

Spring week 8

유채린

chaerin.du.ub@gmail.com



해당 내용은 인프런 강의를 정리한 내용입니다.
잘못된 내용에 대하여 피드백주시면 감사하겠습니다!

(인프런 아이콘 클릭 시, 참고한 인프런 강의 사이트로 이동합니다.)

Index

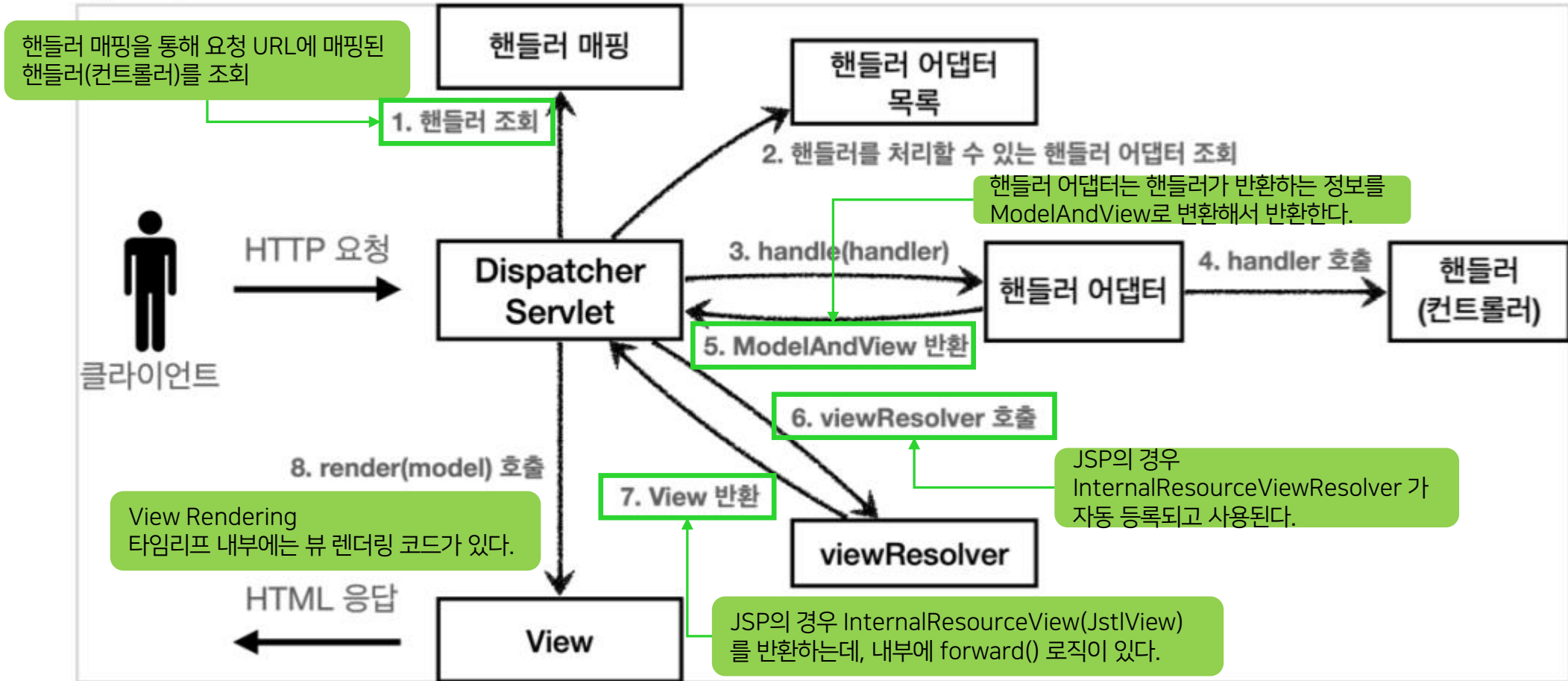
- DispatcherServlet, 핸들러 매핑과 핸들러 어댑터, 뷰 리졸버
- 스프링 MVC
 - HTTP 요청, HTTP 응답, HTTP 메시지 컨버터
- 로깅 Logging

DispatcherServlet

DispatchServlet

- DispatchServlet은 FrontController 패턴으로 구현되어 있다.
- 서블릿 하나로 클라이언트의 요청을 받는다. ➔ 공통 기능 처리
- '어댑터 패턴'을 사용해서 프론트 컨트롤러가 다양한 방식의 컨트롤러를 처리할 수 있다.
- 다형성과 어댑터 덕분에 기존 구조를 유지하면서, 프레임워크의 기능을 확장할 수 있다.
- DispatcherServlet 도 부모 클래스에서 HttpServlet 을 상속 받아서 사용하고, 서블릿으로 동작한다.
 - DispatcherServlet ➔ FrameworkServlet ➔ HttpServletBean ➔ HttpServlet
- 스프링 부트는 DispatcherServlet 을 서블릿으로 자동으로 등록하면서 모든 경로(urlPatterns="/")에 대해서 매핑한다.
 - 참고: 더 자세한 경로가 우선순위가 높다.
- FrameworkServlet.service() 를 시작으로 여러 메서드가 호출되면서 DispatcherServlet.doDispatch() 가 호출된다.

SpringMVC 구조



1. 핸들러 조회 → 2. 핸들러 어댑터 조회 → 3. 핸들러 어댑터 실행 → 4. 핸들러 실행
→ 5. ModelAndView 반환 → 6. viewResolver 실행 → 7. View 반환 → 8. 뷰 렌더링

주요 인터페이스 목록

- 핸들러 매핑
org.springframework.web.servlet.HandlerMapping
- 핸들러 어댑터
org.springframework.web.servlet.HandlerAdapter
- 뷰 리졸버
org.springframework.web.servlet.ViewResolver
- 뷰
org.springframework.web.servlet.View

핸들러 매핑과 핸들러 어댑터

HandlerMapping 핸들러 매핑

- 핸들러 매핑에서 이 컨트롤러를 찾을 수 있어야 한다.
- 예) 스프링 빈의 이름으로 핸들러를 찾을 수 있는 핸들러 매핑이 필요하다.

