6️⃣

Section6. 스프링 MVC - 기본 기능

프로젝트 생성

💡 Project: Gradle
Language: Java
Spring Boot: 3.2.5

Project Metadata
Group: hello
Artifact: springmvc
Name: springmvc
Package name: hello.springmvc
Packaging: Jar (항상 내장 서버(톰캣 등)을 사용하고, 'webapp' 경로 사용 X)
Java: 17

Dependencies

Spring Web
Thymeleaf
Lombok

▼ 로깅 간단히 알아보기

로깅 라이브러리

스프링 부트 라이브러리 사용하면 스프링 부트 로깅 라이브러리가 함께 포함된다.

- `SLF4J` - http://www.slf4j.org
  - 인터페이스, 그 구현체로 Logback 같은 라이브러리를 선택하면 된다.
- `logback` - http://logback.qos.ch
  - 실무에서 대부분 사용한다.

로그 선언

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

로그 호출

```
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);
```

매핑 정보

@RestController

- @Controller 는 반환 값이 String 이면 뷰 이름으로 인식된다.
- @RestController 는 반환 값으로 뷰를 찾는 것이 아니라, **HTTP 메시지 바디에 바로 입력**한다.

테스트

- 로그가 출력되는 포맷
  - 시간, 로그, 레벨, 프로세스 ID, 쓰레드 명, 클래스명, 로그 메시지
- 로그 레벨
  - TRACE > DEBUG > INFO > WARN > ERROR
  - 개발 서버는 DEBUG , 운영 서버는 INFO
- `@Slf4j`로 변경

로그 사용법

`log.debug("data="+data)`

- 로그 출력 레벨을 info 로 설정해도 문자열의 '+' 연산이 이루어져 불필요한 리소스 사용이 발생

`log.debug("data={}", data)`

- `data` 를 param 으로 넘기기 때문에, 위와 같은 불필요한 연산이 이루어지지 않음

요청 매핑

@RequestMapping(”/hello-basic”)

- /hello-basic URL 호출이 오면 이 메서드가 실행되도록 매핑한다.
- 대부분의 속성을 `배열[]` 로 제공하므로 다중 설정이 가능하다.
  - {”hello-basic”, “/hello-go”}

HTTP 메서드

`@RequestMapping` : method 를 지정하지 않으면 GET, HEAD, POST, PUT, PATCH, DELETE 모두 허용

```
@RequestMapping(value = "/mapping-get-v1", method = RequestMethc
```

HTTP 메서드 축약

```
@GetMapping(value = "/mapping-get-v2")
```

- 코드 내부를 보면 @RequestMapping 과 method를 지정해서 사용하는 것을 확인할 수 있다.
- 더욱 직관적

PathVariable(경로 변수) 사용

```
@GetMapping("/mapping/{userId}")
public String mappingPath(@PathVariable("userId") String data)
```

- 최근 HTTP API는 리소스 경로에 식별자를 넣는 스타일을 선호한다.
  - /mapping/userA
  - /users/1
- `@RequestMapping` 은 URL 경로를 템플릿화 할 수 있는데, `@PathVariable` 을 사용하면 매칭되는 부분을 편리하게 조회할 수 있다.
- `@PathVariable` 의 이름과 파라미터 이름이 같으면 생략할 수 있다.

💡 **@PathVariable VS [@RequestMapping|@RequestParam]**

- **Path Variable**은 구체적인 리소스를 식별하는데 사용
  - ex) id가 444인 게시글이라는 구체적인 리소스를 식별하는데 사용
    - /board/444
- **Query String**은 리소스들을 정렬, 필터링 혹은 페이징하는 곳에 사용
  - ex) writer가 nick인 게시글 리스트라는 필터링된 리소스들을 가져오는데 사용
    - /board/list?writer=nick

특정 파라미터 조건 매핑

```
@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 헤더 조건 매핑

