스프링 MVC 활용

Spring Web MVC

스프링 MVC 핵심 기술 소개

30. 스프링 MVC 핵심 기술 소개

애노테이션 기반의 스프링 MVC

- 요청 맵핑하기
- 핸들러 메소드
- 모델과 뷰
- 데이터 바인더
- 예외 처리
- 글로벌 컨트롤러

사용할 기술

- 스프링 부트
- 스프링 프레임워크 웹 MVC
- 타임리프

학습 할 애노테이션

- @RequestMapping
  - @GetMapping, @PostMapping, @PutMapping, ...
- @ModelAttribute
- @RequestParam, @RequestHeader
- @PathVariable, @MatrixVariable
- @SessionAttribute, @RequestAttribute, @CookieValue
- @Valid
- @RequestBody, @ResponseBody
- @ExceptionHandler
- @ControllerAdvice

스프링 MVC에는 요청 매핑하기, 핸들러 메소드, 모델과 뷰와 같은 핵심적인 기술들이 있으며 이외에도 데이터 바인더, 예외처리, 글로벌 컨트롤러와 같이 위 3가지 보다는 핵심적이지는 않지만 중요한 기술들이 있다.

요청 매핑하기 1부 HTTP Method

```java
@Controller
public class SampleController {

    @RequestMapping
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

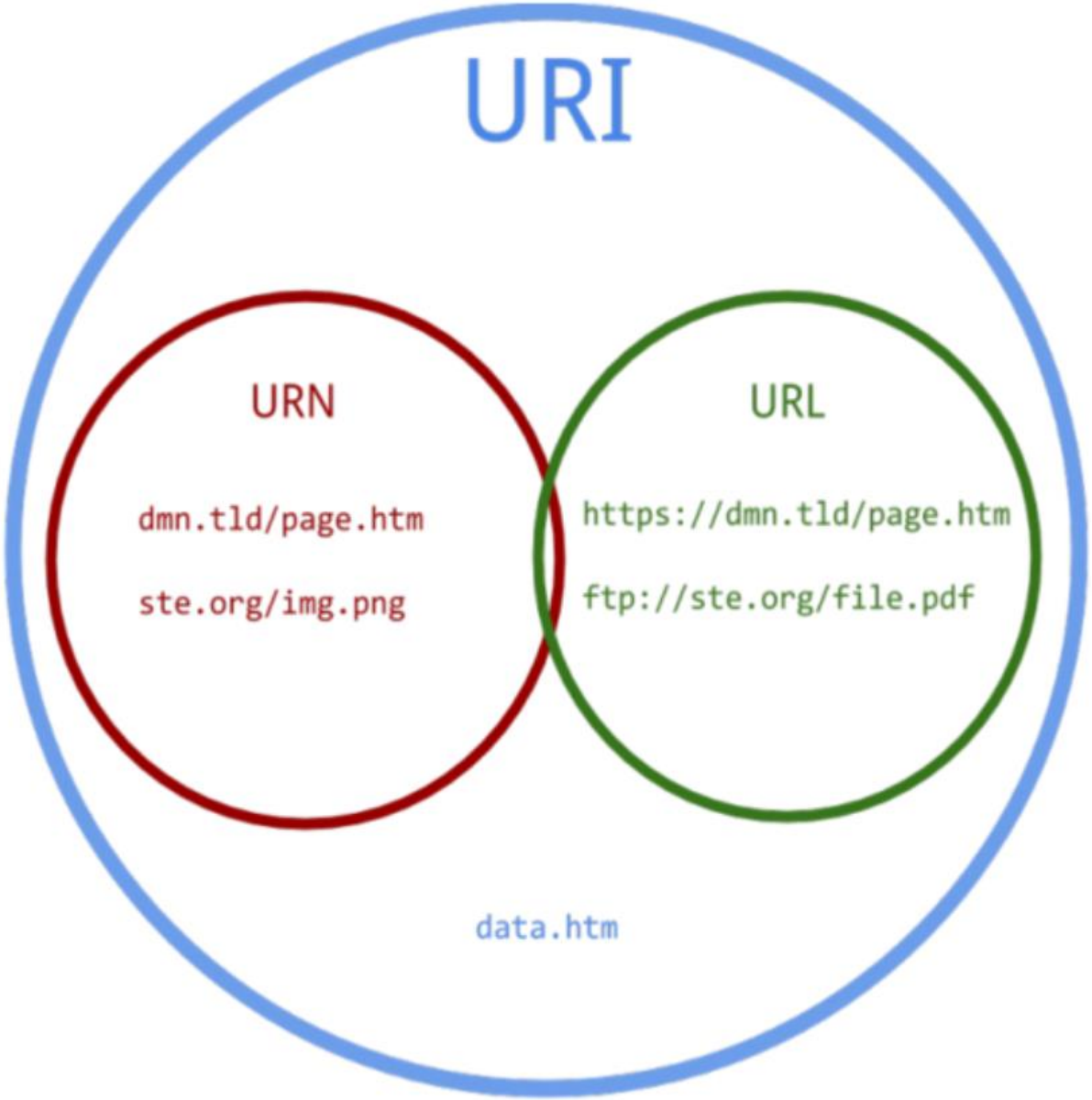
핸들러란 요청을 처리할 수 있는 메소드들을 말하며 @Controller 애노테이션을 붙이고 정의를함.

요청을 매핑할경우 @RequestMapping 애노테이션을 붙여주고

문자열을 그대로 응답으로 보내고 싶으면 @ResponseBody 애노테이션을 붙임!

