Spring MVC 09

[MVC 기본 기능 02]

스프링 MVC 1편 - 백엔드 웹 개발 핵심 기술

김영한

2022.08.27

@ModelAttribute

요청 파라미터를 받아 객체를 만들고 객체에 값을 넣어주는 기능 → 스프링이 지원

```
@RequestMapping("/model-attribute-v1")
@ResponseBody
public String modelAttribute(@RequestParam String username, @RequestParam int age){

   HelloData helloData = new HelloData();
   helloData.setUsername(username);
   helloData.setAge(age);
   log.info("username={}, age={}", helloData.getUsername(), helloData.getAge());
   return "ok";
}
```

```
@RequestMapping("/model-attribute-v1")
@ResponseBody
public String modelAttribute(@ModelAttribute HelloData helloData){
   log.info("username={}, age={}", helloData.getUsername(), helloData.getAge());
   return "ok";
}
```

(@ModelAttribute HelloData helloData) 는 프로퍼티를 찾아서 프로퍼티의 setter를 호출해서 값을 넣어준다. 즉, new HelloData(), set 프로퍼티를 넣어주는 것.

만약 숫자를 넣어야 할 곳에 age=abc 처럼 잘못 넣으면 BindException이 발생한다.

@RequestParam 처럼 @ModelAttribute 도 생략 가능하다.

ArgumentResolver를 제외한 타입에 대해

RequestParam ⇒ String, int, Integer 같은 단순타입

ModelAttribute ⇒ 단순타입을 제외한 나머지

HTTP message body에 데이터를 직접 담아서 요청

HTTP message body 에 데이터를 직접 담아서 요청 하게 된다면?

쿼리파라미터 형식이 아닌 경우 @RequestParam, @ModelAttribute를 사용할 수 없다.

Text 를 받는 경우

```
[1]--------------------------------
@PostMapping("/request-body-string-v1")
public void requestBodyString(HttpServletRequest request, HttpServletResponse
response) throws IOException {

   ServletInputStream inputStream = request.getInputStream();
   String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);
   log.info("messageBody={}", messageBody);
   response.getWriter().write("ok");

}
[2]--------------------------------
@PostMapping("/request-body-string-v2")
public void requestBodyStringV2(InputStream inputStream, Writer responseWriter)
throws IOException {

   String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);
   log.info("messageBody={}", messageBody);
   responseWriter.write("ok");

}
```

ArgumentResolver로 이미 **inpustStream, Writer**가 예약되어 있어서 사용하면 된다**.** 같은 결과가 나오는것**.**

```
@PostMapping("/request-body-string-v3")
public HttpEntity<String> requestBodyStringV3(HttpEntity<String> httpEntity) throws
IOException {
```

```
    String messageBody = httpEntity.getBody();
    log.info("messageBody={}", messageBody);

    return new HttpEntity<>("ok");
}
```

스프링 **MVC**가 지원하는 **HttpEntity** 사용

메시지 바디 정보를 직접 조회 가능, 이것은 요청 파라미터를 조회하는 기능과 관계가 없다

응답에도 사용 가능하다.

메시지 정보 직접 반환 가능하고 헤더 정보도 포함 가능하다.

RequestEntity, ResponseEntity로 사용 가능하다.

```
@PostMapping("/request-body-string-v3-1")
public HttpEntity<String> requestBodyStringV3_1(RequestEntity<String> httpEntity)
throws IOException {

    String messageBody = httpEntity.getBody();
    log.info("messageBody={}", messageBody);

    return new ResponseEntity<String>("ok", HttpStatus.CREATED);
}
```

이를 더욱 편하게 애노테이션을 지원한다.

```
@PostMapping("/request-body-string-v4")
@ResponseBody
public String requestBodyStringV4(@RequestBody String messageBody) throws
IOException {

    log.info("messageBody={}", messageBody);

    return "ok";
}
```

@ResponseBody, @RequestBody로 간단히 쓸수 있다. → 둘은 요청 파라미터와는 관계가 없이 바디를 직접 조회한다.

JSON 을 받는 경우

```
private ObjectMapper objectMapper = new ObjectMapper();

@PostMapping("/request-json-v1")
public void requestBodyJsonV1(HttpServletRequest request, HttpServletResponse
response) throws IOException {

   ServletInputStream inputStream = request.getInputStream();
   String messageBody = StreamUtils.copyToString(inputStream,
StandardCharsets.UTF_8);

   log.info("messageBody={}", messageBody);
   HelloData helloData = objectMapper.readValue(messageBody, HelloData.class);
   log.info("username={} , age= {} ", helloData.getUsername(), helloData.getAge());
   response.getWriter().write("ok");
}
→ @RequestBody, @ResponseBody 사용하는 것
@ResponseBody
@PostMapping("/request-json-v2")
public String requestBodyJsonV2(@RequestBody String messageBody) throws IOException
{

   log.info("messageBody={}", messageBody);
   HelloData helloData = objectMapper.readValue(messageBody, HelloData.class);
   log.info("username={} , age= {} ", helloData.getUsername(), helloData.getAge());

   return "ok";
}
→ @RequestBody에
@ResponseBody
@PostMapping("/request-json-v3")
public String requestBodyJsonV3(@RequestBody HelloData helloData){

   log.info("username={} , age= {} ", helloData.getUsername(), helloData.getAge());

   return "ok";
}
```