- 스프링 부트가 자동 등록하는 우선순위
순서대로 찾고 만약 없으면 다음 순서로 넘어간다.

0 = RequestMappingHandlerMapping (실무 99.9%)
: 애노테이션 기반의 컨트롤러인 @RequestMapping에서 사용

1 = BeanNameUrlHandlerMapping
: 스프링 빈의 이름으로 핸들러를 찾는다.
(url과 같은 이름을 찾는다.)

HandlerAdapter 핸들러 어댑터

- 핸들러 매핑을 통해서 찾은 핸들러를 실행할 수 있는 핸들러 어댑터가 필요하다.
- 예) Controller 인터페이스(@Controller 아님)를 실행할 수 있는 핸들러 어댑터를 찾고 실행해야 한다.

- 스프링 부트가 자동 등록하는 우선순위
순서대로 찾고 만약 없으면 다음 순서로 넘어간다.

0 = RequestMappingHandlerAdapter (실무 99.9%)
: 애노테이션 기반의 컨트롤러인 @RequestMapping에서 사용

1 = HttpRequestHandlerAdapter
: HttpRequestHandler 처리

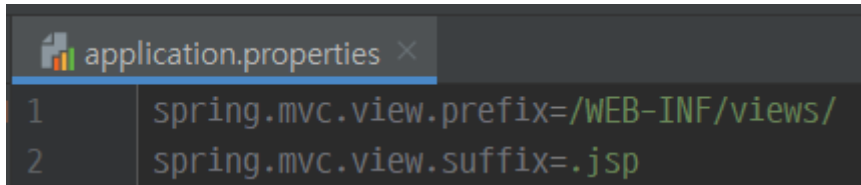
2 = SimpleControllerHandlerAdapter
: Controller 인터페이스(애노테이션X, 과거에 사용) 처리

스프링은 이미 필요한 핸들러 매핑과 핸들러 어댑터를 대부분 구현해 두었다.
개발자가 직접 핸들러 매핑과 핸들러 어댑터를 만드는 일은 거의 없다.

뷰 리졸버

뷰 리졸버

- JSP의 경우, InternalResourceViewResolver 사용
- 스프링 부트는 InternalResourceViewResolver 라는 뷰 리졸버를 자동으로 등록하는데,



```
application.properties
1 spring.mvc.view.prefix=/WEB-INF/views/
2 spring.mvc.view.suffix=.jsp
```

이때 application.properties 에 등록된
spring.mvc.view.prefix , spring.mvc.view.suffix 설정 정보를 사용해서 등록한다.

스프링 부트가 자동 등록하는 우선순위

순서대로 찾고 만약 없으면 다음 순서로 넘어간다.

1 = BeanNameViewResolver

: 빈 이름으로 뷰를 찾아서 반환한다. (예: 엑셀 파일 생성 기능에 사용)

2 = InternalResourceViewResolver

: JSP를 처리할 수 있는 뷰를 반환한다.

스프링 MVC

HTTP 요청, HTTP 응답

HTTP 요청

@RequestMapping ①

- 스프링은 애노테이션을 활용한 매우 유연하고, 실용적인 컨트롤러를 만들었는데 이것이 바로 @RequestMapping 애노테이션을 사용하는 컨트롤러이다.
 - @RequestMapping 기반의 스프링 MVC 컨트롤러

```
@Controller
public class SpringMemberFormControllerV1 {

    @RequestMapping("/springmvc/v1/members/new-form")
    public ModelAndView process() { return new ModelAndView( "new-form"); }

}
```

- @Controller : 스프링이 자동으로 스프링 빈으로 등록한다. (내부에 @Component 애노테이션이 있어서 컴포넌트스캔의 대상이 됨) 스프링 MVC에서 애노테이션 기반 컨트롤러로 인식한다.
- @RequestMapping : 요청 정보를 매핑한다. 해당 URL이 호출되면 이 메서드가 호출된다. 애노테이션을 기반으로 동작하기 때문에, 메서드의 이름은 임의로 지으면 된다.
- ModelAndView : 모델과 뷰 정보를 담아서 반환하면 된다.

@RequestMapping ②

- RequestMappingHandlerMapping 은 스프링 빈 중에서 @RequestMapping 또는 @Controller 가 클래스 레벨에 붙어 있는 경우에 매핑 정보로 인식한다.

```
@Controller => @Component @RequestMapping 으로 써도 됨
public class SpringMemberFormControllerV1 {

    @RequestMapping("/springmvc/v1/members/new-form")
    public ModelAndView process() { return new ModelAndView( viewName: "new-form"); }

}
```

- 컴포넌스 스캔 없이 다음과 같이 스프링 빈으로 직접 등록해도 동작한다.

```
//스프링 빈 직접 등록
@Bean
TestController testController() {
    return new TestController();
}
```

@RequestMapping ③

- 클래스 단위 뿐만 아니라 메서드 단위로도 적용할 수 있다.
따라서 컨트롤러 클래스를 유연하게 하나로 통합할 수 있다.

```
@Controller
@RequestMapping("/springmvc/v2/members")
public class SpringMemberControllerV2 {
    private MemberRepository memberRepository = MemberRepository.getInstance();

    @RequestMapping("/new-form") @RequestMapping("/springmvc/v2/members/new-form")
    public ModelAndView newForm() { return new ModelAndView( viewName: "new-form"); }

    @RequestMapping @RequestMapping("/springmvc/v2/members")
    public ModelAndView members() {
        List<Member> members = memberRepository.findAll();
        ModelAndView mv = new ModelAndView( viewName: "members");
        mv.addObject( attributeName: "members", members);

        return mv;
    }
}
```


@RequestMapping ④

- @RequestMapping("/hello-basic")
 - /hello-basic URL 호출이 오면 이 메서드가 실행되도록 매핑한다.
 - 대부분의 속성을 배열[] 로 제공하므로 다중 설정이 가능하다. {"/hello-basic", "/hello-go"}
→ @RequestMapping({"/hello-basic", "/hello-go"})
- 스프링은 다음 URL 요청들을 같은 요청으로 매핑한다.
 - localhost:8080/hello-basic
 - localhost:8080/hello-basic/
- HTTP Method
 - @RequestMapping 에 method 속성으로 HTTP 메서드를 지정하지 않으면 HTTP 메서드와 무관하게 호출된다.
 - 모두 허용 GET, HEAD, POST, PUT, PATCH, DELETE