```java
@RequestMapping("/hello")
```

특정 URI패턴이 들어올때 응답하도록 다음과 같이 설정할 수 있음



서버 리소스 이름은 통합 자원 식별자(uniform resource identifier) 혹은 URI라고 불린다.
URI는 인터넷의 우편물 주소 같은 것으로, 정보 리소스를 고유하게 식별하고 위치를 지정할 수 있다.
그리고 이 URI에는 두 가지 형태가 있는데 이것이, URL, URN이라는 것이다.

통합 자원 지시자(uniform resource locator, URL)는 URI의 가장 흔한 형태이다.
URL은 특정 서버의 한 리소스에 대한 구체적인 위치를 서술한다.
URL은 리소스가 정확히 어디에 있고 어떻게 접근할 수 있는지 분명히 알려준다.
예를 들자면 아래와 같다. (우리가 흔히 보는 바로 그것이다)

요청 매핑하기 1부 HTTP Method

```java
@Controller
public class SampleController {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

```java
@RunWith(SpringRunner.class)
@WebMvcTest
public class SampleControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void helloTest() throws Exception {
        mockMvc.perform(put( urlTemplate: "/hello"))
                .andDo(print())
                .andExpect(status().isMethodNotAllowed())
        ;
    }

}
```

다음과 같이 GET 요청을 받는다고 Controller에 명시하였을 시 테스트로 PUT 요청을 보내게 되면 에러가 발생함.(405 ERROR)

