역할을 한다.  // IOC 등록역할도 해준다. 그러므로 imlp에 하던 어노테이션을 기입할 필요 X  public interface **NewsRepository** {  public int save(**News** news); //news.xml로 던져준다. (단축키: **art + enter** )  } |

| **mybatis-mapper.xml 설정 파일 생성 [위치 : Java Resources/src/main/resources/mappers]** |
| --- |
| 해당 파일에 아래의 코드 추가 // 쿼리를 추가하는 파일  <!DOCTYPE mapper PUBLIC  "-//mybatis.org/DTD Mapper 3.0/EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> |
| **- 활용 예시 코드** (해당 파일 아래에 바로 이어 기입) |
| <**mapper namespace**="com.boot.mybatis20220923youri.repository.**UserRepository**">  <**insert** id="save" // UserRepository에 있던 save이다.  **parameterType**="com.boot.mybatis20220923youri.domain.User"  **useGeneratedKeys**="true" // 자동으로 만들어진 Key값을 쓰겠다.  **keyProperty**="user\_code" // ↑ 를 user\_code에 쓰겠다.  >  **insert into**  user\_mst  values (  0,  #{user\_id},  #{user\_password}  )  </**insert**>  <**/mapper**> |





## **Spring MVC**

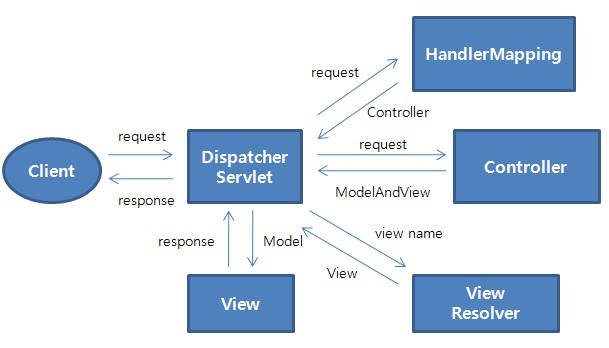
Spring Framework는 간단한 설정만으로 Struts나 Webwork같은 Web Framework을 사용할 수 있지만, 자체적으로 MVC Web Framework을 가지고 있다. Spring MVC는 기본요소인 Model, View, Controller 외에도, 아래와 같은 특성을 가지고 있다.

1. 각 컴포넌트의 역할이 명확하게 분리되어 있다.  
   (DispatcherServlet, HandlerMapping, Controller, Interceptor, ViewResolver, View등)
2. 컴포넌트들에 다양한 인터페이스 및 구현 클래스를 제공함으로써 경우에 따라 선택하여 사용할 수 있다. (HandlerMapping, Controller, View등)
3. Controller(@MVC)나 폼 클래스(커맨드 클래스) 작성시에 특정 클래스를 상속받거나 참조할 필요 없이 POJO 나 POJO-style의 클래스를 작성함으로써 비즈니스 로직에 집중된 코드를 작성할 수 있다.
4. 웹요청 파라미터와 커맨드 클래스간에 데이터 매핑 기능을 제공한다.
5. 데이터 검증을 할 수 있는, Validator와 Error 처리 기능을 제공한다.
6. JSP Form을 쉽게 구성하도록 Tag를 제공한다.

### **- [ 구성 ]**

Spring MVC(Model-View-Controller)의 핵심 Component는 아래와 같다.

| **Component** | **개요** | |
| --- | --- | --- |
| DispatcherServlet | Spring MVC Framework의 Front Controller, 웹요청과 응답의 Life Cycle을 주관한다. | |
| HandlerMapping | 웹요청시 해당 URL을 어떤 Controller가 처리할지 결정한다. | |
| Controller | 비즈니스 로직을 수행하고 결과 데이터를 ModelAndView에 반영한다. | |
| ModelAndView | Controller가 수행 결과를 반영하는 Model 데이터 객체와 이동할 페이지 정보(또는 View객체)로 이루어져 있다. | |
| ViewResolver | 어떤 View를 선택할지 결정한다. | |
| View | 결과 데이터인 Model 객체를 display한다. | |



1. Client의 요청이 들어오면 DispatchServlet이 가장 먼저 요청을 받는다.

2. HandlerMapping이 요청에 해당하는 Controller를 return한다.

3. Controller는 비즈니스 로직을 호출하고 결과 데이터를 ModelAndView에 반영 return한다.

4. ViewResolver는 view name을 받아 해당하는 View 객체를 return한다.

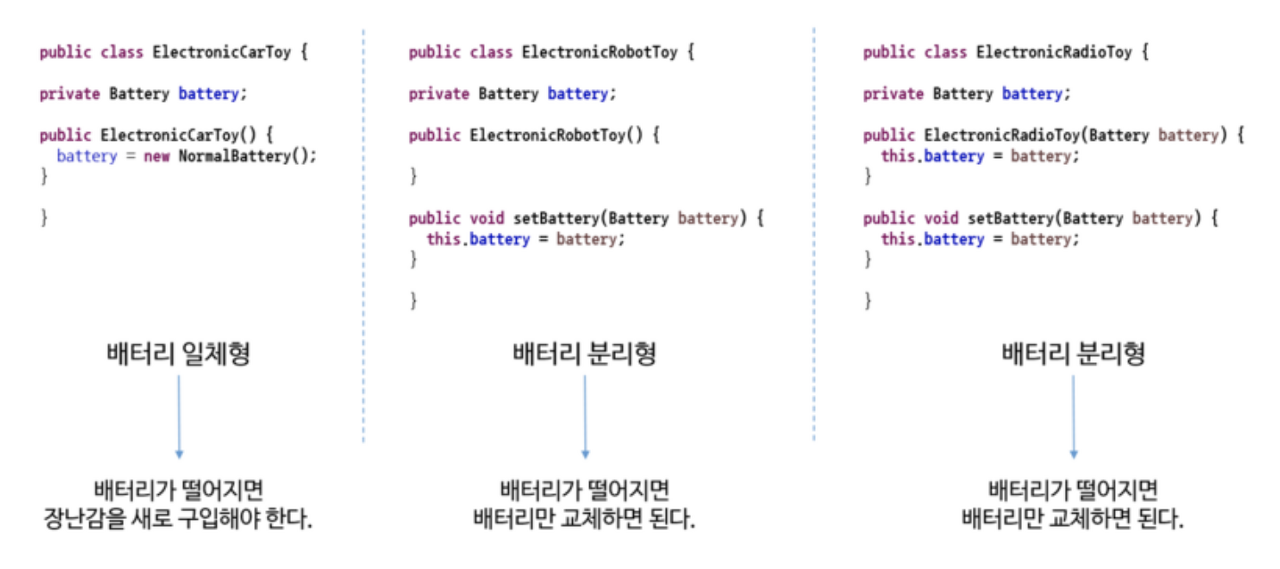
5. View는 Model 객체를 받아 rendering한다.

### **- [ 의존성 주입(DI) ]**

Inversion of Control 이라고도 하는 의존 관계 주입(Dependency Injection)이라고도 하며, 어떤 객체가 사용하는 의존 객체를 직접 만들어 사용하는게 아니라, **주입 받아 사용**하는 방법이다. (new 연산자를 이용해서 객체를 생성하는 것이라고 생각하면 된다)



장난감들은 베터리가 있어야 움직일 수 있으며 즉 베터리에 의존하고 있다. 장난감들에게 베터리를 넣어주는 것을 의존성 주입이라고 생각하면 좋다.

베터리의 일체형인 경우에는 생성자에서만 의존성을 주입해주는 상황이라 배터리가 떨어지면 배터리를 교체하지 못하고 새로운 것으로 바꿔야 하기 때문에 유연하지 못하다.

setter, 생성자를 이용해서 외부에서 주입해주는 상황은 외부에서 베터리를 교체해줄 수 있기 때문에 일체형보다 유연한 상황이다. **객체를 주입 받는다**는 것은 **외부에서 생성된 객체를 인터페이스를 통해서 넘겨받는 것**이다. 이렇게 하면 **결합도를 낮출 수 있고, 런타임시에 의존관계가 결정되기 때문에 유연한 구조를 가진다**.

#### **IoC 컨테이너**

과거에는 자바로 웹 어플리케이션을 개발하기 위해서는 Servlet 클래스를 상속받아서 사용했다. 이 Servlet 클래스는 POJO가 아닌 것이다. 즉 Servlet 클래스를 작성하지 않고 POJO(Plain Old Java Object, 종속되지 않은 객체지향설계)만으로 웹 어플리케이션을 개발할 수 있다는 것이 스프링의 특징이다.

| 1. **재사용성**을 높여준다.  2. 테스트에 용이하다.  3. 코드를 **단순화** 시켜준다.  4. 사용하는 이유를 파악하기 수월하고 **코드가 읽기 쉬워지는 점**이 있다.  5. **종속성이 감소**하기 때문에 변경에 민감하지 않다.  6. 결합도(coupling)는 낮추면서 **유연성과 확장성은 향상** 시킬 수 있다.  7. 객체간의 **의존관계를 설정**할 수 있다. |
| --- |

가장 중요한 인터페이스는 BeanFactory, ApplicatonContext이다. 애플리케이션 컴포넌트의 중앙 저장소이다. 빈 설정 소스로 부터 빈 정의를 읽어들이고, 빈을 구성하고 제공한다. 빈들의 의존 관계를 설정해준다.(객체의 생성을 책임지고, 의존성을 관리한다)

| **인터페이스 호출 방식** | **IoC** |
| --- | --- |
|  |  |

빈(Bean)이란 스프링 IoC 컨테이너가 관리하는 객체이다.

| 스프링 IoC 컨테이너에 등록된 Bean들은 **의존성 관리가 수월**해진다.  스프링 IoC 컨테이너에 등록된 Bean들은 **싱글톤의 형태**이다 |
| --- |

이러한 장점들 때문에 우리가 직접 Inversion of Control의 코드를 작성해서 사용하는 것이 아니라 스프링의 IoC 컨테이너를 사용하는 것이다.

#### **Bean 등록**

Spring-Boot는 어노테이션을 통해 Bean을 설정하고 주입 받는 것을 표준으로 삼는다. Container에 Spring Bean으로 등록시켜주는 Annotation은 아래와 같다.

| **Annotation** | **용도** |
| --- | --- |
| @Bean | 개발자가 컨트롤 할 수 없는 외부 라이브러리 Bean으로 등록하고 싶은 경우 (메소드로 return 되는 객체를 Bean으로 등록) |
| @Component | 개발자가 직접 컨트롤할 수 있는 클래스(직접 만든)를 Bean으로 등록하고 싶은 경우 (선언된 Class를 Bean으로 등록) |
| @Controller, @Service, @Repository  기타 등등 | @Component를 비즈니스 레이어에 따라 명칭을 달리 지정해준 것. Container에 있는 Spring Bean을 찾아 주입시켜주는 Annotation |

위의 어노테이션으로 Container를 등록하면, @Autowired 가 IoC 컨테이너에 있는 참조할 Bean을 찾아 주입한다.(SPRING 표준)

### **- [ @RequiredArgsConstructor ]**

생성자주입의 단점은 생성자를 만들기 번거롭다는 점이다. 하지만 이를 보완하기위해 롬복을 사용하여 간단하게 생성자 주입 방식의 코딩을 할 수 있다. 대표적으로 **@Autowired**가 있으나 이는 일일이 설정 해주어야 한다는 단점이 있다.

| **@RequiredArgsConstructor** | **@RequiredArgsConstructor**  public class NewsController {  private **final** NewsRepository newsRepository1;  private **final** NewsRepository newsRepository2;  // 상수는 무조건 선언 후에 초기화를 해주어야한다.  } |
| --- | --- |
| **@Autowired** | public class NewsController {  **@Autowired**  private NewsRepository newsRepository1;  **@Autowired**  private NewsRepository newsRepository2;  } |

@RequiredArgsConstructor은 final이나 @NotNull 이 붙은 필드의 생성자를 자동 생성해준다.

@Autowired 는 하나하나 다 달아야한다.

### **- [ @Autowired ]**

\* <https://devlog-wjdrbs96.tistory.com/165> 참고 (SPRING 표준)

필요한 의존 객체의 “타입"에 해당하는 (IoC 컨테이너에 있는 참조할) Bean을 찾아 주입한다.

| 생성자, setter, 필드 |
| --- |

위의 3가지의 경우에 Autowired를 사용할 수 있다. 그리고 Autowired는 기본값이 true이기 때문에 의존성 주입을 할 대상을 찾지 못한다면 애플리케이션 구동에 실패한다.