```
@GetMapping(value = "/mapping-header", headers = "mode=debug")
```

- 파라미터 매핑과 비슷하지만, HTTP 헤더를 사용한다.

미디어 타입 조건 매핑 - HTTP 요청 Content-Type, consume

```
@PostMapping(value = "/mapping-consume", consumes = "applicatior
public String mappingContent(){
    log.info("mappingConsumes");
    return "ok";
}
```

- `application/json` 형태의 요청만을 허용

미디어 타입 조건 매핑 - HTTP 요청 Accept, produce

```
@PostMapping(value = "/mapping-produce", produces = "text/html")
public String mappingProduce(){
    log.info("mappingProduce");
    return "ok";
}
```

- 요청 Header - [Accept] 가 일치해야 한다. (불일치 시 406 에러 발생)

요청 매핑 - API 예시

회원 관리를 HTTP API로 만든다 생각하고 매핑을 어떻게 하는 지알아보자.

회원 관리 API

- 회원 목록 조회: `GET` /users
- 회원 등록: `POST` /users
- 회원 조회: `GET` /users/{userId}
- 회원 수정: `PATCH` /users/{userId}
- 회원 삭제: `DELETE` /users/{userId}

> 💡 직접 구현

▼ **회원 관리 Controller 코드**

```
@RequestMapping("/users")
@RestController
public class UserMappingController {
    ArrayList<User> userList = new ArrayList<>();

    @GetMapping
    public List<User> list(){
        userList.add(new User(1L,"이현수", "test1@test.com", 2
        userList.add(new User(2L,"차봉석", "test2@test.com", 2
        return userList;
    }
```

```
    @GetMapping("/{userId}")
    public User findUser(@PathVariable(name = "userId") Long
        for (User user : userList) if(Objects.equals(user.get
        return null;
    }

    @PostMapping("/")
    public String register(
            @RequestParam(name = "id") Long id,
            @RequestParam(name = "name") String name,
            @RequestParam(name = "email") String email,
            @RequestParam(name = "age") int age){
        User user = new User(id,name,email,age);
        userList.add(user);

        return "ok";
    }

    @PatchMapping("{userId}")
    public String update(
            @PathVariable(name = "userId") Long userId,
            @RequestParam(name = "name") String name,
            @RequestParam(name = "email") String email
    ){
        User user = null;
        for (User u : userList) if(Objects.equals(u.getId(),

        if(user != null){
            user.setName(name);
            user.setEmail(email);
            return "ok";
        } else {
            return "fail";
        }
    }

    @DeleteMapping("{userId}")
    public String delete(@PathVariable(name = "userId") Long
        for (User user : userList) {
            if(Objects.equals(userId, user.getId())) {
```

```
                userList.remove(user);
                return "ok";
            }
        }
        return "fail";
    }
}
```

HTTP 요청 - 기본, 헤더 조회

```
@Slf4j
@RestController
public class RequestHeaderController {
    @RequestMapping("/headers")
    public void headers(HttpServletRequest request,
                        HttpServletResponse response,
                        HttpMethod httpMethod,
                        Locale locale,
                        @RequestHeader MultiValueMap<String, S
                        @RequestHeader("host") String host,
                        @CookieValue(value = "myCookie", requi
        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);
    }
}
```

- **MultiValueMap**
  - 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 파라미터 - 쿼리 파라미터, HTML FORM

💡 실습 코드 작성

```
@Slf4j
@Controller
public class RequestParamController {
    @RequestMapping("/request-param-v1")
    public void requestParam(HttpServletRequest request, HttpSer
        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));
        log.info("username = {}, age = {}", username, age);

        response.getWriter().write("ok");
    }
}
```

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<form action="/request-param-v1" method="post">
    username: <input type="text" name="username" />
    age: <input type="text" name="age" />
    <button type="submit">전송</button>
</form>
</body>
</html>
```

- 기존 `HttpServletRequest` 객체의 `getParameter()` 를 활용하여 데이터 조회

HTTP 요청 파라미터 - @RequestParam

스프링이 제공하는 `@ReqeustParam` 을 사용하면 요청 파라미터를 편리하게 사용할 수 있다.