아래와 같이 GET요청만 받는경우 GetMapping 애노테이션으로 줄여서 사용할 수 있음(다른 요청도 마찬가지)

```java
@Controller
public class SampleController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

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

요청 매핑하기 1부 HTTP Method

HTTP Method
- GET, POST, PUT, PATCH, DELETE, ...

GET 요청
- 클라이언트가 서버의 리소스를 요청할 때 사용한다.
- 캐싱 할 수 있다. (조건적인 GET으로 바꿀 수 있다.)
- 브라우저 기록에 남는다.
- 북마크 할 수 있다.
- 민감한 데이터를 보낼 때 사용하지 말 것. (URL에 다 보이니까)
- idempotent

POST 요청
- 클라이언트가 서버의 리소스를 수정하거나 새로 만들 때 사용한다.
- 서버에 보내는 데이터를 POST 요청 본문에 담는다.
- 캐시할 수 없다.
- 브라우저 기록에 남지 않는다.
- 북마크 할 수 없다.
- 데이터 길이 제한이 없다.

PUT 요청
- URI에 해당하는 데이터를 새로 만들거나 수정할 때 사용한다.
- POST와 다른 점은 "URI"에 대한 의미가 다르다.
  - POST의 URI는 보내는 데이터를 처리할 리소스를 지칭하며
  - PUT의 URI는 보내는 데이터에 해당하는 리소스를 지칭한다.
- Idempotent

스프링 웹 MVC에서 HTTP method 맵핑하기
- @RequestMapping(method=RequestMethod.GET)
- @RequestMapping(method={RequestMethod.GET, RequestMethod.POST})
- @GetMapping, @PostMapping, ...

요청 매핑하기 2부 URI 패턴

요청 식별자로 맵핑하기

- @RequestMapping은 다음의 패턴을 지원합니다.
- ?: 한 글자 (“/author/???” => “/author/123”)
- *: 여러 글자 (“/author/*” => “/author/keesun”)
- **: 여러 패스 (“/author/** => “/author/keesun/book”)

클래스에 선언한 @RequestMapping과 조합

- 클래스에 선언한 URI 패턴뒤에 이어 붙여서 맵핑합니다.

정규 표현식으로 맵핑할 수도 있습니다.

- /{name:정규식}

패턴이 중복되는 경우에는?

- 가장 구체적으로 맵핑되는 핸들러를 선택합니다.

URI 확장자 맵핑 지원

- 이 기능은 권장하지 않습니다. (스프링 부트에서는 기본으로 이 기능을 사용하지 않도록 설정 해 줌)
  - 보안 이슈 (RFD Attack)
  - URI 변수, Path 매개변수, URI 인코딩을 사용할 때 할 때 불명확 함.

```java
@Controller
@RequestMapping("/hello")
public class SampleController {

    @RequestMapping("/keesun")
    @ResponseBody
    public String hello(@PathVariable String name) {
        return "hello keesun";
    }

}
```

정규표현식에서 Path에 들어있는 값을 확인하기 위해 @Pathvariable 이라는 애노테이션을 메소드에 적용시켜서 확인함

요청 매핑하기 3부 미디어 타입

```java
@Controller
public class SampleController {

    @GetMapping(
            value = "/hello",
            consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

다음과 같이 특정한 타입의 데이터를 담고 있는 요청만 처리하는 핸들러를 만들기 위해

Content-Type 헤더로 필터링을 해줌(consumes = MediaType~ 을 이용하여)

요청 매핑하기 3부 미디어 타입

```java
@RunWith(SpringRunner.class)
@WebMvcTest
public class SampleControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void helloTest() throws Exception {
        mockMvc.perform(get( urlTemplate: "/hello")
                        .contentType(MediaType.APPLICATION_JSON)
                        .accept(MediaType.APPLICATION_JSON)
        )
                .andDo(print())
                .andExpect(status().isOk())
        ;
    }

}
```

반대로 특정한 타입의 응답을 받기를 원하는 부분에서 accept 헤더를 이용하여 자신이 원하는 응답을 받을 수 있음

미디어 타입의 경우 URI와 달리 클래스에 선언한 매핑과 조합되지 않고 메소드 각자 덮어씀.

요청 매핑하기 4부 헤더와 파라미터 매핑

특정한 헤더가 있는 요청을 처리하고 싶은 경우

- @RequestMapping(**headers** = “key”)

특정한 헤더가 없는 요청을 처리하고 싶은 경우

- @RequestMapping(**headers** = “!key”)

특정한 헤더 키/값이 있는 요청을 처리하고 싶은 경우

- @RequestMapping(**headers** = “key=value”)

특정한 요청 매개변수 키를 가지고 있는 요청을 처리하고 싶은 경우

- @RequestMapping(**params** = “a”)

특정한 요청 매개변수가 없는 요청을 처리하고 싶은 경우

- @RequestMapping(**params** = “!a”)

특정한 요청 매개변수 키/값을 가지고 있는 요청을 처리하고 싶은 경우

- @RequestMapping(**params** = “a=b”)

요청 매핑하기 4부 헤더와 파라미터 매핑 예시 코드

```java
@Controller
public class SampleController {

    @GetMapping(value = "/hello", params = "name=spring")
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

```java
@RunWith(SpringRunner.class)
@WebMvcTest
public class SampleControllerTest {

    @Autowired
    MockMvc mockMvc;