#### **생성자**

생성자 주입은 생성자에 의존성 주입을 받고자 하는 field를 나열하는 방법으로, 권고되는 방법의 하나이다. 자세한 내용은 **의존성 주입 파트**\* 참고.

| **장점** | 필수적으로 사용해야 하는 레퍼런스 없이는 인스턴스를 만들지 못하도록 강제함  Spring 4.3 이상부터는 생성자가 하나인 경우 @Autowired를 사용하지 않아도 됨  Circular Dependency / 순환 참조2 의존성을 알아 차릴 수 있음  생성자에 점차 많은 의존성이 추가 될 경우 리팩토링 시점을 감지 할 수 있음  의존성 주입 **대상 필드를 final**로 불편 객체 선언할 수 있음  테스트 코드 작성시 생성자를 통해 의존성 주입이 용이함 |
| --- | --- |
| **단점** | 어쩔 수 없는 순환 참조는 생성자 주입으로 해결하기 어려움  이러한 경우에는 나머지 주입 방법 중에 하나를 사용  가급적이면 순환 참조가 발생하지 않도록 하는 것이 더 중요 |

\* @RequiredArgsConstructor 를 사용하여 해당 기능 대체 가능하다.

#### **필드**

member field에 @Autowired annotation을 선언하여 주입받는 방법이다.

| **장점** | 가장 간단한 선언 방식 |
| --- | --- |
| **단점** | 의존 관계가 눈에 잘 보이지 않아 추상적이고, 이로 인해 의존성 관계가 과도하게 복잡해질 수 있음  반대로 Constructor injection과 Setter injection은 의존성을 명확히 커뮤니케이션 함  이는 SRP / 단일 책임 원칙에 반하는 안티패턴  DI Container와 강한 결합을 가져 외부 사용이 용이하지 않음  단위 테스트시 의존성 주입이 용이하지 않음  의존성 주입 대상 필드가 final 선언 불가 |

#### **setter**

setter 메소드에 @Autowired annotation을 선언하여 주입받는 방법이다.(메소드 이름을 setter 대신에 다른 걸로 하여도 주입은 가능하지만 좋은 방법은 아니다)

| **장점** | 의존성이 선택적으로 필요한 경우에 사용  생성자에 모든 의존성을 기술하면 과도하게 복잡해질 수 있는 것을 선택적으로 나눠 주입 할 수 있게 부담을 덜어줌  생성자 주입 방법과 Setter 주입 방법을 적절하게 상황에 맞게 분배하여 사용 |
| --- | --- |
| **단점** | 의존성 주입 대상 필드가 final 선언 불가 |

#### **빈이 X 경우**

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  @Service  public class **BookService** {  BookRepository bookRepository;  **@Autowired**  public **BookService** (BookRepository bookRepository) {  **this**.bookRepository = bookRepository;  }  } |
| --- |

생성자에 대해 Autowired 어노테이션이 있는 상태이다. BookRepository가 빈으로 등록되어 있지 않다고 가정하면 생성자에서 에러가 발생한다. 이유는 Autowired가 의존성을 주입하기 위해서는 빈이 등록되어 있는 객체중에서 찾는데 BookRepository라는 객체는 빈으로 등록되어 있지 않기 때문에 의존성 주입에 실패해서 에러가 발생한다.

Autowired의 기본값이 true이기 때문이다.

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  @Service  public class **BookService** {  BookRepository bookRepository;  **@Autowired**  public void **setBookRepository**(BookRepository bookRepository) {  this.bookRepository = bookRepository;  }  } |
| --- |

현재 BookRepository는 빈으로 등록되어 있지 않다고 가정하고, Autowired가 붙은 setter에서 여전히 에러가 발생한다. 여기서 하나 궁금한 것은 이번에는 setter인데 BookService의 빈은 등록할 수 있지 않나? 맞다 BookSevice는 빈으로 등록할 수 있다. 그리고 Autowired가 기본값이 true라고 했는데 그러면 false로 하면 어떻게 될까?

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  @Service  public class **BookService** {  BookRepository bookRepository;    **@Autowired(required = false)**  public void **setBookRepository**(BookRepository bookRepository) {  this.bookRepository = bookRepository;  }  } |
| --- |

위와 같이 쓴다면 BookService의 객체는 빈으로 등록이 되지만 BookRepository는 빈으로 등록되지 않게 된다. (required = false이기 때문이다) 그리고 Autowired를 사용할 수 있는 곳이 필드가 하나 더 있다.

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  @Service  public class **BookService** {  **@Autowired(required = false)**  BookRepository bookRepository;  } |
| --- |

위와 같이 필드에도 사용할 수 있다. setter와 필드에 Autowired를 사용하게 되면 만약 BookRepository가 빈으로 등록되어 있지 않다 해도 BookService는 빈으로 등록이 가능하다. 하지만 생성자에 Autowired를 쓴 상황에서 BookRepository가 빈으로 등록되어 있지 않다면 BookService도 빈으로 등록되지 못하는 경우가 생긴다.

#### **빈이 여러개인 경우**

만약 **BookRepository**라는 인터페이스 아래에 MyBookRepository, YourBookRepository라는 클래스가 존재하고 둘다 빈으로 등록이 되어 있을 때 바로 위의 코드처럼 BookRepository 필드에 Autowired를 적용하면 어떻게 될까? 주입을 해줄 수 없게 된다. 왜냐하면 등록해야 할 빈이 2개이기 때문에 스프링은 어떤걸 해야할지 모르기 때문이다.

| **@Repository @Primary**  public class **MyBookRepository implements BookRepository**{  // test  } |
| --- |

따라서 만약 같은 타입의 빈이 위의 상황처럼 여러개 일 때 내가 만약 MyBookRepository를 빈으로 등록하고 싶다면 **@Primary** 라는 어노테이션을 추가해주면 MyBookRepository가 빈으로 등록된다.

또 다른 방법으로는 **@Qualifier**라는 어노테이션을 이용하는 것인데 일반적으로 빈의 이름은 클래스 이름의 스몰케이스이다. 아래의 예시를 보자.

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Qualifier;  import org.springframework.stereotype.Service;  @Service  public class **BookService** {  **@Autowired @Qualifier("a")** // 이때 impl쪽에도 @Repository("a") 달아줘야함  **BookRepository** bookRepository;  } |
| --- |

이런식으로 Autowired 옆에 **@Qualifier**라는 어노테이션을 이용하여 괄호안에는 등록하고자 할 빈의 이름을 적어주면 된다 하지만 **@Qualifier**보다 **@Primary**가 더 안전한 방법이기 때문에 **@Primary**를 더 추천한다. 그리고 마지막으로는 해당타입의 빈을 모두 주입받는 경우도 있다.

| import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.stereotype.Service;  import java.util.List;  @Service  public class **BookService** {  **@Autowired**  List<**BookRepository**> bookRepositoryList;  } |
| --- |

위와 같이 List를 이용하면 BookRepository에 해당하는 타입의 빈을 모두 주입받을 수 있다. 여러 방법중에서 제일 추천하고 싶은 방법은 @Primary이다.

### **- [ @Configuration ]**

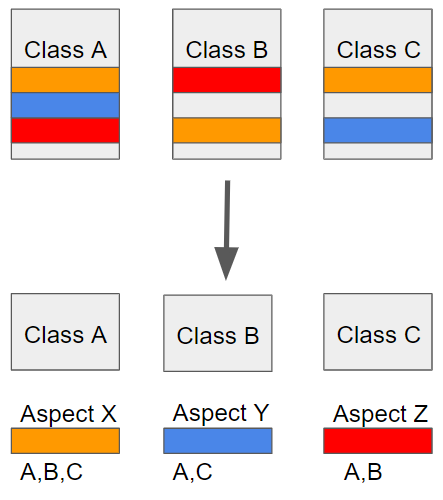
아래는 폴더에 저장된 이미지 파일을 불러오기 위한 세팅이다.

| **@Configuration**  public class **WebConfig** implements **WebMvcConfigurer** {  **@Value**(" ${file.path} ") // yml에 해당 경로를 잡아 두었다.  private String filePath;  **@Override**  public void **addResourceHandlers**(ResourceHandlerRegistry registry) {  // 요청과 응답에 대한 웹 설정  // Spring Boot 프로젝트 외부 경로에 있는 파일 접근할 수 있도록 도와준다.  WebMvcConfigurer.super.addResourceHandlers(registry); //default 세팅.  registry**.addResourceHandler**(" /image/\*\* ")  // http://localhost:8000/image/\*\* 까지의 경로.  // 리소스와 연결될 URL path를 지정합니다. (클라이언트가 파일에 접근하고자 요청하는 url)  **.addResourceLocations**(" file:/// " + filePath)  // 구체적 위치. 실제 리소스가 존재하는 경로.  // 경로의 마지막은 반드시 " / "로 끝나야 하며,  // 로컬 디스크 경로일 경우 file:/// 접두어를 꼭 붙여야 한다.  **.setCachePeriod(60 \* 60)** // 1초 \* 60 = 1시간 동안 캐시를 유지하라  **.resourceChain(true)** //연결시켜주겠다  **.addResolver**(new PathResourceResolver(){ //한글때문에 UTF\_8 필요. 그런 역할  **@Override**  protected Resource getResource(String resourcePath, Resource location) throws IOException {  resourcePath = URLDecoder.decode(resourcePath, StandardCharsets.UTF\_8);  **return** super**.getResource**(resourcePath , location);  }  });  }  } |
| --- |

## **Spring AOP**

AOP는 Aspect Oriented Programming의 약자로 **관점 지향 프로그래밍**이라고 불린다. 관점 지향은 쉽게 말해 **어떤 로직을 기준으로 핵심적인 관점, 부가적인 관점으로 나누어서 보고 그 관점을 기준으로 각각 모듈화하겠다는 것**이다. 여기서 모듈화란 어떤 공통된 로직이나 기능을 하나의 단위로 묶는 것을 말한다.

AOP에서 각 관점을 기준으로 로직을 모듈화한다는 것은 **코드들을 부분적으로 나누어서 모듈화**하겠다는 의미다. 이때, 소스 코드상에서 **다른 부분에 계속 반복해서 쓰는 코드**들을 발견할 수 있는 데 이것을 **흩어진 관심사** (Crosscutting Concerns)라 부른다.



위와 같이 흩어진 관심사를 **Aspect로 모듈화하고 핵심적인 비즈니스 로직에서 분리하여 재사용**하겠다는 것이 AOP의 취지다. 프록시 패턴 기반의 AOP 구현체, 프록시 객체를 쓰는 이유는 **접근 제어 및 부가기능을 추가**하기 위해서이다. **스프링 빈에만 AOP를 적용 가능**하다.

스프링 IoC와 연동하여 엔터프라이즈 애플리케이션에서 **가장 흔한 문제**(중복코드, 프록시 클래스 작성의 번거로움, 객체들 간 관계 복잡도 증가 ...)**에 대한 해결책을 지원**하는 것이 목적이다.

| **개념** | **의미** |
| --- | --- |
| @Aspect | AOP 프레임워크에 포함되며, **AOP를 정의하는 Class**에 할당. 모듈화 한 것. 주로 부가기능을 모듈화함. |
| Target | Aspect를 적용하는 곳 (클래스, 메서드 .. ) |
| Advice | 실질적으로 어떤 일을 해야할 지에 대한 것, 실질적 기능을 담은 구현체.  독립된 클래스의 메소드로 작성한다. (Before, After, around 등등) |
| JoinPoint | 적용될 위치, 끼어들 수 있는 지점. 메서드 진입 지점, 생성자 호출 시점, 필드에서 값을 꺼내올 때 등 다양한 시점에 적용가능 |
| @Pointcut | 기능을 어디에 적용시킬지 선정한다. 즉, Advice를 적용할 JoinPoint를 정의한 것이다. **AOP를 적용시킬 지점**을 설정  @Pointcut("execution(\* 패키지명.패키지..(하위모든패키지)\*(모든클래스).\*(모든메소드)(..))") |

아래는 사용 예문이다.

| @Slf4j  **@Aspect**  **@Component**  public class ExecutionTimer {  // 조인포인트를 어노테이션으로 설정  **@PointCut**(" @annotation(com.aop.annotation.ExeTimer) ")  private void timer(){}; // 메소드 명을 써야하기 때문에 여기에 넣었다  // 메소드명으로 처리하기 위해 만듬. Around에 넣어진거 봐~ 아래쪽!  // 메서드 실행 전,후로 시간을 공유해야 하기 때문  **@Around**(" timer() ")  public void AssumeExecutionTime(**ProceedingJoinPoint** joinPoint) throws Throwable {  **StopWatch** stopWatch = new **StopWatch**();  stopWatch .start();  // ↑전처리  **Object result** = joinPoint.proceed(); // 조인포인트 \*(..) 메서드 실행(후촐) 시점.  // ↓후처리  stopWatch .stop();  log.info("class: {}, method: {} >>> {}",  joinPoint.getSignature().getDeclaringTypeName(), // 클래스명  joinPoint.getSignature().getName(), // 메소드명  stopWatch.getTotalTimeSeconds()); //토탈시간 계산. controller 안의 모든 메소드  **return**  **result** ;  }  } |
| --- |

### **- [ @Aspect ]**

@Aspect 어노테이션은 해당 Class가 **횡단관심사(부가기능) Class임을 알려주는** Annotation 이다. @Aspect Annotation이 부여되었다고 해서 자동으로 Bean으로 등록되는것이 아니므로 **따로 Bean으로 등록을 해주는 작업이 필요**하다.

