Day 10 (2019-02-25)
1. Spring Web Application 개발

(1) RequestMapping

Controller에 @RequestMapping을 선언하여 작성한 메소드는 Client로부터의 요청을 받을 수 있는 HandlerMapping이 된다

RequestMapping 어노테이션이 아래와 같은 field를 가지며 default 값이 있기 때문에 생략 가능하다

또한 @Target에 Method, Type을 지원하므로 메소드 또는 Class에 선언할 수 있다

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Mapping
public @interface RequestMapping {

   String[] value() default {};

   RequestMethod[] method() default {};

   String[] params() default {};

   String[] headers() default {};

   String[] consumes() default {};

   String[] produces() default {};

}
```

필드	설명
value	요청을 처리하기 위한 URI 패턴을 지정한다 URI 패턴은 ANT 스타일의 와일드 카드를 지원한다 @RequestMapping("/hello") @RequestMapping("/main*") @RequestMapping("/view.*") @RequestMapping("/admin/**/user") value 필드는 배열이므로 URI 패턴을 여러개 설정할 수 있다. @RequestMapping({"/main", "/home"}) Path variable을 위한 URI 템플릿을 지원한다 @RequestMapping("/users/{userKey}")

method	HTTP Request Method를 정의한다 RequestMethod는 HTTP method를 정의한 열거형이다. GET, HEAD, POST, PATCH, PUT, DELETE, OPTIONS, TRACE 8개의 HTTP 메소드가 정의되어있다. @RequestMapping에 method를 추가하면 같은 URI이더라도 Request Method에 따라 분기할 수 있다 @RequestMapping(value = "/users/{userKey}", method = RequestMethod.GET) @RequestMapping(value = "/users/{userKey}", method = RequestMethod.PUT) @RequestMapping(value = "/users/{userKey}", method = RequestMethod.PATCH)
params	Request Parameter를 정의한다 같은 URI를 사용하더라도 HTTP Request Parameter에 따라서 별도의 분기가 필요하다면 params를 사용한다 @RequestMapping(value = "/users/{userKey}", params="type=admin") → /users/10?type="admin" 요청이 매핑된다 @RequestMapping(value = "/users/{userKey}", params="type=member") → /users/10?type="member" 요청이 매핑된다 @RequestMapping(value = "/users/{userKey}", params="!type") → type 파라미터가 없는경우 매핑된다
headers	Request Header를 정의한다 같은 URI를 사용하더라도 HTTP Request Header에 따라 별도의 분기가 필요하다면 headers를 사용한다 @RequestMapping(value = "/users/{userKey}", headers="content-type=text/*") → content-type 헤더가 text/plain, text/html, text/css 등 인것들에 대해 매핑된다
consumes	Content-Type Header에 매핑될 값을 정의한다 같은 URI를 사용하더라도 Content-Type Header 값에 따라 분기하려면 사용한다 @RequestMapping(value="/users/{userKey}", method=RequestMethod.POST, consumes="application/json") → Content-Type=application/json인 요청만 매핑한다

produces	Accept Header에 매핑될 값을 정의한다 같은 URI를 사용하더라도 Accept Header 값에 따라 분기하려면 사용한다 @RequestMapping(value="/users/{userKey}", method=RequestMethod.POST, produces="application/json") → Accept=application/json인 요청만 매핑한다

사용예

```
@RequestMapping("/users")
public class UserController {
        @RequestMapping(method=RequestMethod.POST)
        public ResponseEntity<User> addUser(){
        }


        @RequestMapping(method=RequestMethod.GET)
        public ResponseEntity<List<User>> getUsers(){
        }


        @RequestMapping(value="/{userKey}", method=RequestMethod.GET)
        public ResponseEntity<User> getUser(){
                //...
        }


        @RequestMapping(value="/{userKey}", method=RequestMethod.PUT)
        public ResponseEntity<User> updateUser(){
        }