@RequestBody 를 사용하게 되면 HTTP 메시지 컨버터가 HTTP 메시지 바디의 내용을 우리가 원하는 문자나 객체로 변환해 준다. 생략이 불가능하다.

* HttpMessageConverter 사용 -> MappingJackson2HttpMessageConverter (contenttype: application/json 가 사용된것.

HttpEntity로도 바디를 받아 사용할 수 있다.

```
@ResponseBody
@PostMapping("/request-json-v4")
public String requestBodyJsonV4(HttpEntity<HelloData> httpEntity){
   HelloData helloData = httpEntity.getBody();
   log.info("username={} , age= {} ", helloData.getUsername(), helloData.getAge());

   return "ok";
}
```

마지막으로 ResponseBody 같은 경우에도 HTTP 컨버터가 작동해서 보내는 형식을 변경해 준다.

```
@ResponseBody
@PostMapping("/request-json-v5")
public HelloData requestBodyJsonV5(@RequestBody HelloData helloData){
   log.info("username={} , age= {} ", helloData.getUsername(), helloData.getAge());

   return helloData;
}
```

컨버터가 변경하는 형식은 헤더에 **Accept**을 따르게 된다**.**

HTTP 응답 **3**가지

정적 리소스**,** 뷰 템플릿**, HTTP** 메시지

정적 리소스

src/resources/static 에 담아둔 자원은 정적으로 내장톰켓이 서빙을해준다.

정적리소스는 파일을 변경없이 그대로 실행하는것.

뷰 템플릿

HTML을 동적으로 생성하는 용도로 대부분 사용, 다른 형식으로 템플릿도 지원한다.

스프링 부트는 기본적으로 resources/templates 경로.

return 방식 String, void, ModelAndView 형식으로 템플릿에서 위치를 찾아준다.

HTTP 응답 **- HTTP API,** 메시지 바디에 직접 입력

응답은 결국 메시지바디에 데이터를 담아 보내는 것.

@ResponseBody + @Controller = @RestController

HTTP 메시지 컨버터

@ResponseBody를 사용하게 되면

viewResolver 대신에 HttpMessageConverter가 동작해서 타입에 따라 바디에 내용을 담아 응답해준다.

응답의 경우 : 클라이언트 HTTP Accept 해더와 서버의 컨트롤러 반환 타입 정보 등을 조합해서 HttpMessageConverter가 선택된다.

- HTTP 요청 : @RequestBody, HttpEntity(RequestEntity)
- HTTP 응답 : @ResponseBody, HttpEntity(ResponseEntity)

두가지 경우 모두에서 HttpMessageConverter가 사용된다.

[Byte-> String -> MappingJackson2 의 우선 순위로 컨버터를 결정하게 된다.]

(canRead, canWrite 메소드로 사용여부를 파악)

요청 → RequestBody, HttpEntity 파라미터 사용 → 메시지 컨버터 작동

→ 사용가능 컨버터 확인 (Content-Type 미디어 타입확인) → 조건이 맞는 객체 생성, 반환

응답 → 컨트롤러에서 @ResponseBody HttpEntity로 값이 반환 됨

→ 사용 가능한 메시지 컨버터를 확인 → (Accept 타입 지원확인) → write()를 호출해서 HTTP메시지 바디에 데이터를 생성한다.

요청 매핑 핸들러 어댑터 구조

HTTP 메시지 컨버터는 핸들러 어댑터에 관련되어 있고

RequestMapping 핸들러 어댑터 → 컨트롤러 호출할때 요청과 응답 객체의 처리를 어댑터가 맞는형식에 맞게 변환해서 호출해 줘야 한다.

즉, 요청 시 **ArgumentResolver**가 사용되어 요청 파라미터를 처리해준다**.**

핸들러 어댑터가 ArgumentResolver를 호출하고 처리된 파라미터를 컨트롤러로 넘겨준다.

이는 인터페이스로 직접 구현해서 파라미터를 처리할 수 있다.

응답 시에는 **ReturnValueHandler** 가 응답시에 호출되어 반환 타입을 변환하고 처리한다**.**

그렇다면 **HTTP** 메시지 컨버터 위치**?** ArgumentResolver & ReturnValueHandler 에서 사용하는것