@Component등의 어노테이션이 사용된다.

클래스에 @Order를 명시하여 **@Aspect로 등록된 빈 간의 작동 순서**를 정할 수 있다.

* 순서는 int 타입의 정수로 순서를 정할 수 있는데 값이 낮을수록 우선순위가 높음.
* 순서의 기본값은 가장 낮은 우선순위를 가지는 Ordered.LOWEST\_PRECEDENCE

@Aspect가 등록된 빈에는 Advice 메소드를 작성할 수 있다. 대상 스프링 빈의 method의 시점과 방법에 따라 **@Before, @After, @AfterReturning, @AfterThrowing, @Around** 등을 명시할 수 있다. 아래 참고.

### **- [ Advice(실행시점) ]**

타겟 메서드의 Aspect 실행 시점을 지정할 수 있는 어노테이션이다. **오류가 없을 때**의 호출 순서는 **[ Before > Around > After > AfterReturning ]** 이다. **오류가 있을 때**의 호출 순서는 **[ Before > After > AfterThrowing ]** 이다. 이때 Around는 실행되지 않는다.

advice는 끼어들기만 할 뿐 대상 메소드의 제어나 가공은 불가능하다.

| **Annotation** | **의미** |
| --- | --- |
| @Before | 메소드 실행하기 이전에 반드시 실행한다 |
| @After | 메소드가 성공적으로 실행후 예외가 발생 되더라도 실행  // 예외처리 할 수 없다 |
| @AfterReturing | 메소드가 에러(예외)없이 성공적으로 실행이 되면 advice 실행 |
| @AfterThrowing | 메소드가 에러(예외)가 발생하였을때 Exception을 던질때 advice 실행 |
| @Around | Before / after 모두 제어 // 필터처럼 전후 나눌수있음 |

| **@Aspect**  **@Component**  public class ExecutionTimer {  **@Before**(" **@annotation**(com.aop.annotation.ExeTimer) ")  public void AssumeExecutionTime(**ProceedingJoinPoint** joinPoint) throws Throwable {  }  }  **@Aspect**  **@Component**  public class ExecutionTimer {  // 조인포인트를 어노테이션으로 설정  **@PointCut**(" **@annotation**(com.aop.annotation.ExeTimer) ")  private void timer(){}; // 메소드 명을 써야하기 때문에 여기에 넣었다  **@Before**(" timer() ")  public void AssumeExecutionTime(**ProceedingJoinPoint** joinPoint) throws Throwable {  }  } |
| --- |

### **- [ joinPoint ]**

적용될 위치, 끼어들 수 있는 지점. 메서드 진입 지점, 생성자 호출 시점, 필드에서 값을 꺼내올 때 등 다양한 시점에 적용가능하다.

어드바이스 메소드를 의미있게 구현하려면 클라이언트가 호출한 **비즈니스 메소드의 정보**가 필요하다. 예를들면 예외가 터졌는데, 예외발생한 메소드의 이름이 뭔지 등을 기록할 필요가 있을 수 있다. 이럴때 JoinPoint 인터페이스가 제공하는 유용한 API들이 있다.





| **메소드** | **의미** |
| --- | --- |
| Signature **getSignature()** | 클라이언트가 호출한 메소드의 시그니처(리턴타입, 이름, 매개변수) 정보가 저장된 Signature 객체 리턴 |
| Object **getTarget()** | 클라이언트가 호출한 비즈니스 메소드를 포함하는 비즈니스 객체 리턴 |
| Object[] **getArgs()** | 클라이언트가 메소드를 호출할 때 넘겨준 인자 목록을 Object 배열 로 리턴. 매개변수 하나하나 |

| **Signature API** | **의미** |
| --- | --- |
| String **getName()** | 클라이언트가 호출한 메소드 이름 리턴 |
| String **getClass()** | 현재 참조하고 있는 클래스를 확인할 수 있다 |
| String **getDeclaringTypeName()** | 클라이언트가 호출한 클래스명 리턴 |
| String **toLongString()** | 클라이언트가 호출한 메소드의 리턴타입, 이름, 매개변수(시그니처)를 패키지 경로까지 포함 하여 리턴 |
| String **toShortString()** | 클라이언트가 호출한 메소드 시그니처를 축약한 문자열로 리턴 |

JoinPoint를 **어드바이스 메소드 매개변수로 선언**해야한다. 이때 인자는 스프링 컨테이너가 넘겨준다. ex. 메소드명(JoinPoint jp)

이때 **Around 어드바이스**만 다른 어드바이스와 약간 다른데, **ProceedingJoinPoint 객체를 인자로 선언해야한다.** ProceedingJoinPoint 는 JoinPoint를 상속받으며 **proceed() 등이 추가로 구현되어있다**

### **- [ Pointcut ]**

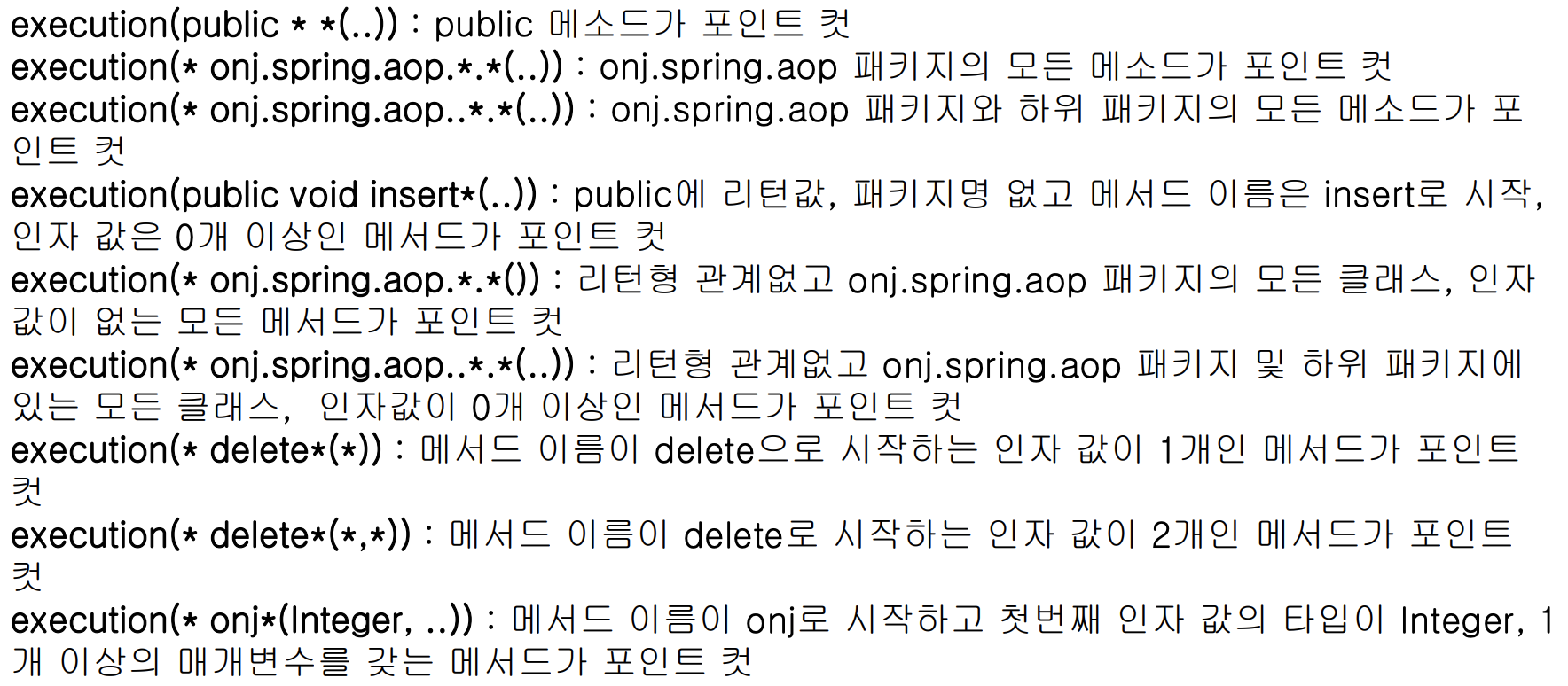
advice 메소드가 적용될 메서드 범위를 정확하게 적용해 줄 수 있다. 주로 @Pointcut 와 사용되며 advice 메소드(@Before, @Around~)에 바로 사용되기도 한다.

#### **execution**

| execution(접근제어자,반환형 패키지를 포함한 클래스 경로 메소드파라미터 )  ex ) **@Pointcut**( "**execution**(\* com.java..\*.\*())" ) |
| --- |

첫번째 \*기호는 역시 접근제어자, 반환형을 상관하지 않는다는 의미이며, com.java..부분의 경우 해당 패키지를 포함한 모든 하위패키지에 적용을 한다는 의미이다. 해석한다면 접근제어자, 반환형을 상관하지 않고 com.java패키지를 포함한 모든 하위디렉토리의 모든 Class의 모든 파라미터가 존재하지 않는 메소드에 적용한다는 의미이다.

| **용어** | **의미** |
| --- | --- |
| 수식어 | public, private 등 수식어를 명시합니다. (생략 가능) |
| 리턴타입 | 리턴 타입을 명시합니다. |
| 클래스이름 및 이름 | 클래스이름과 메서드 이름을 명시합니다.  (클래스 이름은 풀 패키지명으로 명시해야합니다. 생략도 가능) |
| 파라미터 | 메서드의 파라미터를 명시합니다. |
| " \* " | 모든 값을 표현합니다. |
| " .. " | 0개 이상을 의미합니다.  // 메소드 뒤의 .. 은 하위 메소드 전부를 이야기함 |



| **@Pointcut( "execution (public void get\*())" )** |
| --- |
| 퍼블릭형의 반환형이 없는(public void) 형태의 메소드 중 get으로 시작하는(get\*) 모든 메소드중 파라미터가 존재하지 않는(()) 메소드들에게 적용 |

| **@Pointcut( "execution (\* \*(..))" )** |
| --- |
| 첫번째 \*기호는 접근제어자와 반환형 모두를 상관하지 않고 적용하겠다는 의미이다. 두번째 \*기호는 어떠한 경로에 존재하는 클래스도 상관하지 않고 적용하겠다는 의미이다. 마지막으로 (..)은 메소드의 파라미터가 몇개가 존재하던지 상관없이 실행하는 경우이다. |

| **@Pointcut( "execution (\* com.java.ex.Car.accelerate())" )** |
| --- |
| 첫번째 \*기호는 역시 접근제어자, 반환형을 상관하지 않는다는 의미이며, com.java.ex.Car.accelerate()는 해당 Class의 accelerate()(파라미터가 없는) 메소드가 호출될 때 실행하는 경우를 의미한다. |

@Aspect

public class LogAop {

@Around("execution(public void com.java.ex.Car.\*())")

public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

String methodName = joinPoint.getSignature().toShortString();

try {

System.out.println(methodName + "is Start.");

Object obj = joinPoint.proceed();

return obj;

}finally {

System.out.println(methodName + "is Finish.");

}

}

}

#### **within**

Class의 경로. 패키지내의 모든 메소드에 적용할 때 사용한다.