    @Test
    public void helloTest() throws Exception {
        mockMvc.perform(get( urlTemplate: "/hello")
                        .param( name: "name",  ...values: "keesun"))
                .andDo(print())
                .andExpect(status().isOk())
        ;
    }

}
```

HTTP 요청 매핑하기 5부 Head와 Option

```java
@Test
public void helloTest() throws Exception {
    mockMvc.perform(head( urlTemplate: "/hello")
                    .param( name: "name",  ...values: "keesun"))
            .andDo(print())
            .andExpect(status().isOk())
    ;
}
```

```
MockHttpServletResponse:
           Status = 200
    Error message = null
          Headers = [Content-Type:"text/plain;charset=UTF-8", Content-Length:"5"]
     Content type = text/plain;charset=UTF-8
             Body =
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
2019-01-19 09:36:35.135  INFO 1919 --- [       Thread-1] o.s.s.concurrent.ThreadPoolTaskExe
```

HTTP 요청 매핑하기 5부 Head와 Option

```java
@Test
public void helloTest() throws Exception {
    mockMvc.perform(options( urlTemplate: "/hello"))
            .andDo(print())
            .andExpect(status().isOk())
    ;
}
```

```
MockHttpServletResponse:
           Status = 200
    Error message = null
          Headers = [Allow:"GET,HEAD,POST,OPTIONS"]
     Content type = null
             Body =
    Forwarded URL = null
   Redirected URL = null
          Cookies = []
2019-01-19 09:38:54.502  INFO 1972 --- [       Thread-1] o.s.s.concurrent.ThreadPoolTaskExe

Process finished with exit code 0
```

요청 매핑하기 6부 커스텀 애노테이션

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

메타(Meta) 애노테이션

- 애노테이션에 사용할 수 있는 애노테이션
- 스프링이 제공하는 대부분의 애노테이션은 메타 애노테이션으로 사용할 수 있다.

요청 매핑하기 6부 커스텀 애노테이션

```java
@Controller
public class SampleController {

    @GetHelloMapping
    @ResponseBody
    public String hello() {
        return "hello";
    }