HTTP 메서드 지정하는 방식) @RequestMapping(value = "/new-form", method = RequestMethod.GET)

@RequestParam

```
@Controller
@RequestMapping("/springmvc/v3/members")
public class SpringMemberControllerV3 {
    private MemberRepository memberRepository = MemberRepository.getInstance();

    @PostMapping("/save")
    public String save(
        @RequestParam("username") String username,
        @RequestParam("age") int age,
        Model model) {
        // 스프링은 HTTP 요청 파라미터를 @RequestParam으로 받을 수 있다.
        // @RequestParam("username") → request.getParameter("username")
        // GET 쿼리 파라미터, POST Form 방식 모두 지원

        // 스프링은 Model도 파라미터로 받을 수 있다.
        Member member = new Member(username, age);
        memberRepository.save(member);

        model.addAttribute(attributeName: "member", member);
        return "save-result";
    }
}
```

@GetMapping, @PostMapping

```
@Controller
@RequestMapping("/springmvc/v3/members")
public class SpringMemberControllerV3 {
    private MemberRepository memberRepository = MemberRepository.getInstance();

    // @RequestMapping(value = "/new-form", method = RequestMethod.GET)
    @GetMapping("/new-form")
    public String newForm() {
        return "new-form";
    }

    // @RequestMapping(method = RequestMethod.GET)
    @GetMapping
    public String members(Model model) {
        List<Member> members = memberRepository.findAll();

        model.addAttribute("attributeName: members", members);

        return "members";
    }

    // @RequestMapping(value = "/save", method = RequestMethod.POST)
    @PostMapping("/save")
    public String save(
        @RequestParam("username") String username
```

- @RequestMapping은 URL만 매칭하는 것이 아니라, HTTP Method도 함께 구분할 수 있다.
- @GetMapping, @PostMapping으로 더 편리하게 사용할 수 있다.
- 참고로 Get, Post, Put, Delete, Patch 모두 애노테이션이 준비되어 있다.
 - @GetMapping
 - @PostMapping
 - @PutMapping
 - @DeleteMapping
 - @PatchMapping
- 잘못된 HTTP Method로 호출하는 경우 405 에러 발생 (Method Not Allowed)

PathVariable (경로 변수)

- @RequestMapping은 URL 경로를 템플릿화 할 수 있는데, @PathVariable을 사용하면 매칭되는 부분을 편리하게 조회할 수 있다.
- @PathVariable의 이름과 파라미터 이름이 같으면 생략할 수 있다.

```
/**
 * PathVariable 사용
 * 변수명이 같으면 생략 가능
 * @PathVariable("userId") String userId -> @PathVariable userId
 */
@GetMapping("/mapping/{userId}")
public String mappingPath(@PathVariable("userId") String data) {
    log.info("mappingPath userId={}", data);
    return "ok";
}

/**
 * PathVariable 사용 다중
 */
@GetMapping("/mapping/users/{userId}/orders/{orderId}")
public String mappingPath(@PathVariable String userId, @PathVariable Long
    orderId) {
    log.info("mappingPath userId={}, orderId={}", userId, orderId);
    return "ok";
}
```

특정 파라미터 조건 매핑 / 특정 헤더 조건 매핑

```
/**
 * 파라미터로 추가 매핑
 * params="mode",
 * params="!mode"
 * params="mode=debug"
 * params="mode!=debug" (! = )
 * params = {"mode=debug", "data=good"}
 */
// 요즘은 PathVariable 사용한다.
@GetMapping(value = "/mapping-param", params = "mode=debug")
public String mappingParam() {
    log.info("mappingParam");
    return "ok";
}
```

- 특정 파라미터가 있거나 없는 조건을 추가할 수 있다.
잘 사용하지는 않는다.

```
/**
 * 특정 헤더로 추가 매핑
 * headers="mode",
 * headers="!mode"
 * headers="mode=debug"
 * headers="mode!=debug" (! = )
 */
@GetMapping(value = "/mapping-header", headers = "mode=debug")
public String mappingHeader() {
    log.info("mappingHeader");
    return "ok";
}
```

GET http://localhost:8080/mapping-header Postman으로 테스트

Params	Authorization	Headers (10)	Body	Pre-request Script	Tests	Settings
		<input checked="" type="checkbox"/> User-Agent ①	PostmanRuntime/7.28.4			
		<input checked="" type="checkbox"/> Accept ①	*/*			
		<input checked="" type="checkbox"/> Accept-Encoding ①	gzip, deflate, br			
		<input checked="" type="checkbox"/> Connection ①	keep-alive			
		<input checked="" type="checkbox"/> mode	debug			

미디어 타입 조건 매핑

HTTP 요청 Content-Type(consume), Accept(produce)

```
/**
 * Content-Type 헤더 기반 추가 매핑 Media Type
 * consumes="application/json"
 * consumes="!application/json"
 * consumes="application/*"
 * consumes="*/*"
 * MediaType.APPLICATION_JSON_VALUE
 */
@PostMapping(value = "/mapping-consume", consumes = "application/json")
public String mappingConsumes() {
    log.info("mappingConsumes");
    return "ok";
}
```

- HTTP 요청의 Content-Type 헤더를 기반으로 미디어 타입으로 매핑한다.
- 만약 맞지 않으면 HTTP 415 상태코드(Unsupported Media Type)을 반환한다.

```
/**
 * Accept 헤더 기반 Media Type
 * produces = "text/html"
 * produces = "!text/html"
 * produces = "text/*"
 * produces = "*/*"
 */
@PostMapping(value = "/mapping-produce", produces = "text/html")
public String mappingProduces() {
    log.info("mappingProduces");
    return "ok";
}
```

- HTTP 요청의 Accept 헤더를 기반으로 미디어 타입으로 매핑한다.
- 만약 맞지 않으면 HTTP 406 상태코드(Not Acceptable)을 반환한다.