| **@Pointcut( "within (com.java.ex.\*)" )** |
| --- |
| com.java.ex.하위의 모든 클래스의 모든 메소드에 적용하겠다는 의미이다. |

| **@Pointcut( "within (com.java.ex..\*)" )** |
| --- |
| com.java.ex.패키지의 하위 패키지를 포함한 모든 클래스의 모든 메소드에 적용하겠다는 의미이다. |

| **@Pointcut( "within (com.java.ex.Car\*)" )** |
| --- |
| com.java.ex.Car 클래스안의 모든 메소드에 적용하겠다는 의미이다. |

@Aspect

public class LogAop {

@Around("within(com.java.ex.\*)")

public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

String methodName = joinPoint.getSignature().toShortString();

try {

System.out.println(methodName + "is Start.");

Object obj = joinPoint.proceed();

return obj;

}finally {

System.out.println(methodName + "is Finish.");

}

}

}

#### **bean**

해당 bean id를 가지고있는 bean의 모든 메소드에 적용한다는 의미이다.

| Bean(bean id) |
| --- |

| **@Pointcut(bean(car))** |
| --- |
| car라는 bean id를 가지고있는 bean에게 적용한다 |

| **@Pointcut(bean(car\*))** |
| --- |
| 빈의 이름이 car으로 시작하는 빈의 모든 메서드에게 적용한다 |

@Around("bean(car)")

public Object logging(ProceedingJoinPoint joinPoint) throws Throwable {

String methodName = joinPoint.getSignature().toShortString();

try {

System.out.println(methodName + "is Start.");

Object obj = joinPoint.proceed();

return obj;

}finally {

System.out.println(methodName + "is Finish.");

}

}

#### **커스텀 Annotation**

사용자가 직접 만든 Annotation(인터페이스)에 기반해 Aspect를 적용할 수 있다.

| // 이 어노테이션을 부여하면 해당 메소드의 성능을 로깅합니다.  **@Retention**( RetentionPolicy.RUNTIME )  // 런타임 동안에 유지되고, 런타임 동안에 프로그램에서 접근도 가능  **@Target**( {ElementType.TYPE, ElementType.METHOD} )  // 타입(클래스) 위에 사용할 수 있다. 메소드 위에 사용할 수 있다. 선언.  public **@interface** **PerfLogging** {  } |
| --- |

**@Retention**으로 어느 시점까지 어노테이션의 메모리를 가져갈 지 설정하고, **@Target**으로 필드, 메소드, 클래스, 파라미터 등 선언할 수 있는 타입을 설정하면 대부분 커스텀 어노테이션을 쉽게 생성할 수 있다

| **@Aspect**  public class **LogAop** {  **@Around("@annotation(PerfLogging)")** // ("어노테이션 class명")  public **Object logging**(ProceedingJoinPoint joinPoint) throws Throwable {  String methodName = **joinPoint**.getSignature().toShortString();  try {  System.out.println(methodName + "is Start.");  **Object** obj = **joinPoint**.proceed();  return obj;  }finally {  System.out.println(methodName + "is Finish.");  }  }  } |
| --- |

### **- [ 에러처리 ]**

아래와 같은 오류가 발생할 시에는 PointCut 표현식의 오류일 확률이 높습니다. PointCut 표현식을 다시 한번 확인해야 한다.





만약 위와 같은 에러도 나타나지 않고 그냥 메소드만 출력되는 경우에는 IOC Container를 통해 해당 Bean이 생성되었는지를 생각해야 한다.

Spring AOP의 경우 AOP를 제공받을 Bean를 IOC Container에 의해 생성될 때 ProxyBean을 만들어서 AOP를 제공하기 때문에 직접 생성하는 것이 아니라 꼭 IOC Container에 의해 생성 돼야 한다.

에러가 났을 때 가져올 객체이름이 ex 라면 throwing="ex"라고 써주어야 한다.

@AfterThrowing(pointcut="execution(\* com.sist.dao.BoardImpl.get\*())||execution(\* com.sist.dao.BoardImpl.check\*(..))",throwing="ex")

public void afterThrowing(JoinPoint p,Throwable ex){

System.out.println("==AfterThrowing 수행==");

//어떤 메소드에서 실행되었느냐

System.out.println("Method Name:"+p.getSignature().getName());

//무슨 에러냐

System.out.println("Exception:"+ex.getMessage());

}

https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=bb\_&logNo=220384767015

## **REST API**

REST는 Representational State Transfer라는 용어의 약자로서 웹(HTTP)의 장점을 최대한 활용할 수 있는 아키텍처이다. 쉽게 말해 REST API는 다음의 구성으로 이루어져있다.

| **자원(RESOURCE) : URI** + **행위(Verb) : HTTP METHOD** + **표현(Representations)** |
| --- |

**특징1 Uniform (유니폼 인터페이스)**

Uniform Interface는 URI로 지정한 리소스에 대한 조작을 통일되고 **한정적인 인터페이스로 수행**하는 아키텍처 스타일을 말합니다.

**특징2 Stateless (무상태성)**

REST는 무상태성 성격을 갖습니다. 다시 말해 작업을 위한 **상태정보를 따로 저장하고 관리하지 않습니다**. 세션 정보나 쿠키정보를 별도로 저장하고 관리하지 않기 때문에 API 서버는 **들어오는 요청만을 단순히 처리**하면 됩니다. 때문에 서비스의 자유도가 높아지고 서버에서 불필요한 정보를 관리하지 않음으로써 구현이 단순해집니다.

**특징3 Cacheable (캐시 가능)**

REST의 가장 큰 특징 중 하나는 HTTP라는 기존 웹표준을 그대로 사용하기 때문에, **웹에서 사용하는 기존 인프라를 그대로 활용**이 가능합니다. 따라서 HTTP가 가진 캐싱 기능이 적용 가능합니다. HTTP 프로토콜 표준에서 사용하는 Last-Modified태그나 E-Tag를 이용하면 **캐싱 구현이 가능**합니다.

**특징4 Self-descriptiveness (자체 표현 구조)**

REST의 또 다른 큰 특징 중 하나는 **REST API 메시지만 보고도 이를 쉽게 이해** 할 수 있는 자체 표현 구조로 되어 있다는 것입니다.

**특징5 Client - Server 구조**

REST 서버는 API 제공, 클라이언트는 사용자 인증이나 컨텍스트(세션, 로그인 정보)등을 직접 관리하는 구조로 **각각의 역할이 확실히 구분**되기 때문에 클라이언트와 서버에서 개발해야 할 내용이 명확해지고 서로간 의존성이 줄어들게 됩니다.

**특징6 계층형 구조**

REST 서버는 다중 계층으로 구성될 수 있으며 보안, 로드 밸런싱, 암호화 계층을 추가해 구조상의 유연성을 둘 수 있고 PROXY, 게이트웨이 같은 네트워크 기반의 중간매체를 사용할 수 있게 합니다.

| 첫 번째, **URI**는 **정보의 자원을 표현**해야 한다.  두 번째, 자원에 대한 행위는 **HTTP Method(GET, POST, PUT, DELETE)로 표현**한다. |
| --- |

### **- [ CRUD Method ]**

HTTP METHOD의 알맞은 역할

POST, GET, PUT, DELETE 이 4가지의 Method를 가지고 CRUD를 할 수 있습니다.

| **Method** | **역할** | |
| --- | --- | --- |
| POST | POST를 통해 해당 URI를 요청하면 리소스를 **생성**합니다. | |
| GET | GET를 통해 해당 리소스를 **조회**합니다.  리소스를 조회하고 해당 도큐먼트에 대한 자세한 정보를 가져온다. | |
| PUT | PUT를 통해 해당 리소스를 **수정**합니다. | |
| DELETE | DELETE를 통해 리소스를 **삭제**합니다. | |

#### **URI 설계**

**1) 슬래시 구분자(/)는 계층 관계를 나타내는 데 사용**

* http://restapi.example.com/houses/apartments
* http://restapi.example.com/animals/mammals/whales

**2) URI 마지막 문자로 슬래시(/)를 포함하지 않는다.**

* http://restapi.example.com/houses/apartments/ ( X )
* http://restapi.example.com/houses/apartments ( 0 )

**3) 하이픈(-)은 URI 가독성을 높이는데 사용**

* URI를 쉽게 읽고 해석하기 위해, 불가피하게 긴 URI경로를 사용하게 된다면 하이픈을 사용해 **가독성**을 높일 수 있습니다.

**4) 밑줄(\_)은 URI에 사용하지 않는다.**

* 글꼴에 따라 다르긴 하지만 밑줄은 보기 어렵거나 밑줄 때문에 문자가 가려지기도 한다. 이런 문제를 피하기 위해 **밑줄 대신 하이픈(-)을 사용**하는 것이 좋다.(가독성)

**5) URI 경로에는 소문자가 적합하다.**

* URI 경로에 대문자 사용은 피하도록 해야 합니다. 대소문자에 따라 다른 리소스로 인식하게 되기 때문입니다. RFC 3986(URI 문법 형식)은 URI 스키마와 호스트를 제외하고는 대소문자를 구별하도록 규정하기 때문이다.
* RFC 3986 is the URI (Unified Resource Identifier) Syntax document

**6) 파일 확장자는 URI에 포함시키지 않는다.**

* http://restapi.example.com/members/soccer/345/photo.jpg (X)

**7) REST API에서는 파일 확장자를 URI 안에 포함시키지 않는다.**

* Accept header를 사용하도록 합시다.
* GET/members/soccer/345/photo HTTP/1.1 Host: restapi.example.com Accept: image/jpg

#### **리소스 표현**

REST 리소스 간에는 연관 관계가 있을 경우 다음과 같은 표현방법으로 사용한다. 만약 관계명이 복잡하다면 이를 서브 리소스에 명시적으로 표현할 수도 있다.

| /리소스명/리소스 ID/관계가 있는 다른 리소스명  ex1) GET : /users/{userid}/devices (일반적으로 소유 ‘has’의 관계를 표현할 때)  ex2) GET : /users/{userid}/likes/devices (관계명이 애매하거나 구체적 표현이 필요할 때) |
| --- |

Collection과 Document에 대해 알면 URI 설계가 더 쉽다. DOCUMENT는 한 객체, 컬렉션은 객체들의 집합이다. 컬렉션과 도큐먼트는 모두 리소스로 URI에 표현된다.

| sports라는 컬렉션과 soccer라는 도큐먼트  ex1) http:// restapi.example.com/sports/soccer  sports, players 컬렉션과 soccer, 13(13번인 선수)를 의미하는 도큐먼트로 이루어진 URI  ex2) http:// restapi.example.com/sports/soccer/players/13 |
| --- |

직관적인 REST API를 위해서는 컬렉션과 도큐먼트를 사용할 때 단수 복수도 구분한다.

### **- [ HTTP 응답 상태 코드 ]**

정확한 응답의 상태코드만으로도 많은 정보를 전달할 수가 있기 때문에 응답의 상태코드 값을 명확히 돌려주는 것은 생각보다 중요하다.

| **코드** |  |
| --- | --- |
| 200 | 클라이언트의 요청을 정상적으로 수행함 |
| 201 | 클라이언트가 어떠한 리소스 생성을 요청, 해당 리소스가 성공적으로 생성됨(POST를 통한 리소스 생성 작업 시) |

| **코드** |  |
| --- | --- |
| 400 | 클라이언트의 요청이 부적절 할 경우 사용하는 응답 코드 |
| 401 | 클라이언트가 인증되지 않은 상태에서 보호된 리소스를 요청했을 때 사용하는 응답 코드 |
| (로그인 하지 않은 유저가 로그인 했을 때, 요청 가능한 리소스를 요청했을 때) |
| 403 | 유저 인증상태와 관계 없이 응답하고 싶지 않은 리소스를 클라이언트가 요청했을 때 사용하는 응답 코드 |
| (403 보다는 400이나 404를 사용할 것을 권고. 403 자체가 리소스가 존재한다는 뜻이기 때문에) |
| 405 | 클라이언트가 요청한 리소스에서는 사용 불가능한 Method를 이용했을 경우 사용하는 응답 코드 |

| **코드** |  |
| --- | --- |
| 301 | 클라이언트가 요청한 리소스에 대한 URI가 변경되었을 때 사용하는 응답 코드 |
| (응답 시 Location header에 변경된 URI를 적어 줘야 합니다.) |
| 500 | 서버에 문제가 있을 경우 사용하는 응답 코드 |

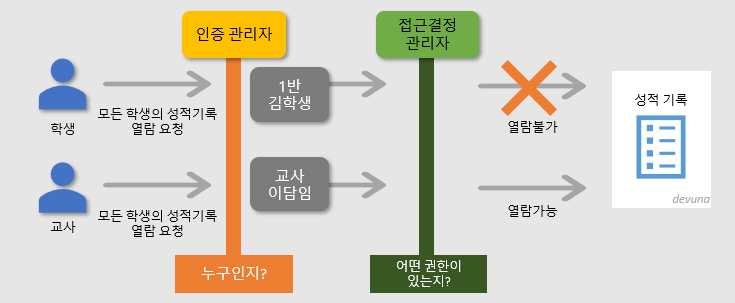
## **시큐리티**

스프링 시큐리티는 스프링 기반의 애플리케이션의 보안(인증과 권한, 인가 등)을 담당하는 스프링 하위 프레임워크이다. 즉 인증(Authenticate, 누구인지?) 과 인가(Authorize, 어떤것을 할 수 있는지?)를 담당하는 프레임워크를 말한다.