        @RequestMapping(value="/{userKey}", method=RequestMethod.DELETE)
        public ResponseEntity<User> deleteUser(){
        }
}
```

1) GetMapping

Spring 4.3 이후에 등장한 RequestMapping을 위한 애노테이션이다

RequestMapping(method=RequestMethod.GET)을 포함하는 개념이다

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@RequestMapping(method = RequestMethod.GET)
public @interface GetMapping {

   /**
    * Alias for {@link RequestMapping#name}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String name() default "";

   /**
    * Alias for {@link RequestMapping#value}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] value() default {};

   /**
    * Alias for {@link RequestMapping#path}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] path() default {};

   /**
    * Alias for {@link RequestMapping#params}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] params() default {};

   /**
    * Alias for {@link RequestMapping#headers}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] headers() default {};

   /**
    * Alias for {@link RequestMapping#consumes}.
    * @since 4.3.5
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] consumes() default {};

   /**
    * Alias for {@link RequestMapping#produces}.
    */
   @AliasFor(annotation = RequestMapping.class)
   String[] produces() default {};

}
```

2) PostMapping

GetMapping와 동일하며 method=RequestMethod.POST가 선언되어있다

3) PutMapping

GetMapping와 동일하며 method=RequestMethod.PUT가 선언되어있다

4) DeleteMapping

GetMapping와 동일하며 method=RequestMethod.DELETE가 선언되어있다

사용예

```
@RequestMapping("/users")
public class UserController {
        @PostMapping()
        public ResponseEntity<User> addUser(){
        }

        @GetMapping()
        public ResponseEntity<List<User>> getUsers(){
        }

        @GetMapping(value="/{userKey}")
        public ResponseEntity<User> getUser(){
        }

        @PutMapping(value="/{userKey}")
        public ResponseEntity<User> updateUser(){
        }

        @DeleteMapping(value="/{userKey}")
        public ResponseEntity<User> deleteUser(){
        }
}
```

5) RequestParam과 PathVariable

Web 요청을 전달하면서 Client에서 보낸 전달하는 파라미터는 크게 두가지로 나뉜다

Request parameter

요청파라미터로 HTTP GET 메소드를 사용할 경우에 요청 URL 뒤에 ? 로 시작하는 Query parameter를 의미하며,

HTTP POST 메소드를 사용할 경우 Request body를 의미한다

Path variable

RESTful API가 대두되면서 Resource에 대한 ID를 URL에 표기함에 따라 사용하는 경로 변수

Spring에서는 이러한 유형의 파라미터를 Controller의 HandlerMapping에서 쉽게 사용할 수 있는 기술을 제공한다

● Request Parameter

[POST] /role-modify.json?role=1&name=changeRole&description=desc&reason=forChangeTest

사용예

```
@RequestMapping(value = "/role-modify.json", method = RequestMethod.POST)
public void modifySomeRoleName(ModelMap model,
                        Authentication authentication,
                        @RequestParam("role") int roleKey,
                        @RequestParam("name") String roleName,
                        @RequestParam("description") String description,
                        @RequestParam("reason") String reason) {
   User user = (User) authentication.getPrincipal();
   roleService.updateSomeManageRoleName(user.getUserKey(), user.getSponsorKey(), roleKey, roleName,
description, false, reason);
   model.clear();
   model.addAttribute("result", "succeed");
}
```