다양한 파라미터를 지원하는 애노테이션 기반의 스프링 컨트롤러

```
@Slf4j // log
@RestController
public class RequestHeaderController {

    @RequestMapping("/headers")
    public String headers(HttpServletRequest request,
                        HttpServletResponse response,
                        HttpMethod httpMethod, HTTP Method: GET, POST, DELETE
                        Locale locale, 언어 정보
                        @RequestHeader MultiValueMap<String, String> headerMap, 헤더 정보
                        @RequestHeader("host") String host, 특정 헤더 정보
                        @CookieValue(value = "myCookie", required = false) String cookie
                        ) {
        특정 쿠키를 조회한다.
        Cookie 이름, required default: true
        log.info("request={}", request);
        log.info("response={}", response);
        log.info("httpMethod={}", httpMethod);
        log.info("locale={}", locale);
        log.info("headerMap={}", headerMap);
        log.info("header host={}", host);
        log.info("myCookie={}", cookie);

        return "ok";
    }
}
```

- 스프링 애노테이션 기반 컨트롤러는 인터페이스로 정형화되어 있지 않아 다양한 파라미터를 받을 수 있다.
→ 스프링이 지원하는 것 모두 가능
- @RequestHeader MultiValueMap<String, String> headerMap
 - 모든 HTTP 헤더를 MultiValueMap 형식으로 조회한다.

MultiValueMap

- MAP과 유사한데, 하나의 키에 여러 값을 받을 수 있다.
- HTTP header, HTTP 쿼리 파라미터와 같이 하나의 키에 여러 값을 받을 때 사용한다.
- keyA=value1&keyA=value2

HTTP 요청 데이터 전달 방법 3 가지

- 클라이언트에서 서버로 요청 데이터를 전달하는 세 가지 방법
- GET – 쿼리 파라미터
 - /url?username=hello&age=20
 - 메시지 바디 없이, URL의 쿼리 파라미터에 데이터를 포함해서 전달
 - 예) 검색, 필터, 페이징등에서 많이 사용하는 방식
- POST – HTML Form
 - content-type: application/x-www-form-urlencoded
 - 메시지 바디에 쿼리 파라미터 형식으로 전달 username=hello&age=20
 - 예) 회원 가입, 상품 주문, HTML Form 사용
- HTTP message body에 데이터를 직접 담아서 요청
 - HTTP API에서 주로 사용, JSON, XML, TEXT
 - 데이터 형식은 주로 JSON 사용
 - POST, PUT, PATCH



HTTP 요청 파라미터



HTTP 메시지 바디

HTTP 요청 파라미터

- 쿼리 파라미터, HTML Form

- GET 쿼리 파라미터 전송 방식이든, POST HTML Form 전송 방식이든 둘 다 형식이 같으므로 구분없이 조회할 수 있다.
- 이것을 간단히 **요청 파라미터(request parameter)** 조회라 한다.
- HttpServletRequest 의 request.getParameter() 를 사용
- GET - 쿼리 파라미터
 - http://localhost:8080/request-param?username=hello&age=20
- POST - HTML Form

```
POST /request-param ...  
content-type: application/x-www-form-urlencoded  
  
username=hello&age=20 메시지 바디
```

HTTP 요청 파라미터

@RequestParam ①

- @RequestParam : 파라미터 이름으로 바인딩
- @ResponseBody : View 조회를 무시하고, HTTP message body에 직접 해당 내용 입력
→ @RestController와 같은 역할
- @RequestParam의 name(value) 속성이 파라미터 이름으로 사용
 - @RequestParam("username") String memberName
→ request.getParameter("username")
- HTTP 파라미터 이름이 변수 이름과 같으면 @RequestParam(name="xx") 생략 가능

```
@ResponseBody View 조회를 무시하고, HTTP message body에 직접 해당 내용 입력
@RequestMapping("/request-param-v2") @RestController와 같은 역할
public String requestParamV2 (
    @RequestParam("username") String memberName,
    @RequestParam("age") int memberAge
) {
    // 파라미터 이름으로 바인딩
    log.info("username={}, age={}", memberName, memberAge);

    return "ok";
}
```

HTTP 요청 파라미터

@RequestParam ②

- String, int, Integer 등의 단순 타입이면 @RequestParam 도 생략 가능
- @RequestParam 애노테이션을 생략하면 스프링 MVC는 내부에서 required=false 를 적용한다.

```
@ResponseBody
@RequestMapping("/request-param-v4")    필수 아님
public String requestParamV4 (String username, int age) {
    log.info("username={}, age={}", username, age);

    return "ok";
}
```

HTTP 요청 파라미터

@RequestParam ③

- @RequestParam.required
 - 파라미터 필수 여부
 - 기본값이 파라미터 필수(true)이다.
- 파라미터 이름 없이 호출할 경우, 400 예외가 발생한다.
ex) localhost:8080/request-param-required?age=20
- 파라미터 이름만 있고 값이 없는 경우, 빈 문자로 통과
ex) localhost:808/request-param-required?username= → username=, age=null
- 기본형(primitive)에 null 입력할 경우, 500 에러 발생
→ null을 받을 수 있는 reference형 타입(ex.Integer)으로 변경하거나 defaultValue사용
- defaultValue 는 빈 문자의 경우에도 설정한 기본 값이 적용된다.
ex) localhost:8080/request-param-default?age=
→ username=guest, age=-1

```
@ResponseBody
@RequestMapping("/request-param-required")
public String requestParamRequired (
    @RequestParam(required = true) String username,
    @RequestParam(required = false) Integer age) {
    log.info("username={}, age={}", username, age);

    return "ok";
}
```

required = true는 default라서
font 색상이 회색으로 보여진다.

```
@ResponseBody
@RequestMapping("/request-param-default")
public String requestParamDefault (
    @RequestParam(required = true, defaultValue = "guest") String username,
    @RequestParam(required = false, defaultValue = "-1") int age) {
    log.info("username={}, age={}", username, age);

    return "ok";
}
```

이미 기본 값이 있어 required는
의미가 없다.