스프링 시큐리티에서는 주로 서블릿 필터(filter)와 이들로 구성된 필터체인으로의 구성된 위임모델을 사용한다. 그리고 보안과 관련해서 체계적으로 많은 옵션을 제공해주기 때문에 개발자 입장에서는 일일이 보안관련 로직을 작성하지 않아도 된다는 장점이 있다.

| **용어** | **역할** | |
| --- | --- | --- |
| 접근 주체(Principal) | 보호된 리소스에 접근하는 대상 | |
| 인증(Authentication) | 보호된 리소스에 접근한 대상에 대해 누구인지, 애플리케이션의 작업을 수행해도 되는 주체인지 확인하는 과정(ex. Form 기반 로그인)  => 즉, **누구인지?** | |
| 인가(Authorize) | 해당 리소스에 대해 접근 가능한 권한을 가지고 있는지 확인하는 과정(After Authentication, 인증 이후) => 즉, **어떤 것을 할 수 있는지?** | |
| 권한 | 어떠한 리소스에 대한 접근 제한, 모든 리소스는 접근 제어 권한이 걸려있음. 인가 과정에서 해당 리소스에 대한 제한된 최소한의 권한을 가졌는지 확인 | |

보안과 관련하여 체계적으로 많은 옵션을 제공하여 편리하게 사용할 수 있다. **Filter 기반**으로 동작하여 **MVC와 분리하여 관리 및 동작**한다. **어노테이션을 통한 간단한 설정**이 가능하며, Spring Security는 기본적으로 **세션 & 쿠키방식으로 인증**한다.



인증관리자(Authentication Manager)와 접근 결정 관리자(Access Decision Manager)를 통해 사용자의 리소스 접근을 관리한다. 인증 관리자는 UsenamePasswordAuthenticationFilter, 접근 결정 관리자는 FilterSecurityInterceptor가 수행한다.

### **- [ 필터별 기능 ]**

Role은 역할이고 Authority는 권한이지만 사실은 표현의 차이이다.

Role은 “ADMIN”으로 표현하고 Authority는 “ROLE\_ADMIN”으로 표기한다.

| **Method** | **역할** |
| --- | --- |
| SecurityContextPersistenceFilter | SecurityContextRepository에서 SecurityContext를 로드하고 저장하는 일을 담당함 |
| LogoutFilter | 로그아웃 URL로 지정된 가상URL에 대한 요청을 감시하고 매칭되는 요청이 있으면 사용자를 로그아웃시킴 |
| UsernamePasswordAuthenticationFilter | 사용자명과 비밀번호로 이뤄진 폼기반 인증에 사용하는 가상 URL요청을 감시하고 요청이 있으면 사용자의 인증을 진행함 |
| DefaultLoginPageGeneratingFilter | 폼기반 또는 OpenID 기반 인증에 사용하는 가상URL에 대한 요청을 감시하고 로그인 폼 기능을 수행하는데 필요한 HTML을 생성함 |
| BasicAuthenticationFilter | HTTP 기본 인증 헤더를 감시하고 이를 처리함 |
| RequestCacheAwareFilter | 로그인 성공 이후 인증 요청에 의해 가로채어진 사용자의 원래 요청을 재구성하는데 사용됨. SecurityContextHolderAwareRequestFilter HttpServletRequest를 HttpServletRequestWrapper를 상속하는 하위 클래스(SecurityContextHolderAwareRequestWrapper)로 감싸서 필터 체인상 하단에 위치한 요청 프로세서에 추가 컨텍스트를 제공함 |
| AnonymousAuthenticationFilter | 이 필터가 호출되는 시점까지 사용자가 아직 인증을 받지 못했다면 요청 관련 인증 토큰에서 사용자가 익명 사용자로 나타나게 됨 |
| SessionManagementFilter | 인증된 주체를 바탕으로 세션 트래킹을 처리해 단일 주체와 관련한 모든 세션들이 트래킹되도록 도움 |
| ExceptionTranslationFilter | 이 필터는 보호된 요청을 처리하는 동안 발생할 수 있는 기대한 예외의 기본 라우팅과 위임을 처리함 |
| FilterSecurityInterceptor | 이 필터는 권한부여와 관련한 결정을 AccessDecisionManager에게 위임해 권한부여 결정 및 접근 제어 결정을 쉽게 만들어 줌 |

### **- [ 예문 ]**

**WebSecurityConfigurerAdapter**를 상속받은 config 클래스에 **@EnableWebSecurity** 어노테이션을 달면SpringSecurityFilterChain이 자동으로 포함된다.

| **@EnableWebSecurity** // 스프링시큐리티 사용을 위한 어노테이션선언  public class **SecurityConfig** extends **WebSecurityConfigurerAdapter** {  } |
| --- |

| **application.ym**l에서 시큐리티 암호 설정 |
| --- |

작성한 뒤, configure 를 오버라이딩 하여 접근 권한을 작성한다.

| **@EnableWebSecurity** // 스프링시큐리티 사용을 위한 어노테이션선언  public class **SecurityConfig** extends **WebSecurityConfigurerAdapter** {  @Bean // @Configuration와 세트로 사용. IoC빈등록방법  public **BCryptPasswordEncoder** passwordEncoder(){  **return** new **BCryptPasswordEncoder**(); // 비밀번호 암호화.  // 아래처럼 DTO에서 하기도 한다.    }  @Override // 스프링 시큐리티 룰을 무시하게 하는 Url 규칙(여기 등록하면 규칙 적용하지 않음)  public void **configure**(WebSecurity web) throws **Exception** {  web.**ignoring**() // .ignoring : 필터 적용을 예외로 둔다  **.antMatchers**(" /resources/\*\* ") // .antMatchers : 해당 요청 주소들은  .**antMatchers**(" /css/\*\* ")  .**antMatchers**(" /vendor/\*\* ")  .**antMatchers**(" /js/\*\* ")  .**antMatchers**(" /favicon\*/\*\* ")  .**antMatchers**(" /img/\*\* ") ;  }  @Override // 스프링 시큐리티가 사용자를 인증하는 방법이 담긴 객체.  protected void **configure**(HttpSecurity http) throws **Exception** {   | 기본세팅:  super.configure(http);  // security password: 71fa6981-7015-4029-a171-e6cbfc40aef4 (basic 암호 설정)  // logout도 자동으로 만들어짐. 관련 페이지 전부 막힌다. (localhost:8000 전부 막힘) | | --- |   http.csrf().disable();  http.httpBasic().disable(); // httpBasic에서 제공해주는 로그인 사용X 선언. 비활성화.  http.authorizeRequests() // 모든 요청시에 실행을 해라  /\*<<<<<<<<< Page >>>>>>>>>\*/  .**antMatchers**("/admin/\*\*")  .hasRole(("ADMIN")) // ADMIN 권한이 필요하다. 하나의 권한  .access("hasRole('ADMIN') or hasRole('MANAGER')")  **.antMatchers("/account")** // [메인객체] 해당 요청 주소들은  .access("hasRole('USER') or hasRole('ADMIN') or hasRole('MANAGER')")  // USER ADMIN MANAGER 라는 롤을 가진 사용자만 접근 허용한다.  .authenticated() // 단, 인증이 필요하다.  .**antMatchers**("/", "/index", "/collections/")  .permitAll() // 전체 접근 허용  .**antMatchers**("/account/login", "/account/register")  .permitAll()  /\*<<<<<<<<< Resource >>>>>>>>>\*/  .**antMatchers**("/static/\*\*", "/image/\*\*")  .permitAll() // 전체 접근 허용  /\*<<<<<<<<< API >>>>>>>>>\*/  .**antMatchers**("/api/account/register")  .permitAll() // 전체 접근 허용  .anyRequest() // antMatchers외의 다른 모든 요청들은  .permitAll() // 수업을 위해 잠시 허용 시켜줌  .denyAll() // 모두 접근 권한 차단하라  .and() // 그리고!!!  **.formLogin()** // [메인객체] 폼 로그인 방식으로 인증해라  .usernameParameter("email") //username일 경우 생략가능 (기본값이 username)  //PrincipalDetailService 에서 loadUserByUsername의 (파라미터)와 값이 동일해야함!  .loginPage("/account/login") // 우리가 만든 로그인 페이지를 사용해라 (GET 요청)  .loginProcessingUrl("/account/login")  // 로그인 로직(PrincipalDetailService)의 loadUserByUsername 메소드 호출 (post 요청)  // security 라이브러리에서 컨트롤러가 이미 만들어져있다. 거기에 PostMapping한다.  // html form 에서 /account/login 로 요청을 날림!  .failureHandler()  .failureHandler(new **AuthFailureHandler**()) // Form Login 실패시 수행되는 핸들러  // 실패시 로그인 페이지로 리다이렉트 하는 역할을 수행  .defaultSuccessUrl("/index");  // "이전 페이지가 없는 경우"에만 로그인 성공시 index로 보낸다.  // 보통 자동으로 이전 페이지를 기억했다가 로그인 페이지 전 페이지로 돌아간다.  }  } |
| --- | --- |

### **- [ antMatchers() ]**

Role은 역할이고 Authority는 권한이지만 사실은 표현의 차이이다.

Role은 “ADMIN”으로 표현하고 Authority는 “ROLE\_ADMIN”으로 표기한다.

| **Method** | **역할** | |
| --- | --- | --- |
| hasRole()  hasAnyRole() | 특정 권한을 갖는 사용자만 접근할 수 있다. | |
| hasAuthority()  hasAnyAuthority() | 특정 권한을 갖는 사용자만 접근할 수 있다 | |
| hasIpAddress() | 특정 아이피 주소를 갖는 사용자만 접근할 수 있다 | |
| permitAll() | 접근을 전부 허용한다 | |
| denyAll() | 접근을 전부 제한한다 | |
| rememberMe() | 리멤버 기능을 통해 로그인한 사용자만 접근할 수 있다 | |
| anonymous() | 인증되지 않은 사용자가 접근할 수 있다 | |
| authenticated() | 인증된 사용자만 접근할 수 있다 | |

### **- [ AuthenticationFailureHandler ]**

Form Login 실패시 수행되는 핸들러이다. SimpleUrlAuthenticationFailureHandler를 상속 받으며, 실패시 로그인 페이지로 리다이렉트 하는 역할을 수행합니다.

| public class **AuthFailureHandler** implements **AuthenticationFailureHandler** {  @Override  public void **onAuthenticationFailure**(**HttpServletRequest** request, **HttpServletResponse** response, **AuthenticationException** exception) **throws** IOException, ServletException {  // 로그인에서 아이디나 비밀번호가 잘못되었습니다.  if(exception.getClass() == UsernameNotFoundException.class // 계정이 없거나  || exception.getClass() == BadCredentialsException.class){ // 비밀번호가 틀릴 경우  response.sendRedirect(" **/account/login?error=auth** "); // 해당 링크로 이동한다  }else{  response.sendRedirect(" /account/login?error ");  }  }  } |
| --- |

## **properties / yml**

개발을 진행하다 보면 **데이터베이스 연결 정보**나 **외부 API 주소** 등과 같은 메타 정보들을 관리해야 한다. 해당 메타정보들을 클래스 파일에 넣을 수도 있겠지만 다음과 같은 이유로 프로퍼티 파일(properties)나 야믈 파일(yaml, yml)로 분리해서 관리할 것을 권장한다.

| - 환경에 따라 유연하게 값을 설정할 수 있음  - 초기값을 설정할 수 있음  - 불필요한 컴파일을 줄일 수 있음 |
| --- |

### **- [ @Value ]**

@Value로 메타 정보를 가져오는 방법으로는 **PropertyPlaceHolderConfigurer**를 통해 **${}**을 사용하는 방식과 **SpEL**을 통해 **#{}**을 이용하는 방식 크게 2가지가 있다.