💡 실습 코드 작성

- `@RequestParam` : 파라미터 이름으로 바인딩
- `@ResponseBody` : View 조회를 무시하고, HTTP message body에 직접 해당 내용 입력

@RequestParam의 name(value) 속성이 파라미터 이름으로 사용

- `@RequestParam("username") String memberName`
  - `request.getParameter("username")`
- HTTP 파라미터 이름이 변수 이름과 같으면 `name` 속성 생략 가능

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

  - 추가적으로 `String`, `int`, `Integer` 등의 단순 타입이면 어노테이션 자체 생략 가능

💡 너무 없는 것도 약간 과하다는 생각도 있으면 붙이는 것도 좋다.

`@RequestParam`이 있으면 명확하게 요청 파리미터에서 데이터를 읽는 것을 알 수 있다.

```
@RequestParam(required = [true|false])
```

- 파라미터 필수 여부, default: true
- ※ 주의! ※ - 기본형(primitive)에 null 입력
  - `null` 을 `int` 에 입력하는 것은 불가능(500 예외 발생)
  - `null` 을 받을 수 있는 `Integer` 로 변경하거나, 또는 다음에 나오는 `defaultValue` 사용

@Defaultvalue 속성

```
@ResponseBody
@RequestMapping("/request-param-default")
```

```
    public String requestParamDefault(
            @RequestParam(defaultValue = "guest") String usernam
            @RequestParam(defaultValue = "-1") int age) {
        log.info("username = {}, age = {}", username, age);
        return "ok";
    }
```

- `required` 와는 상관없이 파라미터 지정이 안되어도 `defaultValue` 를 사용할 수 있다.
- 파라미터의 값이 빈 문자("`?username=&age=`") 일 경우에도 `defaultValue` 가 적용된다.

파라미터를 Map 으로 조회하기 - requestParamMap

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

- `MultiValueMap` 도 사용가능하다.

```
@ResponseBody
@RequestMapping("/request-param-multi-map")
public String requestParamMap(@RequestParam MultiValueMap<Str
    log.info("username={}, age={}", paramMap.get("username"),
    return "ok";
}
```

```
username = [spring], age = [1, 10]
```

  - 파라미터의 값이 1개가 확실하다면 `Map` 을 사용해도 되지만, 그렇지 않다면 MVM을 사용하자.

HTTP 요청 파라미터 - @ModelAttribute

요청 파라미터를 받아서 필요한 객체를 만들고 그 객체에 값을 넣어주어야 한다.
보통은 다음과 같은 코드 사용

```
@RequestParam String username;
@RequestParam int age;

HelloData data = new HelloData();
data.setUsername(username);
data.setAge(age);
```

스프링은 이 과정을 자동화해주는 '`@ModelAttribute`' 기능을 제공한다.

💡 실습 코드 작성

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

스프링MVC는 `@ModelAttribute` 가 있으면 다음을 실행한다.

- "HelloData" 객체를 생성한다.
- 요청 파라미터의 이름으로 "HelloData" 객체의 프로퍼티를 찾는다. 그리고 해당 프로퍼티의 setter를 호출해서 파라미터의 값을 바인딩 한다.
    - 예) 파라미터 이름이 "`username`" 이면 `setUsername()` 메서드를 호출하면서 값을 입력한다.

프로퍼티

객체의 `getUsername()`, `setUsername()` 메서드가 있으면, 이 객체는 `username` 이라는 프로퍼티를 가지고 있다.

`username` 프로퍼티의 값을 변경하면 `setUsername()` 이 호출되고, 조회하면 `getUsername()` 이 호출된다.