HTTP 요청 파라미터

@RequestParam ④

- 파라미터를 Map, MultiValueMap으로 조회할 수 있다.
- @RequestParam Map
Map(key=value)
- @RequestParam MultiValueMap
MultiValueMap(key=[value1, value2, ...] ex) (key=userIds, value=[id1, id2])
- 파라미터의 값이 1개가 확실하다면 Map을 사용, 그렇지 않다면 MultiValueMap 을 사용하자.
→ Map일 때, 하나의 파라미터에 2가지 value를 넘긴다면 첫 번째 값이 log에 출력된다.
ex) /request-param-map?username=user1&username=user2 → username=user1, age=null

```
@ResponseBody
@RequestMapping("/request-param-map")
public String requestParamMap (@RequestParam Map<String, Object> paramMap) {
    log.info("username={}, age={}", paramMap.get("username"), paramMap.get("age"));

    return "ok";
}
```

HTTP 요청 파라미터

@ModelAttribute

- 스프링MVC는 @ModelAttribute 가 있으면 다음을 실행한다.
 - 예시
 - HelloData 객체를 생성한다.
 - 요청 파라미터의 이름으로 HelloData 객체의 프로퍼티를 찾는다.
그리고 해당 프로퍼티의 setter를 호출해서 파라미터의 값을 입력(바인딩) 한다.
 - 예) 파라미터 이름이 username 이면 setUsername() 메서드를 찾아서 호출하면서 값을 입력한다.

```

@ResponseBody
@RequestMapping("/model-attribute-v1")
public String modelAttributeV1 (@ModelAttribute HelloData helloData) {
    log.info("username={}, age={}", helloData.getUsername(), helloData.getAge());
    log.info("helloData={}", helloData);
    return "ok";
}

@ResponseBody
@RequestMapping("/model-attribute-v2")
public String modelAttributeV2 (HelloData helloData) {
    log.info("username={}, age={}", helloData.getUsername(), helloData.getAge());
    log.info("helloData={}", helloData);
    return "ok";
}

```

스프링에서 파라미터 애노테이션 생략 시,
매핑 되는 규칙

- @RequestParam:**
String, int, Integer 같은 단순 타입
- @ModelAttribute:**
나머지 (argument resolver 로 지정해둔 타입 외)
ex) HelloData, HttpServletResponse

HTTP 요청 메시지

단순 텍스트 ① InputStream

- HTTP message body에 데이터를 직접 담아서 요청한다.
 - HTTP API에서 주로 사용한다. (JSON, XML, TEXT 형식으로 데이터를 보낸다.)
 - HTTP Method로는 POST, PUT, PATCH 사용
- @RequestParam, @ModelAttribute 사용 불가능
(HTML Form 형식으로 전달되는 경우 요청 파라미터로 인정)

- HTTP 메시지 바디의 데이터를 InputStream을 사용해서 직접 읽을 수 있다.

```
@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);
    // HTTP 요청 메시지 바디 내용 직접 조회

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

    response.getWriter().write(s: "ok");
}

@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);
    // HTTP 요청 메시지 바디 내용 직접 조회

    responseWriter.write(str: "ok");
    // HTTP 응답 메시지의 바디에 직접 결과 출력
}
```

HTTP 요청 메시지

단순 텍스트 ② HttpEntity, @RequestBody

- HttpEntity: HTTP header, body 정보를 편리하게 조회
 - 메시지 바디 정보를 직접 조회한다. (@RequestParam, @ModelAttribute 사용 안 함)
 - 응답에서도 HttpEntity 사용 가능 (view 조회안하고 메시지 바디 정보 직접 반환)
- HTTP 메시지 컨버터(HttpMessageConverter) 사용

```
@PostMapping("/request-body-string-v3") HttpEntity → ResponseEntity 사용 가능 (ResponseEntity는 HttpEntity 상속 받음)
public HttpEntity<String> requestBodyStringV3(HttpEntity<String> httpEntity) throws IOException {
    String messageBody = httpEntity.getBody();
    log.info("messageBody={}", messageBody);
    return new HttpEntity<>(body: "ok"); HttpEntity → ResponseEntity 사용 가능
} → return new ResponseEntity<String>("ok", HttpStatus.CREATED); (ResponseEntity는 HttpEntity 상속 받음)

@ResponseBody
@PostMapping("/request-body-string-v4")
public String requestBodyStringV4(@RequestBody String messageBody) throws IOException {
    log.info("messageBody={}", messageBody); HTTP 메시지 바디 정보 편리하게 조회 가능 (view 사용 안 함)
    return "ok"; 헤더 정보 필요한 경우 HttpEntity 또는 @RequestHeader 사용하기
}
```


HTTP 요청 메시지

JSON ① Jackson 라이브러리

- 문자로 된 JSON 데이터는 Jackson 라이브러리인 objectMapper를 사용해서 자바 객체로 변환

```
@PostMapping("/request-body-json-v1")
public void requestBodyJsonV1(HttpServletRequest request, HttpServletResponse response) throws IOException {
    ServletInputStream inputStream = request.getInputStream();    직접 HTTP 메시지 바디에서 데이터를 읽어온다.
    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(s: "ok");
}
```

```
@ResponseBody // return
@PostMapping("/request-body-json-v2")
public String requestBodyJsonV2(@RequestBody String messageBody) throws IOException {
    HTTP 메시지 바디에서 데이터를 꺼내고 messageBody에 저장한다.

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

    return "ok";
}
```

➔ 문자로 변환하고 다시 JSON 변환하기 불편함

HTTP 요청 메시지

JSON ② HttpEntity, @RequestBody

HTTP 요청시 content-type: application/json인지 확인 필수!!
(JSON을 처리할 수 있는 HTTP 메시지 컨버터 실행 가능한지 확인이 된다.)