#### **${ }**

PropertyPlaceHolderConfigurer은 빈 팩토리 후처리기로써 매번 빈 오브젝트가 만들어진 직후에 오브젝트의 내용이나 오브젝트 자체를 변경하는 빈 후처리기와 달리 빈 설정 메타정보가 모두 준비됐을 때 빈 메타정보 자체를 조작하기 위해 사용된다.

| database.username = MangKyu  database.className=org.mariadb.jdbc.Driver  **@Value**( "${database.username}" ) // @Value("파일 경로"). @autowired같은 역할을 한다  private String **userName**;  **@Value**( "${database.className}" )  private String **className**; |
| --- |

database.username과 같이 properties에 작성된 키 값을 ${}안에 넣어주면 Spring이 PropertyPlaceHolderConfigurer를 통해 초기화 작업 중에 해당 값을 실제 값으로 치환한다.

하지만 이 방법은 **대체할 위치를 치환자로 지정해두고, 별도의 후처리기가 값을 변경 해주기를 기대**하므로 수동적이다. 그래서 초기 빈 메타정보에는 ${database.username}와 같은 문자열이 등록되어 있다. 그러다가 스프링 컨테이너가 설정 파일에서 대체할 적절한 키 값을 찾지 못하면 해당 문자열이 그대로 남아있게 된다. 그래서 **치환자의 값이 변경되지 않더라도 예외가 발생하지 않으므로** SpEL을 통한 능동 변환을 권장한다.

초기에는 PropertyPlaceHolderConfigurer를 사용하는 방식을 사용했어야 했지만 Spring3부터 SpEL을 지원하면서 이를 이용하는 방식을 권장한다.

최근 Spring의 버전이 높아지며 문제를 개선한 PropertySourcesPlaceholderConfigurer가 등장하여 적절한 값을 찾지 못하면 에러가 발생하도록 수정되었지만 그래도 수동적인 방법보다는 빈에서 값을 꺼내는 SpEL이 권장된다.

| **application.yml** | **file:** #파일 저장 경로 #@Value("${file.path}")로 사용한다.  **path:** C:/Korea IT/web/springboot/springboot-mybatis-20220923-youri/upload/ |
| --- | --- |
| **MvcConfi.java** | **@Configuration** //설정 관련된 객체  public class **MvcConfig** implements **WebMvcConfigurer** {  **@Value("${file.path}")** // 이걸로 이미지 불러오는 걸 해 볼 거다  private String filePath;  **@Override**  public void **addResourceHandlers**(**ResourceHandlerRegistry** registry) {  WebMvcConfigurer.super.**addResourceHandlers**(registry); //default 세팅.  registry.**addResourceHandler**( " /image/\*\* " )  .**addResourceLocations**( " file:/// " + filePath );  } |

#### **#{ }**

SpEL을 통한 능동 변환

## **Controller**

## 사용자의 요청이 진입하는 지점(entry point)이며, 요청에 따라 어떤 처리를 할지 결정해준다. 단, controller는 단지 결정만 해주고 실질적인 처리는 서비스에서 담당한다.

사용자에게 View(또는 서버에서 처리된 데이터를 포함하는 View)를 응답으로 보내준다.

| package com.god.bo.test.controller;  import org.springframework.stereotype.Controller;  import org.springframework.web.bind.annotation.RequestMapping;  **@Controller** // 컨트롤러에 필수로 달림  public class **TestController** {  **@RequestMapping(value = "/", "/home")**  // 둘 중 하나로 선택되어 아래가 실행된다  public String **home**(){  **return** "index.html"; // 파일명이 index.html인 View 파일을 반환!  }  } |
| --- |

대규모 서비스로 갈수록 처리해야할 서비스들이 많아지는데, 이를 하나의 클래스에서 몰아 처리할 게 아니라 controller라는 **중간 제어자 역할**을 하는 것을 만들어서 A요청에 대한 것은 A-controller가 맡아 필요한 로직처리를 위한 서비스를 호출하게 된다.

controller는 MVC 패턴에 포함되는 것인데, Model View Controller의 역할에 따라 설계하고 코딩하면 주먹구구식으로 개발할 때보다 개발비용이나 유지보수비용이 대폭 줄어든다. **역할분담이 핵심**이다.

## 

### **- [ @Controller ]**

- Controller의 역할을 수행 한다고 명시하는 어노테이션.

즉, 해당 클래스를 Controller로 사용한다고 Spring FrameWork에 알린다.

- 필요한 로직을 호출하여 전달할 Model과 이동할 View 정보를 DispatherServlet에 반환.

- Bean으로 등록한다.

### **- [ @RequestMapping ]**

- 요청에 대해 어떤 Controller, 어떤 메소드가 처리할지 맵핑하기 위한 어노테이션

- **view를 반환**할 경우 사용된다. (**data를 반환**해야 하는 경우라면 **@ResponseBody**를 사용)

- 클래스나 메서드 선언부에 @RequestMapping과 함께 **URL을 명시**하여 사용한다.

- viewName **생략시** @RequestMapping의 **path로 설정한 URL**이 default viewName

#### **value : URL 값**

**속성1.** value(String[]) : URL 값

| EX1) @RequestMapping(**value** = **"**/login**"**)  EX2) @RequestMapping(**"**/login**"**) |
| --- |

#### **method : HTTP Request 메소드 값**

**속성2.** method (RequestMethod[]) : HTTP Request 메소드 값

- GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE

| EX1) @RequestMapping(**value** = "/login", **method** = @RequestMethod.GET)  == @GetMapping("/login") \* Spring4.3 이후부터 이렇게 쓸수 있다.  EX2) @RequestMapping(**value** = "/login", **method** = @RequestMethod.POST)  == @PostMapping("/login")  EX3) @RequestMapping("/login")  + @PostMapping("/v1") // 주소 칠 때 "/login/v1"  + @PostMapping("/v2") // "/login/v2" 로 입력해야함 |
| --- |

#### **params : HTTP Request 파라미터**

**속성3.** params(String[]) : HTTP Request 파라미터

- @RequestParam : 사용자가 원하는 **매개변수에 값을 매핑**하기위해 사용한다.

| EX)  **@PostMapping**("/member")  public String **member**(@RequestParam String name, @RequestParam Int age) |
| --- |

- @PathVariable : url 경로를 **변수화**하여 사용할 수 있도록 해준다.

| EX)  @RequestMapping("/member/{name}/{age}")  public String **member**(**@PathVariable("name")** String name, **@PathVariable("age")** String age) |
| --- |

#### **consumes : Request Body**

**속성4.** consumes(String[]) : Request Body에 담는 타입을 제한할 수 있다.

| EX) **@PostMapping**("/login", **consumes**="application/json") |
| --- |

헤더에 **application/json**이 존재 해야 처리한다.

### **- [ @ResponseBody ]**

@ResponseBody 로 객체 정보 전달 하기

- view가 아닌 **data를 반환**해야 하는 경우라면 **@ResponseBody를 사용**하면 된다.

- @ResponseBody 어노테이션을 통해 간단하게는 **String, Map, JSON 등을 전달**할 수 있다.

- JSON 형태로 전달한다.

https://tecoble.techcourse.co.kr/post/2021-05-10-response-entity/

#### **@RestController**

Spring 4.0부터 @Controller와 @ResponseBody 대신 @RestController을 제공한다.

@RestController를 사용하면

- @ResponseBody를 **추가 할 필요가 없고**,

- @ResponseBody 어노테이션은 **기본적으로 활성화**되어 있다.

| **@RestController**  public class **ResponseTestController** {  (생략) @ResponseBody  **@GetMapping**("/api/v1/data1")  **public** Map<String, Object> **getData1**(){  Map<String, Object> data = new HashMap<String, Object>();  data.put("이름", "홍길동");  return data;  }  (생략) @ResponseBody  **@GetMapping**("/api/v1/data2")  **public** Map<String, Object> **getData2**(){  return ResponseDataDto.builder()  .value1("그냥 문자열")  .value2(1000)  .value3(ture)  .build();  }  } |
| --- |

Map과 List 활용 예문

| *package* com.boot.mvc20220916youri.web.dto StudentRespDto.Java  **@Builder** // 생성하기 편하게 하기 위함  **@Getter** // jackson은 이게 없으면 에러남  public class StudentRespDto {  private int studentCode;  private String studentName;  private int studentYear;  }  *package* com.boot.mvc20220916youri.web.controller.api ResponseController.Java  **@RestController**  public class **ResponseTestController** {  **@GetMapping**("/api/v1/students")  public Object **getData4**(){  Map<String, Object> responseMap = new HashMap<String, Object>();  List<StudentRespDto> studentList = new ArrayList<StudentRespDto>();  **StudentRespDto** s1 = **StudentRespDto**.builder()  .studentCode(20220001)  .studentName("김준일")  .studentYear(1)  .build();  **StudentRespDto** s2 = **StudentRespDto** .builder()  .studentCode(20220002)  .studentName("김준이")  .studentYear(4)  .build();  studentList .add(s1);  studentList .add(s2);  responseMap .put("students1", studentList);  responseMap .put("students2", studentList);  return responseMap ;  }  } |
| --- |

### **- [ @ResponseEntity ]**

@ResponseBody 와 달리 Annotation 이 아닌 객체로 사용된다. 즉, 응답으로 변환될 정보를 모두 담은 요소들을 객체로 만들어서 반환해준다. 일반적인 API는 반환하는 리소스에 Value만 있지 않다.

**HttpStatus**, **HttpHeaders**, **HttpBody** 가 사용될 때 쓰이는 것이 ResponseEntity Class 이다. ResponseEntity는 HttpEntity를 상속받는다. <body, header, status> 가 성립한다.

Http Header에는 응답/요청에 대한 요구사항이, http body에는 그 내용이 적혀있다.

| public class **ResponseEntity<T>** extends **HttpEntity<T>**{  private final Object status;  } |
| --- |

이와 같이 ResponseEntity 는 **HttpEntity 를 상속**하여 구현된다.

#### **@GetMapping**

ResponseController

| **@AllArgsConstructor**  **@Getter**  public class **CMRespDto**<T> {  private int **code**; // 1 : 성공, -1 : 실패  private String **msg**; // commit 메시지  private T **data**; // 응답데이터  }  **@GetMapping**("/api/v1/entity/data")  public **ResponseEntity** <?> Data() {  **HttpHeaders** headers = new **HttpHeaders** ();  headers .add("Game", "Chess"); // headerName이 Game, headerValue가 Chess이다.    **return** **ResponseEntity**.**ok**()  .**headers**(headers)  .**body**(new **CMRespDto**<>( 1, "전송성공", "테스트데이터" ));  } |
| --- |

#### **@PostMapping**

RequestController

| **@AllArgsConstructor**  **@Getter**  public class **CMRespDto**<T> {  private int **code**; // 1 : 성공, -1 : 실패  private String **msg**; // commit 메시지  private T **data**; // 응답데이터  }  **@Data**  public class **AddPostReqDto** {  private String **title**;  private String **writer**;  private String **content**;  private MultipartFile **file**;  private List<MultipartFile> **files**;  }  **@PostMapping**("/api/v1/file/post") //포스트맨에서 form-data 사용  public **ResponseEntity** <?> addPost(**AddPostReqDto** addPostReqDto) {  List<String> fileNames = new ArrayList<String>();    String fileName1 = addPostReqDto.getFile().getOriginalFilename();  fileNames.add(fileName1);  if(addPostReqDto.getFiles() != null){ // List 인 files가 없을 때 사용  String fileName2 = addPostReqDto.getFiles().get(0).getOriginalFilename(); //리스트  String fileName3 = addPostReqDto.getFiles().get(1).getOriginalFilename(); //리스트  fileNames.add(fileName2);  fileNames.add(fileName3);  }    **return** **ResponseEntity**.**ok**(new **CMRespDto**<>(1, "게시글 작성 완료", fileNames))  // application.yml에 파일 크기 세팅해줘야 한다.    } |
| --- |

| **@PostMapping**("/api/v1/json/post") //포스트맨에서 json 사용  // Json은 늘 @RequestBody가 따라와야한다  public **ResponseEntity**<?> addPost(**@RequestBody** **AddPostReqDto** addPostReqDto) {  log.info("{}", addPostReqDto);  return ResponseEntity.ok(new CMRespDto<>(1, "Json으로 게시글 작성 완료", addPostReqDto));  } |
| --- |