바인딩 오류

`age=abc` 처럼 숫자가 들어가야 할 곳에 문자를 넣으면 `BindException` 오류가 발생한다.

```
@ResponseBody
    @RequestMapping("/model-attribute-v2")
```

```
    public String modelAttributeV2(HelloData helloData){
        log.info("username = {}, age = {}", helloData.getUsernam
        return "ok";
}
```

- `@ModelAttribute`는 생략할 수 있다.

스프링은 해당 생략 시 다음과 같은 규칙을 적용한다.

- `String`, `int`, `Integer` 같은 단순 타입 = `@RequestParam`
- 나머지 - `@ModelAttribute`(argument resolver 로 지정해둔 타입 외)

HTTP 요청 메시지 - 단순 텍스트

```
// 기존 메시지 바디의 텍스트 출력 코드
// 바이트 코드 변환을 위한 stream 생성 - 출력 ....

@PostMapping("/request-body-string-v1")
public void requestBodyStringV1(HttpServletRequest request, Http
    ServletInputStream inputStream = request.getInputStream();
    String messageBody = StreamUtils.copyToString(inputStream, S

    log.info("messageBody = {}", messageBody);
    response.getWriter().write("OK");
}
```

스프링 MVC는 다음 파라미터를 지원한다.

```
@PostMapping("/request-body-string-v3")
public HttpEntity<String> requestBodyStringV3(HttpEntity<String>
    String messageBody = httpEntity.getBody();
    log.info("messageBody = {}", messageBody);

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

- **HttpEntity**: HTTP header, body 정보를 편리하게 조회
  - 메시지 바디 정보를 직접 조회
  - 요청 파라미터를 조회하는 기능과 관계 없음

- HttpEntity 는 응답에도 사용 가능
  - 메시지 바디 정보 직접 반환
  - 헤더 정보 포함 가능
  - view 조회 X

`HttpEntity` 를 상속받은 다음 객체들도 같은 기능을 제공한다.

- **RequestEntity**
  - HttpMethod, url 정보가 추가 ← 요청에서 사용
- **ResponseEntity**
  - HTTP 상태 코드 설정 가능, 응답에서 사용
  - `return new ResponseEntity<>("ok", HttpStatus.CREATED);`

> 💡 스프링MVC 내부에서 HTTP 메시지 바디를 읽어서 문자나 객체로 변환해서 전달해 주는데, 이때 HTTP 메시지 컨버터라는 기능을 사용한다.

@RequestBody

HTTP 메시지 바디 정보를 편리하게 조회할 수 있다.
헤더 정보가 필요하다면
`HttpEntity` 를 사용하거나 '@RequestHeader'를 사용하면 된다.

@ResponseBody

응답 결과를 HTTP 메시지 바디에 직접 담아서 전달할 수 있다.
물론 이 경우에도 view를 사용하지 않는다.

> 💡 **요청 파라미터 vs HTTP 메시지 바디**
>
> - 요청 파라미터를 조회하는 기능: '`@RequestParam`', '`@ModelAttribute`'
> - HTTP 메시지 바디를 직접 조회하는 기능 '`@RequestBody`'

HTTP 요청 메시지 - JSON

- 기존에는 `HttpServletRequest` 를 사용해서 직접 HTTP 메시지 바디에서 데이터를 읽어와서, 문자로 변환했다.

- 문자로 된 JSON 데이터를 Jackson 라이브러리인 `objectMapper` 를 사용해서 자바 객체로 변환한다.