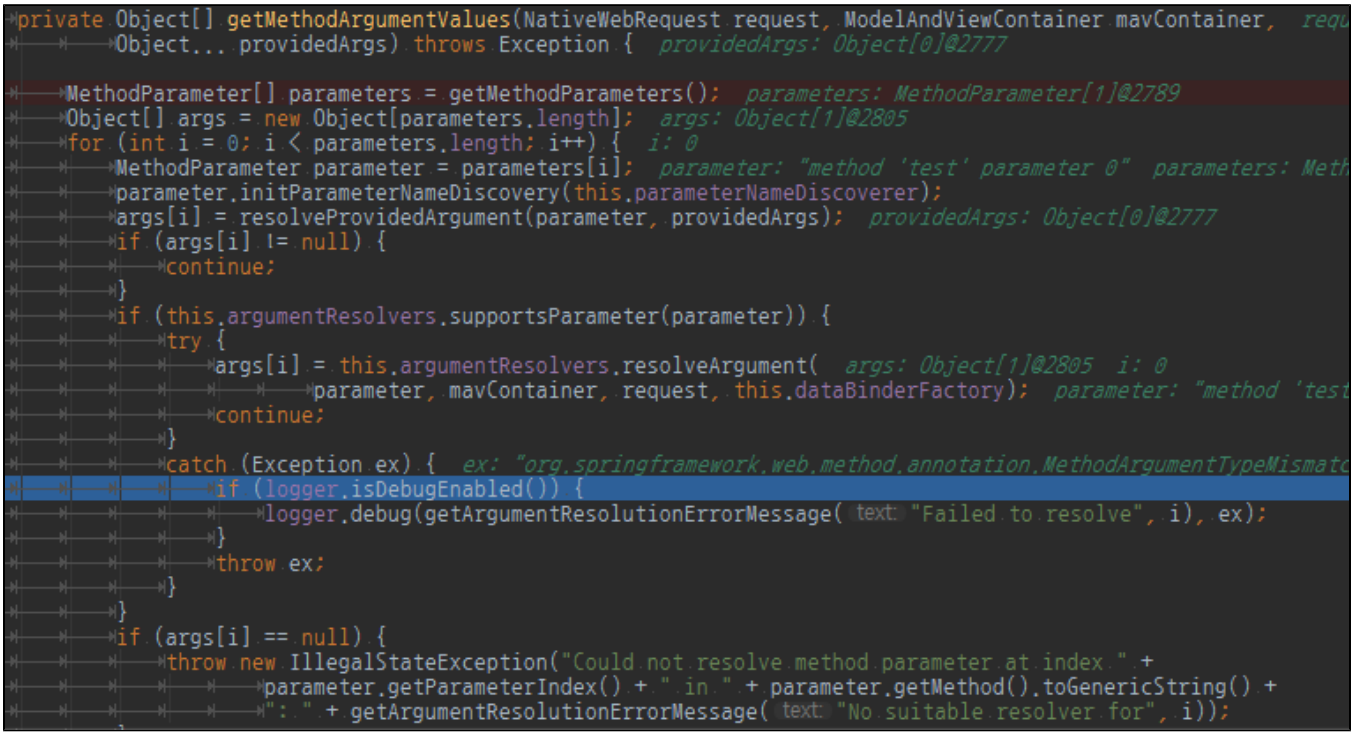
● Path variable

[GET] /users/10

사용예

```
@GetMapping(value = "/users/{key}", consumes = MediaType.APPLICATION_XML_VALUE,
        produces = MediaType.APPLICATION_XML_VALUE)
public ResponseEntity<UserForXML> getUserByXML(@PathVariable("key") Long key) {
    return ResponseEntity.ok(
            UserController.userRepository.stream()
                    .filter(user -> user.getKey().equals(key))
                    .findAny()
                    .map(UserForXML::fromUser)
                    .orElse(null));
}
```

※ 파라미터 바인딩 시 에러가 난다면?



(2) 커맨드 객체와 값 검증

1) Comment Object

앞서 소개한대로 Request body에 전달한 RequestParam 어노테이션을 통해서 처리할 수 있지만

이것은 매우 불편하고 번거로운 작업이다.

Spring에선 이를 위해 별도로 Command Object라는 개념을 제공한다

사용예

```
@GetMapping(value = "/command")
public User Command(@ModelAttribute User user) {
    return user;
}
```

HandlerMethod에 파라미터 선언을 Object 형태로 선언하면 Spring은 Request parameter(Query string 또는 Request body)를

바인딩하여 우리가 원하는 Object 타입에 해당하는 객체를 생성하여 전달하며 이를 Command Object라고 부른다

@ModelAttribute는 생략 가능하다

※파라미터 바인딩 시 발생하는 문제를 Application에서 처리하고 싶을때

앞서 @RequetParam을 이용할 때 파라미터 바인딩이 실패한 경우 MethodArgumentTypeMismatched 예외가 발생하며,
우리가 정의한 HandlerMethod는 실행되지 않고 400 Bad request로 응답이 완료되어 우리가 원하는 에러 응답을 내보낼 수 없다

Command object를 사용한다면 바인딩 관련 에러가 있더라도 HandlerMethod가 실행되도록 할 수 있는데
사용법은 아래와 같다

사용예

```
@GetMapping(value = "/binding-object-error")
@ResponseBody
public Object bindingErrorTest(CommandObj commandObj,
                               Errors errors) {
    return errors.getErrorCount();
}
```