#### **@DeleteMapping**

RequestController

| **@AllArgsConstructor**  **@Getter**  public class **CMRespDto**<T> {  private int **code**; // 1 : 성공, -1 : 실패  private String **msg**; // commit 메시지  private T **data**; // 응답데이터  }  **@DeleteMapping**("/api/v1/post/**{id}**") //포스트맨에서 DELETE > Params 사용  public **ResponseEntity** <?> deletePost(**@PathVariable** int id){  **return** **ResponseEntity**.**ok**(new **CMRespDto**<>(1, "삭제 성공", id))  } |
| --- |

## **데이터 전송**

### **- [ $.ajax ]**

비동기 통신. 클라이언트 - 서버 간에 XML 또는 JSON 데이터를 주고 받는 기술이다. 페이지 이동 없이 데이터 처리가 가능하다. 새로 고침 없이 화면이 변한다.

| < script type="text/javascript" src="<http://code.jquery.com/jquery-latest.min.js>" ></script> |
| --- |

| **실제표현** | **비고** |
| --- | --- |
| async | 비동기로 할 거냐 말 거냐 |
| type | 통신 타입을 설정 |
| url | 요청할 url |
| data | 서버에 요청할 때 보낼 매개변수 설정 |
| dataType | 응답 받을 데이터 타입(XML,JSON,TEXT,HTML 등) |
| success | 요청/응답에 성공했을 때 처리할 구문 |
| error | 요청/응답에 실패했을 때 처리할 구문 |
| complete | 작업을 마친 후에 처리할 구문 |

#### **"key" : "value"**

| function **request**(){  **$.ajax(**{  **async:** false, // async 동기(false), 비동기(true\_default) 선택  **type:** "post", // RequestMethod (get요청 post요청 선택)  **url:** "/api/v1/rp/post", // 요청 URL (보통 api)  **data:** { // 전송데이터  "title": "request param 제목",  "writer": "홍길동",  "content": "내용을 입력합니다"  },  **dataType:** "json", // 응답데이터 형식(Response ContentType)  **success:** (response) => { // response 응답 성공시 실행되는 function  console.log(response);  appendData(response);  },  **error:** (error) => { // 요청 또는 응답이 실패하였을 때 오류처리  console.log(error);  }  });  } |
| --- |
| function **appendData**(response){  const body = document.querySelector("body");  innerHTML = `  <h1> ${response.code} </h1>  <h1> ${response.mgs} </h1>  <h1> ${response.data.title} </h1>  <h1> ${response.data.writer} </h1>  <h1> ${response.data.content} </h1>  `;  } |

#### **FormData**

form 태그에 where, how를 기재하여 전송, Controller는 객체에 담아 받는다.

| const formData = new **FormData**( **[form]** );  // [form] = document.querySelector("form"); |
| --- |

FormData 객체를 바디로 받는다는 건 FormData의 특징이다. 이때 브라우저가 보내는 HTTP 메시지는 인코딩되고 Content-Type 속성은 multipart/form-data로 지정된 후 전송된다.

서버 관점에선 FormData를 사용한 방식과 일반 폼 전송 방식에 차이가 없다.

| **메서드** | **비고** |
| --- | --- |
| .append(name, value) | name과 value를 가진 폼 필드 추가. 이름(name)이 같은 필드 여러 개를 허용하기 때문에 append 메서드를 여러 번 호출해 이름이 같은 필드를 계속 추가해도 문제가 없다. |
| .append(name, blob, fileName) | <input type="file"> 형태의 필드를 추가. 세 번째 인수 fileName은 (필드 이름이 아니고) 사용자가 해당 이름을 가진 파일을 폼에 추가한 것처럼 설정해줌 |
| .delete(name) | name에 해당하는 필드를 삭제 |
| .get(name) | name에 해당하는 필드의 값을 가져옴 |
| .has(name) | name에 해당하는 필드가 있으면 true를, 않으면 false를 반환 |
| .set(name, value)  =.append(name, value)  .set(name, blob, fileName)  =.append(name, blob, fileName) | .append와 유사. name과 동일한 이름을 가진 필드를 모두 제거하고 새로운 필드 하나를 추가한다. 따라서 set 메서드를 쓰면 name을 가진 필드가 단 한 개만 있게끔 보장할 수 있다. |

**폼을 전송할 때** HTTP 메시지의 **Content-Type 속성은 항상 multipart/form-data**고 메시지는 인코딩되어 전송된다. 파일이 있는 폼도 당연히 이 규칙을 따르기 때문에 <input type="file">로 지정한 필드 역시 일반 폼을 전송할 때와 유사하게 전송됩니다.

| **실제표현** | **비고** |
| --- | --- |
| CRUD (Create, Read, Update, Delete) | Create |
| form data | <form> 태그에 담긴 데이터 (택배 상자에 비유) |
| DTO(Data Transfer Object) | form data를 받는 객체 |

| <**form** enctype=**"multipart/form-data"**>  <input type = "text" **name** = "formDataText">  <input type = "**file**" **name** = "**file**" class = "file-upload-input"  **accept**="image/gif, image/png, image/jpeg, image/jpg"  **multiple**> // multiple이 있으면 파일 여러개 첨부 가능  <**/form**>  <img class="file-preview"> |
| --- |
| const fileInput = document.querySelector(".file-upload-input");  fileInput.onchange = () => {  getPreview();  if( confirm("이미지 데이터 전송을 하시겠습니까?") ){  request();  }  } |
| function **getPreview**(){  const reader = new **FileReader()**; // 리더 객체 생성  reader **.onload** = (e) => { // readAsDataURL이게 호출될 때 실행된다.  //정확히는 읽기 동작이 성공적으로 완료되었을 때 발생  const filePreview = document.querySelector(".file-preview");  filePreview .src = e.target.**result**; // result값은 src 경로 (파일의 내용을 반환)  }  reader**.readAsDataURL**(fileInput .files[0]);  // 바이너리 Base64 Encode 문자열로 반환하는 메소드  } |
| function **getFormData**(){  const form = document.querySelector("form");  const formData = new FormData(form);  //form을 객체로 잡아줄때 name속성을 key로 잡아준다  //Dto랑 form데이터가 일치해야함  formData .append("title", "폼데이터 제목");  formData .append("writer", "김준일");  formData .append("content", "파일을 포함한 게시글");  for(let pair of formData.**entries**()) {  console.log(`key: ${pair[0]}, value: ${pair[1]}`);  }  return formData ;  }  // **entries()**: 배열에 사용하는 반복자로 실행하면 반복자 객체를 반환한다. 이 반복자 객체를 사용하여 내부의 값들을 돌면서 키(key)와 밸류(value)를 얻을 수 있다. |
| function **request**(){  **$.ajax(**{  **async:** false, // async 동기(false), 비동기(true\_default) 선택  **type:** "post", // RequestMethod (get요청 post요청 선택)  **url:** "/api/v1/rp/post", // 요청 URL (보통 api)  **enctype: "multipart/form-data",** // form데이터 할때 필수  **contentType: false,**  // form데이터 할때 필수  **processData: false,**  // form데이터 할때 필수  **data:** getFormData(), // 전송데이터  **dataType:** "json", // 응답데이터 형식(Response ContentType)  **success:** (response) => { // response 응답 성공시 실행되는 function  console.log(response);  appendData(response);  },  **error:** (error) => { // 요청 또는 응답이 실패하였을 때 오류처리  console.log(error);  }  });  } |
| function **appendData**(response){  const body = document.querySelector("body");  innerHTML = `  <h1> ${response.code} </h1>  <h1> ${response.mgs} </h1>  <h1> ${response.data.title} </h1>  <h1> ${response.data.writer} </h1>  <h1> ${response.data.content} </h1>  `;  } |

#### **json**

form 태그에 where, how를 기재하여 전송, Controller는 객체에 담아 받는다.

| const formData = new **FormData**( **[form]** );  // [form] = document.querySelector("form"); |
| --- |

FormData 객체를 바디로 받는다는 건 FormData의 특징이다. 이때 브라우저가 보내는 HTTP 메시지는 인코딩되고 Content-Type 속성은 multipart/form-data로 지정된 후 전송된다.

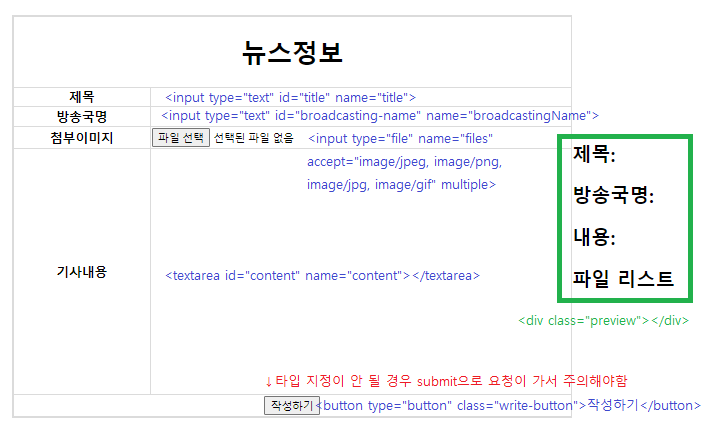
서버 관점에선 FormData를 사용한 방식과 일반 폼 전송 방식에 차이가 없다.

| **apiController** |
| --- |
| @PostMapping("/api/v1/json/post") //Json은 늘 @RequestBody가 따라와야한다  public **ResponseEntity**<?> addPost(**@RequestBody** **AddPostReqDto** addPostReqDto) {  log.info("{}", addPostReqDto);  return **ResponseEntity**.ok(new CMRespDto<>(1, "Json으로 게시글 작성 완료", addPostReqDto));  } |

| let **post** = {  title: "json 으로 전송하는 제목",  writer: "홍길동",  content: "내용을 전송합니다.."  }  window.**onload** = () => {  **load**();  }  function **load**(){  **request**();  } |
| --- |
| function **request**(){  **$.ajax(**{  **async:** false, // async 동기(false), 비동기(true\_default) 선택  **type:** "post", // RequestMethod (get요청 post요청 선택)  **url:** "/api/v1/json/post", // 요청 URL (보통 api)  **contentType: application/json,** // 데이터를 JSON 보낼때 필수  **data:** JSON.stringify(**post**), // 데이터를 JSON 보낼때 필수.  // 객체를 문자열 형태로 바꿔준다  **dataType:** "json", // 응답데이터 형식(Response ContentType)  **success:** (response) => { // response 응답 성공시 실행되는 function  console.log(response);  },  **error:** (error) => { // 요청 또는 응답이 실패하였을 때 오류처리  console.log(error);  }  });  } |