}
```

```java
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@RequestMapping(method = RequestMethod.GET, value = "/hello")
public @interface GetHelloMapping {
}
```

자신이 Getmapping만을 주로 쓴다면 직접 인터페이스를 구현하여서 애노테이션을 커스터마이즈하여서 사용할 수 있음. 여기서 Retention의 경우 자바가 바이트코드가 로드될때 메모리에 올라가서 애노테이션들이 사라지기 때문에 Runtime까지 갖고 있게하도록 Runtime을 붙여주고 Target의 경우 어디에서 사용할 것인지 지정해주는 것.

요청 매핑하기 6부 커스텀 애노테이션

@RequestMapping 애노테이션을 메타 애노테이션으로 사용하기
- @GetMapping 같은 커스텀한 애노테이션을 만들 수 있다.

메타(Meta) 애노테이션
- 애노테이션에 사용할 수 있는 애노테이션
- 스프링이 제공하는 대부분의 애노테이션은 메타 애노테이션으로 사용할 수 있다.

조합(Composed) 애노테이션
- 한개 혹은 여러 메타 애노테이션을 조합해서 만든 애노테이션
- 코드를 간결하게 줄일 수 있다.
- 보다 구체적인 의미를 부여할 수 있다.

@Retention
- 해당 애노테이션 정보를 언제까지 유지할 것인가.
- Source: 소스 코드까지만 유지. 즉, 컴파일 하면 해당 애노테이션 정보는 사라진다는 이야기.
- Class: 컴파인 한 .class 파일에도 유지. 즉 런타임 시, 클래스를 메모리로 읽어오면 해당 정보는 사라진다.
- Runtime: 클래스를 메모리에 읽어왔을 때까지 유지! 코드에서 이 정보를 바탕으로 특정 로직을 실행할 수 있다.

@Target
- 해당 애노테이션을 어디에 사용할 수 있는지 결정한다.

@Documented
- 해당 애노테이션을 사용한 코드의 문서에 그 애노테이션에 대한 정보를 표기할지 결정한다.

핸들러 메소드 1부 아규먼트와 리턴 타입

핸들러 메소드 아규먼트: 주로 요청 그 자체 또는 요청에 들어있는 정보를 받아오는데 사용한다.

핸들러 메소드 아규먼트	설명
WebRequest NativeWebRequest ServletRequest(Response) HttpServletRequest(Response)	요청 또는 응답 자체에 접근 가능한 API
InputStream Reader OutputStream Writer	요청 본문을 읽어오거나, 응답 본문을 쓸 때 사용할 수 있는 API
PushBuilder	스프링 5, HTTP/2 리소스 푸쉬에 사용
HttpMethod	GET, POST, ... 등에 대한 정보
Locale TimeZone ZoneId	LocaleResolver가 분석한 요청의 Locale 정보
@PathVariable	URI 템플릿 변수 읽을 때 사용
@MatrixVariable	URI 경로 중에 키/값 쌍을 읽어 올 때 사용
@RequestParam	서블릿 요청 매개변수 값을 선언한 메소드 아규먼트 타입으로 변환해준다. 단순 타입인 경우에 이 애노테이션을 생략할 수 있다.
@RequestHeader	요청 헤더 값을 선언한 메소드 아규먼트 타입으로 변환해준다.
...	...

핸들러 메소드 1부 아규먼트와 리턴 타입

핸들러 메소드 리턴: 주로 응답 또는 모델을 랜더링할 뷰에 대한 정보를 제공하는데 사용한다.

@ResponseBody	리턴 값을 HttpMessageConverter를 사용해 응답 본문으로 사용한다.
HttpEntity ReponseEntity	응답 본문 뿐 아니라 헤더 정보까지, 전체 응답을 만들 때 사용한다.

String	ViewResolver를 사용해서 뷰를 찾을 때 사용할 뷰 이름.
View	암묵적인 모델 정보를 랜더링할 뷰 인스턴스
Map Model	(RequestToViewNameTranslator를 통해서) 암묵적으로 판단한 뷰 랜더링할 때 사용할 모델 정보
@ModelAttribute	(RequestToViewNameTranslator를 통해서) 암묵적으로 판단한 뷰 랜더링할 때 사용할 모델 정보에 추가한다. 이 애노테이션은 생략할 수 있다.
...	...

핸들러 메소드 2부 URI 패턴

@PathVariable

- 요청 URI 패턴의 일부를 핸들러 메소드 아규먼트로 받는 방법.
- 타입 변환 지원.
- (기본)값이 반드시 있어야 한다.
- Optional 지원.

@MatrixVariable

- 요청 URI 패턴에서 키/값 쌍의 데이터를 메소드 아규먼트로 받는 방법
- 타입 변환 지원.
- (기본)값이 반드시 있어야 한다.
- Optional 지원.
- 이 기능은 기본적으로 비활성화 되어 있음. 활성화 하려면 다음과 같이 설정해야 함.

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void configurePathMatch(PathMatchConfigurer configurer) {
    UrlPathHelper urlPathHelper = new UrlPathHelper();
    urlPathHelper.setRemoveSemicolonContent(false);
    configurer.setUrlPathHelper(urlPathHelper);
  }
}
```

둘다 타입변환이 지원이 가능하고 Matrix의 경우에는 직접 WebConfig 설정을 해 주어야 인식을 함!

핸들러 메소드 2부 URI 패턴

```java
@Controller
public class SampleController {

    @GetMapping("/events/{id}")
    @ResponseBody
    public Event getEvent(@PathVariable("id") Integer idValue, @MatrixVariable St
        Event event = new Event();
        event.setId(id);
        event.setName(name);
        return event;
    }

}
```

Pathvariable의 경우 파라미터가 매핑되는 값과 다른 변수이면 위의 그림과 같이 id라고 지정을 해 주어야하기 때문에 보통은 변수이름을 매핑되는 값과 같게 함.

핸들러 메소드 3부 요청 매개변수

```java
@Controller
public class SampleController {

    @GetMapping("/events/{id}")
    @ResponseBody
    public Event getEvent(@RequestParam String name) {
        Event event = new Event();
        event.setId(id);
        event.setName(name);
        return event;
    }

}
```

요청 매개변수에 들어있는 단순 타입 데이터를 메소드 아규먼트로 받아올 수 있으며 이 애노테이션은 생략 할 수 있다.