Errors 또는 BindingResult 타입의 파라미터를 선언하면 Spring은 바인딩 에러에 대한 처리를

HandlerMethod로 위임하며, Errors 또는 BindingResult 객체에 에러와 관련한 정보를 전달한다

2) Bean Validation을 이용한 값 검증

요청에 대해 Front-end에서 항상 올바른 값이 전달될 것이라는 가정은 위험하다
따라서 Handler가 요청을 받은 후 Validation은 필수이다

Spring은 Bean validation이라는 방법을 통해 validation을 할수있도록 지원한다

아래의 의존설정 후

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.1.0.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.1.0.Final</version>
</dependency>
```

아래와 같이 우리가 사용할 Command object에 Annotation 기반으로 Validation 조건을 필드에 명시할 수 있다.

사용예

```
public class User {
    private Long key;

        @Email
        private String id;

        @NotNull(message = "User name must be not null")
        private String name;

        @Size(max = 11, min = 11)
        private String phone;
}
```

그 후 아래와 같이 @Valid` 어노테이션을 커맨드 객체에 붙이면 자동으로 Validation이 처리된다

사용예

```
@PostMapping(value = "/users")
public ResponseEntity<String> addUser(@Valid User user, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return ResponseEntity.badRequest()
                .body(bindingResult.getAllErrors().stream()
                        .map(objectError -> objectError.getObjectName() + ":" + objectError.getCode())
                        .collect(Collectors.joining("    ")));
    } else {
        return ResponseEntity.ok("success");
    }
}
```

3) Custom validatior를 구현하여 Validation 하기

앞선 방법을 사용하면 Request DTO에 일일이 어노테이션을 설정하여 처리해야한다

하지만 어노테이션을 이용해서 동적인 검사조건을 지정하려고 하면 아래와 같이 Validation group을 명시하는 형태로 정의해야한다. (참조자료)

사용예

```
public class User {
    private Long key;

    @Email
    private String id;

    @NotNull(message = "User name must be not null", groups = {})
    private String name;

}
```

여기에 만약 Lombok과 같이 전처리를 위한 어노테이션과 JPA를 위한 @Column 등의 어노테이션을 사용하고 있다고 가정하면 위 클래스는 매우 복잡해질 것인데

이를 Annotation hell이라고 칭한다

(보통 JPA의 Entity와 Request DTO는 분리한다)

이러한 사태를 막기위해 Spring에선 InitBinder라는 것을 통해 커스텀 validator를 등록하여 사용되게 합니다

먼저 아래와 같이 Validator 인터페이스를 구현한 컴포넌트를 생성합니다

validation 메소드에서 전달된 Object의 값 검증을 진행합니다.

사용예

```
@Component("userValidatorForAdd")
public class UserValidatorForAdd implements Validator {

    @Override
    public boolean supports(Class<?> aClass) {
        return User.class.equals(aClass);
    }

    @Override
    public void validate(Object o, Errors errors) {
        User user = (User) o;

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "id", "user.id.empty", "user id must not be empty");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "name", "user.name.empty", "user name must not be
empty");

        if (user.getPhone() != null && user.getPhone().contains("-")) {
            errors.rejectValue("phone", "user.phone.invalid");
        }
    }
}
```

.그후 아래와 같이 컨트롤러에 InitBinder에서 우리가 정의한 Validator를 등록합니다

이후 User타입의 Command objet의 바인딩 시 우리가 등록한 Validator가 동작합니다

주의할 점은 해당 컨트롤러가 아닌 다른 곳에 존재하는 User 타입의 Command object에 대해서도 validation을 진행합니다

→ SRP 원리를 잘 지켜야 함을 생각할 수 있습니다

사용예

```
@Controller
public class UserAddController {
    @Autowired
    @Qualifier("userValidatorForAdd")
    private Validator validator;

    @InitBinder
    private void initBinder(WebDataBinder binder) {
        binder.setValidator(validator);
    }

    @PostMapping(value = "/users")
    public ResponseEntity<String> addUser(@Valid User user, BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return ResponseEntity.badRequest()
                    .body(bindingResult.getAllErrors().stream()
                            .map(objectError -> objectError.getObjectName() + ":" + objectError.getCode() + ":"
+ objectError.getDefaultMessage())
                            .collect(Collectors.joining("    ")));
        } else {
            return ResponseEntity.ok("success");
        }
    }

}
```

여기서 WebDataBinder는 Command object를 바인딩하는 작업을 처리하는 객체입니다.