- HttpEntity, @RequestBody 를 사용하면 HTTP 메시지 컨버터가 HTTP 메시지 바디의 내용을 우리가 원하는 문자나 객체 등으로 변환해준다.
- HTTP 메시지 컨버터는 문자 뿐만 아니라 JSON도 객체로 변환

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

```
@ResponseBody
@PostMapping("/request-body-json-v3")
public String requestBodyJsonV3(@RequestBody HelloData helloData) throws IOException {
    log.info("username={}, age={}", helloData.getUsername(), helloData.getAge());
    return "ok";
}
```

주의!!!

HelloData에 @RequestBody 를 생략하면
@ModelAttribute 가 적용되어버린다.

→ HTTP 메시지 바디가 아닌
요청 파라미터 처리함

HTTP 요청 메시지

JSON ③ @ResponseBody 이용한 직접 반환

- 응답의 경우에도 @ResponseBody 를 사용하면 해당 객체를 HTTP 메시지 바디에 직접 넣어줄 수 있다.
- 물론 이 경우에도 HttpEntity 를 사용해도 된다.

```
@ResponseBody
@PostMapping("/request-body-json-v5") HttpEntity도 가능
public HelloData requestBodyJsonV5(@RequestBody HelloData data) throws IOException {
    log.info("username={}, age={}", data.getUsername(), data.getAge());
    return data;
}
```

HTTP 응답

HTTP 응답 데이터 만드는 방법 3 가지

- 스프링(서버)에서 응답 데이터를 만드는 방법은 크게 3가지이다.
- 정적 리소스
 - 웹 브라우저에 정적인 HTML, css, js 제공할 때
- 뷰 템플릿 사용
 - 웹 브라우저에 동적인 HTML(ex. SSR) 제공할 때
- HTTP message 사용
 - HTML이 아니라 데이터를 전달한다.
 - HTTP 메시지 바디에 JSON 같은 형식으로 데이터를 실어 보낸다.

HTTP 응답 - 정적 리소스

- 스프링 부트에서 정적 리소스를 제공하는 디렉토리 위치
 - /static , /public , /resources , /META-INF/resources
- src/main/resources 는 리소스를 보관하는 곳, 클래스패스의 시작 경로
- 정적 리소스 경로: src/main/resources/static

예시)

src/main/resources/static/basic/hello-form.html에 파일이 있는 경우
웹 브라우저에서는
<http://localhost:8080/basic/hello-form.html>로 실행하면 된다.

HTTP 응답 - 뷰 템플릿 ①

- 뷰 템플릿 경로: src/main/resources/template

```
@RequestMapping("/response-view-v1")
public ModelAndView responseViewV1() {
    ModelAndView mav = new ModelAndView( viewName: "response/hello")
        .addObject( attributeName: "data", attributeValue: "hello!");
    return mav;
}
```

- String을 반환하는 경우 - view 또는 HTTP 메시지
 - @ResponseBody가 없으면 response/hello로 뷰를 찾고 렌더링 한다.
 - 실행: templates/response/hello.html
 - @ResponseBody가 있으면 뷰 리졸버 실행하지 않고, HTTP 메시지 바디에 직접 response/hello라는 문자가 입력된다.

```
@ResponseBody 있으면 안됨!
@RequestMapping("/response-view-v2")
public String responseViewV2(Model model) {
    model.addAttribute( attributeName: "data", attributeValue: "hello!");
    return "response/hello";
}
```

HTTP 응답 - 뷰 템플릿 ②

- Void를 반환하는 경우
 - @Controller를 사용하고 HttpServletResponse, OutputStream(Writer)같은 HTTP 메시지 바디를 처리하는 파라미터가 없으면 요청 URL을 참고해서 논리 뷰 이름으로 사용한다.
 - 요청 URL: /response/hello
 - 실행: templates/response/hello.html
 - 명시성 떨어져서 권장하지 않는다.

```
@RequestMapping("/response/hello")  
public void responseViewV3(Model model) {  
    model.addAttribute(attributeName: "data", attributeValue: "hello!");  
}
```


HTTP 응답

HTTP API, 메시지 바디에 직접 입력 ①

```
@Slf4j
@Controller
public class ResponseBodyController {

    @GetMapping("/response-body-string-v1")
    public void responseBodyV1(HttpServletResponse response) throws IOException {
        response.getWriter().write("ok");
    }

    @GetMapping("/response-body-string-v2")
    public ResponseEntity<String> responseBodyV2() { return new ResponseEntity<>("ok", HttpStatus.OK); }

    @ResponseBody
    @GetMapping("/response-body-string-v3")
    public String responseBodyV3() { return "ok"; }
```

@RestController 사용하게 되면 모두 **@ResponseBody**가 적용되는 효과
@Controller (뷰 템플릿 사용이 아닌 http 메시지 바디에 직접 데이터 입력) → **@RestController**는 REST API(HTTP API)를 만들 때 사용하자.

HttpServletResponse response

HTTP 메시지의 헤더, 바디 정보를 갖고 있을 뿐만 아니라 HTTP 응답 코드 설정 가능

@ResponseBody View 사용하지 않고 HTTP 메시지 컨버터를 통한 HTTP 메시지 직접 입력

HTTP 응답

HTTP API, 메시지 바디에 직접 입력 ②

```
@GetMapping("/response-body-json-v1")
public ResponseEntity<HelloData> responseBodyJsonV1() {
    HelloData helloData = new HelloData();
    helloData.setUsername("userA");
    helloData.setAge(20);

    return new ResponseEntity<>(helloData, HttpStatus.OK);
}
```

HTTP 메시지 컨버터를 통해 JSON 형식으로 변환되어 반환

```
@ResponseStatus(HttpStatus.OK)
@ResponseBody
@GetMapping("/response-body-json-v2")
public HelloData responseBodyJsonV2() {
    HelloData helloData = new HelloData();
    helloData.setUsername("userA");
    helloData.setAge(20);

    return helloData;
}
```

@ResponseBody 사용하면서 HTTP 응답코드 설정하기
하지만 응답코드 동적 변경은 불가능하다 → ResponseEntity 사용한다.