```
/**
 * @RequestBody 생략 불가능(@ModelAttribute 가 적용되어 버림)
 * HttpMessageConverter 사용 -> MappingJackson2HttpMessageConvert
 *
 */
@ResponseBody
@PostMapping("/request-body-json-v3")
public String requestBodyJsonV3(@RequestBody HelloData helloData
    log.info("username = {}, age = {} ", helloData.getUsername()
    return "ok";
}
```

@RequestBody 객체 파라미터

'`HttpEntity`', '`@RequestBody`' 를 사용하면 **HTTP 메시지 컨버터**가 HTTP 메시지 바디의 내용을 우리가 원하는 문자나 객체 등으로 변환해준다.

HTTP 메시지 컨버터는 문자 뿐만 아니라 JSON도 객체로 변환해준다.

@RequestBody 는 생략 불가능

HelloData에 '@ReqeustBody'를 생략하면 '@ModelAttribute' 가 적용된다.

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

- **HelloData** 반환
  - `@RequestBody` 에서 JSON 요청 → HTTP 메시지 컨버터 → 객체
  - `@ResponseBody` 에서 객체 → HTTP 메시지 컨버터 → JSON 응답

HTTP 응답 - 정적 리소스, 뷰 템플릿

스프링에서 응답 데이터를 만드는 방법은 크게 3가지이다.

1. **정적 리소스**

- HTML, CSS, JS

2. **뷰 템플릿 사용**
   - 동적 HTML
3. **HTTP 메시지 사용**
   - 데이터 전달

정적 리소스

클래스패스의 다음 디렉토리에 있는 정적 리소스 제공

`/static`, `/public`, `/resources`, `/META-INF/resources`

`/src/main/resources` 는 리소스를 보관하는 곳이고, **클래스패스의 시작 경로**이다.

뷰 템플릿

스프링 부트는 기본 뷰 템플릿 경로를 제공한다.

`/src/main/resources/templates`

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

// Model 사용
@RequestMapping("/response-view-v2")
public String responseViewV2(Model model){
    model.addAttribute("data", "hello");
    return "response/hello";
}
```

String을 반환하는 경우

- `@ResponseBody` 가 있다면, 뷰 리졸버 대신 converter 가 호출되기 때문에 문자 그대로 반환된다.
- `@ResponseBody` 가 없다면, 뷰 리졸버가 호출되기 때문에 반환되는 경로의 template 리소스가 반환된다.

HTTP 메시지 사용

`@ResponseBody`, `HttpEntity` 를 사용하면, 뷰 템플릿이 아닌 HTTP 메시지 바디에 직접 응답 데이터를 출력할 수 있다.

HTTP 응답 - HTTP API, 메시지 바디에 직접 입력

HTTP 메시지 바디에 JSON 같은 형식으로 데이터를 실어 보낸다.

> 💡 실습 코드 작성

```
@ResponseBody
@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);
}

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

    return helloData;
}
```

response-body-json-v1

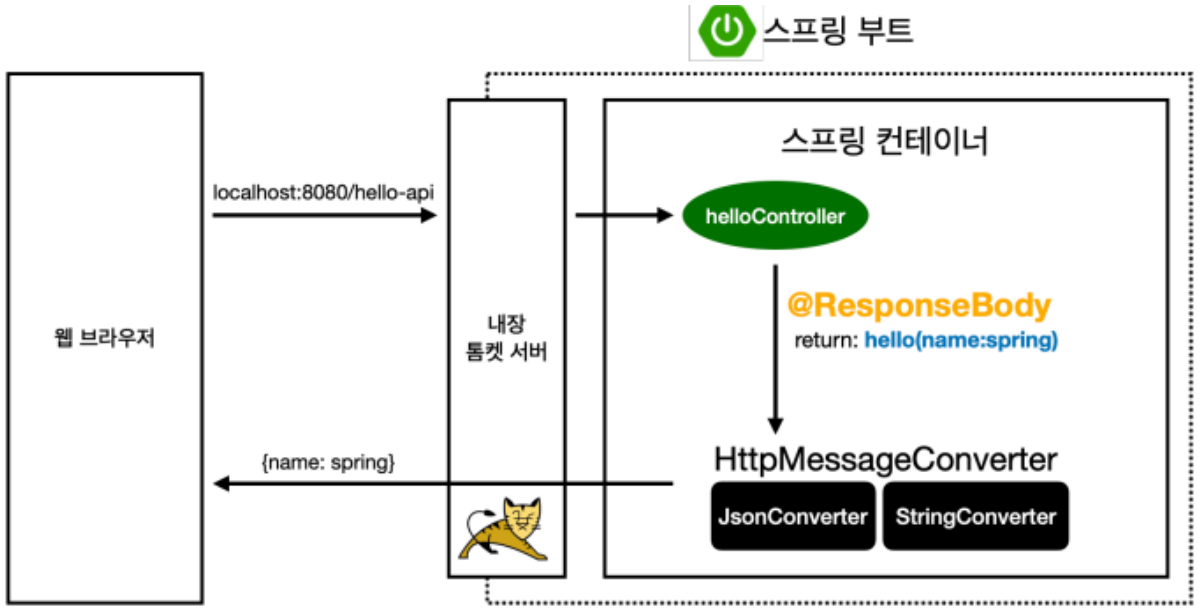
- `ResponseEntity<Object>` 리턴
  - HTTP 상태를 정의해서 return 할 수 있어서 유용하다.