핸들러 메소드 4부 폼 서브밋

요청 매개변수로 폼 데이터또한 처리가 가능하다.

```java
@Controller
public class SampleController {

    @GetMapping("/events/form")
    public String eventsForm(Model model) {
        Event newEvent = new Event();
        newEvent.setLimit(50);
        model.addAttribute( s: "event", newEvent
        return "events/form";
    }
```

```html
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Create New Event</title>
</head>
<body>
<form action="#" th:action="@{/events}" method="post" th:object="${event}">
    <input type="text" title="name" th:field="*{name}"/>
    <input type="text" title="limit" th:field="*{limit}"/>
    <input type="submit" value="Create"/>
</form>

</body>
</html>
```

```
{"id":null,"name":"spring web mvc","limit":20}
```

핸들러 메소드 5부 @ModelAttribute

```
    @PostMapping("/events/name/{name}")
    @ResponseBody
    public Event getEvent(@Valid @ModelAttribute Event event, BindingResult bind
        if(bindingResult.hasErrors()) {
            System.out.println("=========================");
            bindingResult.getAllErrors().forEach(c -> {
                System.out.println(c.toString());
            });
        }
        return event;
    }

}
```

```
    @Test
    public void postEvent() throws Exception {
        mockMvc.perform(post( urlTemplate: "/events?name=keesun")
                    .param( name: "limit",  ...values: "20"))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath( expression: "name").value( expectedValue: "keesur
        ;
    }
```

Param이나 path 등등 다양한 곳에있는 단순 타입 데이터들을 복합 타입객체로 받아오거나 새로 객체를 만들때 사용할 수 있으며 Binding Result의 경우 값을 바인딩한 결과를 표시해주고 바인딩 이후 검증 작업은 Valid 애노테이션으로 함!

핸들러 메소드 6부 @Validated

```java
public class Event {

    interface ValidateLimit {}
    interface ValidateName{}

    private Integer id;

    @NotBlank(groups = ValidateName.class)
    private String name;

    @Min(value = 0, groups = ValidateLimit.class)
    private Integer limit;
```

스프링 MVC 핸들러 메소드 아규먼트에 사용할 수 있으며 validation group이라는 힌트를 사용할 수 있다.

@Valid 애노테이션에는 그룹을 지정할 방법이 없다.

@Validated는 스프링이 제공하는 애노테이션으로 그룹 클래스를 설정할 수 있다.

```java
@PostMapping("/events/name/{name}")
@ResponseBody
public Event getEvent(@Validated(Event.ValidateLimit.class) @ModelAttribute
    if(bindingResult.hasErrors()) {
        System.out.println("==========================");
        bindingResult.getAllErrors().forEach(c -> {
            System.out.println(c.toString());
        });
    }
    return event;
}
```

핸들러 메소드 7부 폼 서브밋 에러 처리

```java
@PostMapping("/events")
public String getEvent(@Validated @ModelAttribute Event event,
                       BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {
        return "/event/form";
    }
    return "/event/list";
}
```

```java
@Test
public void postEvent() throws Exception {
    ResultActions result = mockMvc.perform(post( urlTemplate: "/events")  resul
            .param( name: "name",  ...values: "keesun")
            .param( name: "limit",  ...values: "-10"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(model().hasErrors());
    ModelAndView mav = result.andReturn().getModelAndView();  mav: "ModelAnd
    Map<String, Object> model = mav.getModel();  model:  size = 2  mav: "Mod
    System.out.print(model.size());  model:  size = 2
}
```

```java
@PostMapping("/events")
public String createEvent(@Validated @ModelAttribute Event event,
                          BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {
        return "/events/form";
    }

    return "redirect:/events/list";
}

@GetMapping("/events/list")
public String getEvents(Model model) {
    Event event = new Event();
    event.setName("spring");
    event.setLimit(10);

    List<Event> eventList = new ArrayList<>();
    eventList.add(event);

    model.addAttribute(eventList);

    return "/events/list";
}
}
```