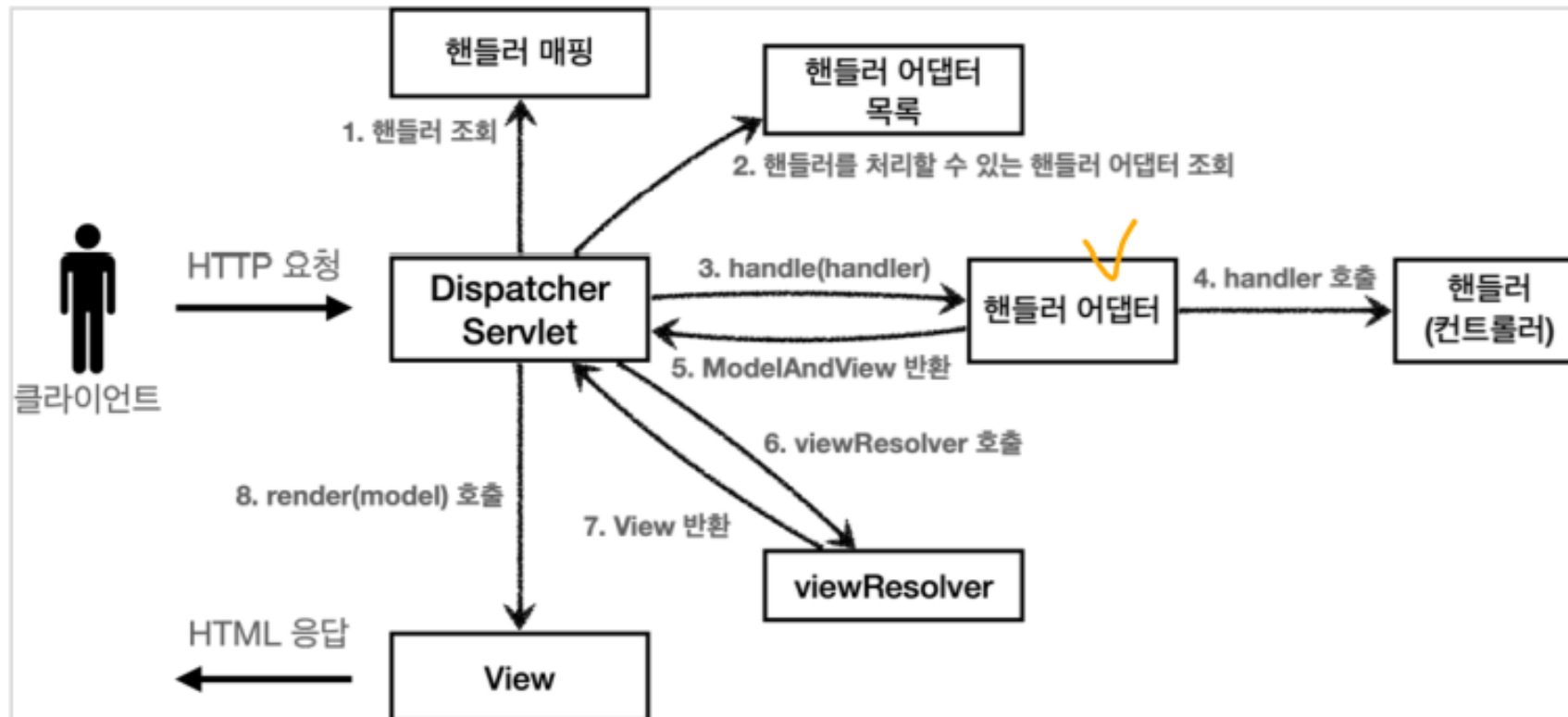
HTTP 메시지 컨버터

HTTP 메시지 컨버터

- HTTP 메시지 컨버터는 HTTP 요청, HTTP 응답 둘 다 사용된다.
- 스프링 MVC는 다음의 경우에 HTTP 메시지 컨버터를 적용한다.
- HTTP 요청
 - @RequestBody
 - HttpEntity(RequestEntity)
- HTTP 응답
 - @ResponseBody
 - HttpEntity(ResponseEntity)