response-body-json-v2

- 형식을 DTO로 설정한 뒤 DTO를 반환

HTTP 메시지 컨버터

HTTP 메시지 바디에서 직접 읽거나 쓰는 경우 HTTP **메시지 컨버터를 사용**하면 편리하다.



> 응답의 경우 클라이언트의 HTTP Accept 헤더와 서버의 컨트롤러 반환 타입 정보 둘을 조합해서 Converter 가 선택된다. (실제로는 더 많은 고려사항이 있다.)

스프링 MVC는 다음의 경우에 HTTP 메시지 컨버터를 적용한다.

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

HTTP 메시지 컨버터 인터페이스

```
boolean canRead(Class<?> clazz, @Nullable MediaType mediaType);
boolean canWrite(Class<?> clazz, @Nullable MediaType mediaType);

T read(Class<? extends T> clazz, HttpInputMessage inputMessage)
void write(T t, @Nullable MediaType contentType, HttpOutputMessa
```

HTTP 메시지 컨버터는 HTTP 요청, 응답 둘 다 사용된다.

- `canRead()`, `canWrite()` : 메시지 컨버터가 해당 클래스, 미디어타입 지원하는 지 체크
- `read()`, `write()`: 메시지 컨버터를 통해 메시지를 읽고 쓴다.

```
0 = ByteArrayHttpMessageConverter
1 = StringHttpMessageConverter
2 = MappingJackson2HttpMessageConverter
```

StringHttpMessageConverter

- 클래스 타입: `String` , 미디어타입: `*/*`
- 요청 예) `@RequestBody String data`
- 응답 예) `@ResponseBody return "ok"` 쓰기 미디어타입 text/plain

MappingJackson2HttpMessageConverter

- 클래스 타입: 객체 또는 `HashMap`, 미디어타입 `application/json` 관련
- 요청 예) `@RequestBody HelloData data`
- 응답 예) `@ResponseBody return helloData` 쓰기 미디어타입 application/json 관련