#### **활용예제**

| **@AllArgsConstructor**  **@Getter**  public class **CMRespDto**<T> {  private int **code**; // 1 : 성공, -1 : 실패  private String **msg**; // commit 메시지  private T **data**; // 응답데이터  }  **@Data**  public class **AddNewReqDto** {  private String **title**;  private String **broadcastingName**;  private List<MultipartFile> **files**;  private String **content**;  }  **@PostMapping**("/api/news")  public **ResponseEntity** <?> addPost(**AddPostReqDto** addPostReqDto) {  **Map**<String, Object> map = new **HashMap**<String, Object>();  map .put(" title ", addPostReqDto**.getTitle**());  map .put(" broadcastingName ", addPostReqDto**.getBroadcastingName**());  map .put(" content ", addPostReqDto**.getContent**());  **List**<String> fileNameList = new **ArrayList**<String>();  addPostReqDto.**getFiles**().forEach((file) -> {  **fileNameList** .add(file.**getOriginalFilename**()); //업로드되는 파일 이름 적어준다.  });  map .put(" fileNames ", **fileNameList** );  **return** **ResponseEntity**.**ok**(new **CMRespDto**<>(1, " 뉴스등록완료 ", map ))  // application.yml에 파일 크기 세팅해줘야 한다. // map 이 **request**()의 response.**data**가 된다.  } |
| --- |



| const **writeButton** = document.querySelector(".write-button");  writeButton.onclick = () => {  **request**();  }  **request**(); |
| --- |
| function **getFormData**(){  const form = document.querySelector("form");  **return** new **FormData**(form); // key랑 value 값을 form 데이터에서 얻어옴  } |
| function **request**(){  **$.ajax(**{  **async:** false, // async 동기(false), 비동기(true\_default) 선택  **type:** "post", // RequestMethod (get요청 post요청 선택)  **url:** "/api/news", // 요청 URL (보통 api)  **enctype: "multipart/form-data",** // form데이터 할때 필수  **contentType: false,**  // form데이터 할때 필수  **processData: false,**  // form데이터 할때 필수  **data:** **getFormData**(), // 전송데이터  **dataType:** "json", // 응답데이터 형식(Response ContentType)  **success:** (response) => { // response 응답 성공시 실행되는 function  console.log(response);  const preview = document.querySelector(".preview");  preview.innerHTML = `  <h2>제목: ${response.**data**.title}</h2> // input의 value값 추출  // key = name. title  <h2>방송국명: ${response.**data**.broadcastingName}</h2>  <h2>내용: ${response.**data**.content}</h2>  <h2>파일 리스트</h2>  `;  for(let i = 0; i < response.data.fileNames.length; i++){  preview .innerHTML += `  <h3>${response.**data**.fileNames[i]}</h3>  `;  }  },  **error:** (error) => { // 요청 또는 응답이 실패하였을 때 오류처리  console.log(error);  }  });  } |

## **파일 업로드**

### **- [ MultipartFile ]**

스프링 MVC에는 MultipartFile 타입을 제공해서 업로드 되는 파일 데이터를 쉽게 처리한다. 화면에서 첨부파일을 여러 개 선택할 수 있으므로 배열 타입으로 설정한 후 파일을 업로드 할 수 있다.

| **Method** | **역할** | |
| --- | --- | --- |
| String getName() | 파라미터의 이름 <input> 태그의 이름 | |
| String getOriginalFileName() | 업로드 되는 파일의 이름 | |
| boolean isEmpty() | 파일이 존재하지 않는 경우 true | |
| long getSize() | 업로드 되는 파일의 크기 | |
| byte[] getBytes() | byte[]로 파일 데이터 반환 | |
| InputStream getInputStream() | 파일데이터와 연결된 InputStream을 반환 | |
| transferTo(File file) | 파일의 저장 write로 사용할 수 있다. | |

| @Builder  @Data  @NoArgsConstructor  @AllArgsConstructor  public class **NewsFile** { // domain 파일  private int file\_id;  private int news\_id;  private String file\_origin\_name;  private String file\_temp\_name;  } |
| --- |
| **@Value**("${file.path}") //aplication.yml 경로가지고옴 @autowired같은 역할  private String **filePath**;  **@PostMapping("/news")**  public **ResponseEntity**<?> write(**NewsWriteReqDto** newsWriteReqDto) {  List<**NewsFile**> newsFileList = null;  **MultipartFile** firstFile = newsWriteReqDto.getFiles().get(0);  String firstFileName = firstFile .getOriginalFilename(); // 업로드 되는 파일의 이름  // 첨부 파일의 개수만큼 반환함.  if(!firstFileName .isBlank()) { // 첨부된 파일이 있을 경우 if 실행  // 파일 첨부를 안했을 때도 파일을 넣었다고 인지한다.  //그래서 파일이없는 파일이 1개 들어간걸로 인식.  //하기 때문에 isBlank로 한 번 걸러줘야함  // List<NewsFile> newsFileList = null; 위에 이걸로 걸러짐  newsFileList = new ArrayList<NewsFile>();  for(**MultipartFile** file : **newsWriteReqDto**.getFiles()){  String originFileName = file .getOriginalFilename();  //파일 이름들을 하나씩 출력한다.  String uuid = UUID.randomUUID().toString();  //파일명이 겹치면 안되므로 사용  String extension = originFileName.substring(originFileName.lastIndexOf("."));  //확장자명 가져오기  String tempFileName = uuid + extension ;  **Path** uploadPath = **Paths**.get(**filePath**, "news/" + tempFileName);  //log.info("파일 업로드 경로: {}", uploadPath.toString());  **File** f = new **File** (**filePath** + "news");  if( !f.exists() ) { // 저 경로(폴더)가 없을때  f.mkdirs(); //모든 경로를 다 만들어줌...  }  try {  **Files**.write(uploadPath, file.getBytes()); //transferTo 사용가능  } catch (IOException e) { //파일이 있을수도있고, 없을수도 있어서 예외  throw new RuntimeException(e);  }  **NewsFile** newsFile = **NewsFile**.builder()  .file\_origin\_name(originFileName)  .file\_temp\_name(tempFileName)  .build();  newsFileList .add(newsFile);  }  }  **News** news = **newsWriteReqDto**.toEntity("홍길동");  //toEntity()메소드에서 객체 이미 만들어둠  //newsWriteReqDto를 바로 toEntity로 만든 후 news에 대입  int result = **newsRepository**.**save**(news);  // namespace로 잡아뒀으므로 impl역할을 하는 .xml의 save메소드가 호출된다.  if(result == 0) { // 1이면 성공, 0이면 실패  return **ResponseEntity**.internalServerError()  .body(new **CMRespDto**<>(-1, "새 글 작성 실패", news ));  }  if(newsFileList != null) { // 파일이 첨부되었다면  for(**NewsFile** newsFile : newsFileList) {  newsFile.setNews\_id(news.getNews\_id());  // news\_mst에 있는 id를 저쪽에 set 해주는 과정  }  result = **newsRepository**.**saveFiles**(newsFileList); //@Mapper  // result값은 안에 있는 것의 개수. 실패시 0인 이유 = 아무것도 없으니까.  if(result != newsFileList.size()) {  return **ResponseEntity**.internalServerError()  .body(new CMRespDto<>(-1, "파일 업로드 실패", newsFileList));  }  }  **NewsWriteRespDto** newsWriteRespDto = news.toNewsWriteRespDto(newsFileList);  return **ResponseEntity**  .ok(new CMRespDto<>(1, "새 글 작성 완료", newsWriteRespDto));  }  **@GetMapping("/news/{newsId}")** // 업로드 파일 불러오는 페이지  public **ResponseEntity**<?> read(**@PathVariable** int newsId) {  **News** news = **newsRepository**.getNews(newsId); //@Mapper  **NewsReadRespDto** newsReadRespDto = news .toNewsReadRespDto();  return **ResponseEntity**  .ok(new CMRespDto<>(1, "게시글 불러오기 성공", newsReadRespDto));  } |

### **- [ FileReader() ]**

비동기적으로 파일의 내용을 읽어들이는 데 사용된다.

| var reader = new FileReader(); |
| --- |

| **속성** | **설명** | |
| --- | --- | --- |
| error | 읽기 도중에 에러가 발생한 경우  NOT\_FOUND\_ERR: 1 읽을 파일을 찾지 못했을 때  SECURITY\_ERR: 2 읽을 파일에 변경이 있을 때  ABORT\_ERR: 3 읽기가 중지되었을 때  NOT\_READABLE\_ERR: 4 파일에 접근할 수 없을 때  ENCODING\_ERR: 5 Data URL 길이가 상한을 초과했을 때 | |
| readyState | 파일의 상태를 숫자로 표현  EMPTY: 0 아직 데이터가 로드되지 않았음  LOADING: 1 데이터가 로딩 중…  DONE: 2 읽기 요청이 완료됨 | |
| result | 파일의 내용을 반환 | |

| **이벤트** | **설명** | |
| --- | --- | --- |
| onload | 읽기 동작이 성공적으로 완료되었을 때 발생 | |
| onerror | 읽기 동작에 에러가 생길 때마다 발생 | |
| onabort | 읽기 동작이 중단될 때마다 발생 | |
| onloadstart | 읽기 동작이 실행될 때마다 발생 | |
| onloadend | 읽기 동작이 끝났을 때마다 발생 (읽기 성공이나 실패 여부와 상관없이 발생) | |
| onprogress | 컨텐츠를 읽는 동안 호출 | |

| **메소드** | **설명** | |
| --- | --- | --- |
| .abort() | 읽기 요청을 중단시킴. (readyState 는 DONE이 됨) | |
| .readAsDataURL() | 바이너리 파일을 Base64 Encode 문자열로 반환  Ex.) data:image/jpeg; base64, GDYG…. | |
| .readAsText() | 텍스트 파일을 읽어 들임.  file: 읽어 들일 파일. 필수 사항  charset: 문자 코드. 기본값 UTF-8 | |

## **파일 다운로드**

downloadController. 다운로드 파일을 만들 경우에는 MIME으로 변환하는 header가 필요하다

| @Slf4j  @RequestMapping("/download")  @RestController  public class DownloadController {  @Value("${file.path}")  private String filePath;  @GetMapping("/news")  public **ResponseEntity<?>** download  (@RequestParam String originFileName, @RequestParam String tempFileName)  throws IOException {  **Path** path = **Paths**.**get**(filePath + " news/ " + tempFileName);  String contentType = Files.probeContentType(path);  //probeContentType 는 MIME형식으로 바꿔주는 역할  **ContentDisposition** contentDisposition = **ContentDisposition**.builder("attachment")  .filename(originFileName, StandardCharsets.UTF\_8)  .build(); // 원래 오리진파일명으로 다운받기 위함  **HttpHeaders** headers = new **HttpHeaders**(); //header가 있어야 다운로드가 됨  headers **.setContentDisposition**(contentDisposition);  headers **.add**(HttpHeaders.CONTENT\_TYPE, contentType );  **Resource** resource = new **InputStreamResource**(**Files.newInputStream**(path));  // 외부에 있는 데이터를 InputStream 객체를 통해 자바에 입력하겠다.  return **ResponseEntity**.ok().headers(headers).body(resource);  }  } |
| --- |

### **- [ MIME ]**

웹에서 파일의 확장자는 별 의미가 없다. 그러므로, 각 문서와 함께 올바른 MIME 타입을 전송하도록, 서버가 정확히 설정하는 것이 중요하다. 브라우저들은 리소스를 내려받았을 때 해야 할 기본 동작이 무엇인지를 결정하기 위해 대게 MIME 타입을 사용한다.

| type/subtype |
| --- |

MIME의 구조는 '/'로 구분된 두 개의 문자열인 타입과 서브타입으로 구성된다. 스페이스는 허용되지 않는다. type은 카테고리를 나타내며 개별(discrete) 혹은 멀티파트 타입이 될 수 있다. subtype 은 각각의 타입에 한정된다.

MIME 타입은 대소문자를 구분하지는 않지만 전통적으로 소문자로 쓰여진다.

#### **probeContentType()**

실제 파일의 내용이 아니라 파일의 확장자를 이용하여 마임타입을 판단한다.

| String mimeType = **Files.probeContentType**(source); |
| --- |

확장자가 없으면 null을 반환한다. 실제 파일이 존재하지 않아도 확장자로 타입을 반환한다.

#### **ContentDisposition**

MIME 프로토콜 Content-Disposition 헤더를 나타낸다.

| public class **ContentDisposition** |
| --- |

파일 첨부 파일의 ContentDisposition 경우 파일 크기를 설정하는 속성과 파일을 만든 날짜, 마지막 읽기 및 마지막으로 수정한 날짜를 사용할 수 있다. 모든 첨부 파일의 경우 첨부 파일이 받는 컴퓨터에 저장되는 경우 권장되는 파일 이름을 설정할 수 있다.

| **속성** | **설명** | |
| --- | --- | --- |
| CreationDate | 첨부 파일을 만든 날짜를 가져오거나 설정 | |
| DispositionType | 이메일 첨부 파일에 대한 처리 유형을 가져오거나 설정 | |
| FileName | 이메일 첨부 파일의 권장 파일 이름을 가져오거나 설정 | |
| Inline | 이메일 첨부 파일의 처리 유형(Inline 또는 Attachment)을 결정하는 Boolean 값을 가져오거나 설정 | |
| ModificationDate | 첨부 파일을 수정한 날짜를 가져오거나 설정 | |
| Parameters | 이 인스턴스에서 나타내는 Content-Disposition 헤더에 포함된 매개 변수를 가져온다 | |
| ReadDate | 첨부 파일을 읽은 날짜를 가져오거나 설정 | |
| Size | 첨부 파일의 크기를 가져오거나 설정 | |