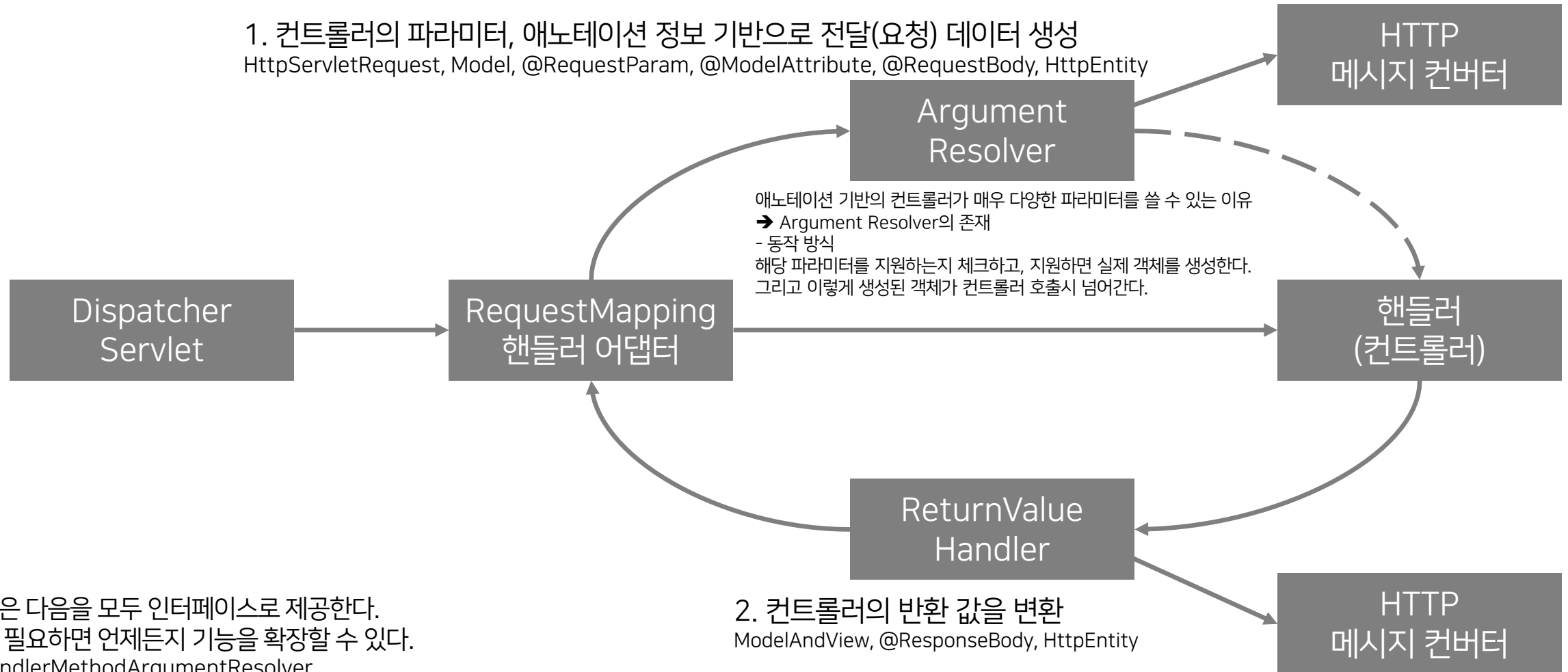
HTTP 메시지 컨버터

- HTTP 메시지 컨버터는 @RequestMapping을 처리하는 핸들러 어댑터인 RequestMappingHandlerAdapter에 있다.



HTTP 메시지 컨버터

1. 컨트롤러의 파라미터, 애노테이션 정보 기반으로 전달(요청) 데이터 생성
HttpServletRequest, Model, @RequestParam, @ModelAttribute, @RequestBody, HttpEntity



스프링은 다음을 모두 인터페이스로 제공한다.
따라서 필요하면 언제든지 기능을 확장할 수 있다.

- `HandlerMethodArgumentResolver`
- `HandlerMethodReturnValueHandler`
- `HttpMessageConverter`

(거의 없다.)

2. 컨트롤러의 반환 값을 변환
`ModelAndView`, `@ResponseBody`, `HttpEntity`

로깅 Logging

로깅 라이브러리

- 스프링 부트 라이브러리를 사용하면 스프링 부트 로깅 라이브러리(spring-boot-starter-logging)가 함께 포함된다.
- 스프링 부트 로깅 라이브러리는 기본으로 SLF4J, Logback 로깅 라이브러리를 사용한다.
 - SLFJ4 (인터페이스), Logback(SLFJ4 구현체 중 하나)
- 로그 선언 방식
 - `private Logger log = LoggerFactory.getLogger(getClass());`
 - `private static final Logger log = LoggerFactory.getLogger(Xxx.class);`
 - `@Slf4j` : 롬복 사용 가능
- 로그 호출
 - `log.info("hello")` → 실무에서는 항상 로그를 사용해서 출력하자
 - `System.out.println("hello")`

```
@RestController
public class LogTestController {

    private final Logger log = LoggerFactory.getLogger(getClass());

    @RequestMapping("/log-test")
    public String logTest() {
        String name = "Spring";

        System.out.println("name = "+name);

        log.trace("trace log = {}", name);
        log.debug("debug log = {}", name);
        log.info(" info log = {}", name);
        log.warn(" warn log = {}", name);
        log.error("error log = {}", name);

        return "ok";
    }
}
```



```

@RestController 반환으로 HTTP 메시지 바디에 바로 입력
public class LogTestController {

    private final Logger log = LoggerFactory.getLogger(getClass());

    @RequestMapping("/log-test")
    public String logTest() {
        String name = "Spring";

        System.out.println("name = "+name);

        log.trace("trace log = {}", name);
        log.debug("debug log = {}", name);
        log.info(" info log = {}", name);
        log.warn(" warn log = {}", name);
        log.error("error log = {}", name);

        return "ok";
    }
}

```

- @Controller는 반환 값이 String이면 뷰 이름으로 인식되어 뷰를 찾고 뷰가 rendering 된다.

로그 레벨

- LEVEL: TRACE > DEBUG > INFO > WARN > ERROR
- 개발 서버는 debug 출력
- 운영 서버는 info 출력

로그 레벨 설정

```

application.properties
1 #전체 로그 레벨 설정(기본 info)
2 logging.level.root=info
3
4 #hello.springmvc 패키지과 그 하위 로그 레벨 설정
5 logging.level.hello.springmvc=debug
6

```

올바른 로그 사용법

- log.debug("data="+data)
 - 로그 출력 레벨보다 높더라도 해당 코드가 실행된다. (더하기 연산)
 - 메모리 사용, CPU 사용 → 쓸 데 없이 리소스 소비
- log.debug("data={}", data)
 - 로그 출력 레벨을 info로 설정하면 아무 일도 발생하지 않는다.
 - 따라서 앞과 같은 의미 없는 연산이 발생하지 않는다.

→ log.debug("data="+data) 보다는
log.debug("data={}", data) 으로 쓰기

로그 사용 장점

- 쓰레드 정보, 클래스 이름 같은 부가 정보를 함께 볼 수 있고, 출력 모양을 조정할 수 있다.
- 로그 레벨에 따라 개발 서버에서는 모든 로그를 출력하고, 운영서버에서는 출력하지 않는 등 로그를 상황에 맞게 조절할 수 있다.
- 시스템 아웃 콘솔에만 출력하는 것이 아니라, 파일이나 네트워크 등, 로그를 별도의 위치에 남길 수 있다.
- 특히 파일로 남길 때는 일별, 특정 용량에 따라 로그를 분할하는 것도 가능하다.
- 수십배 차이 난다.
- 성능도 일반 System.out보다 좋다. (내부 버퍼링, 멀티 쓰레드 등등)
그래서 실무에서는 꼭 로그를 사용해야 한다.

그 외

롬복 Lombok

- 롬복 @Data
 - @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor 를 자동으로 적용해준다.
 - 간결한 코드

```
import lombok.Data;

@Data // Lombok
public class HelloData {
    private String username;
    private int age;
}
```

감사합니다.