HTTP 요청 데이터 읽기

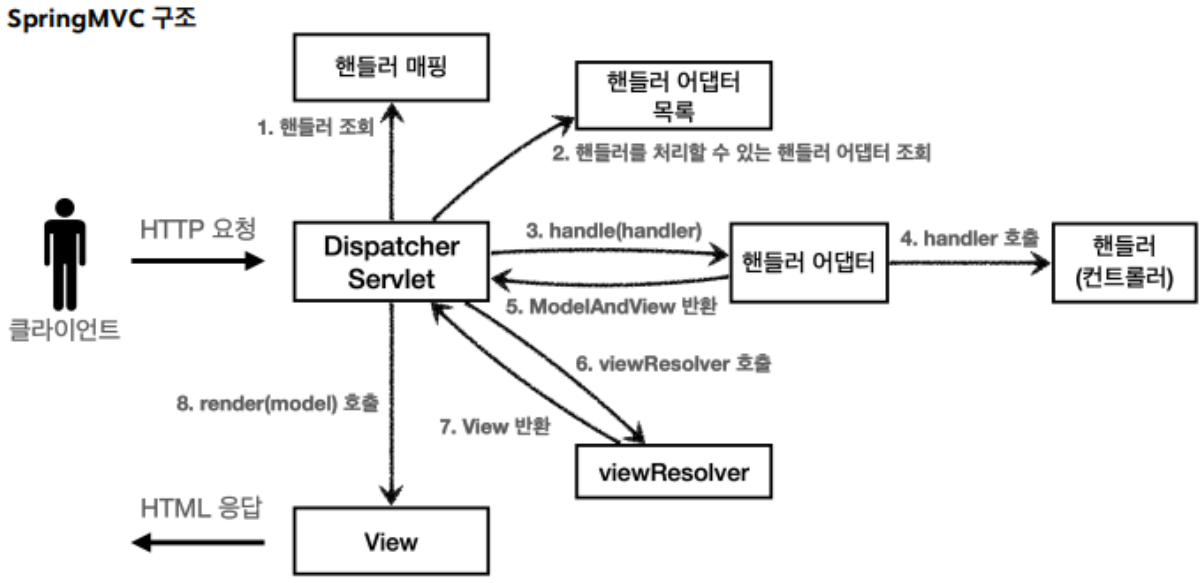
- CanRead()
  - 대상 클래스 타입을 지원하는가
  - HTTP 요청의 Content-Type 미디어 타입을 지원하는가

    → read()

HTTP 응답 데이터 쓰기

- CanWrite()
  - 대상 클래스 타입을 지원하는가
  - HTTP 요청의 Accept 미디어 타입을 지원하는가? (`@RequestMapping` 의 `produces`)
    → canWrite()

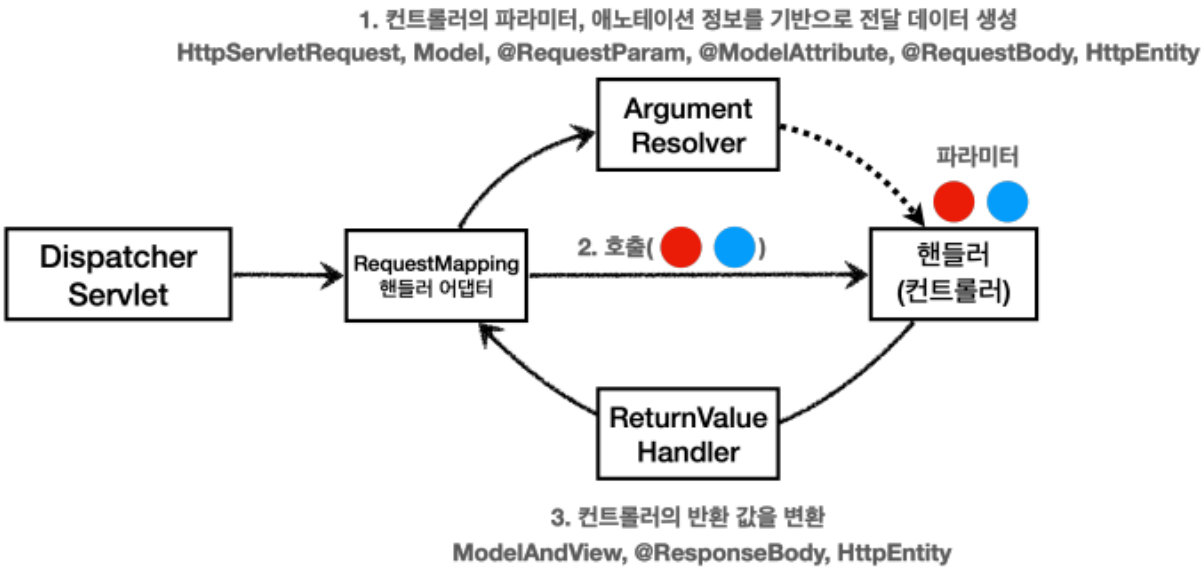
요청 매핑 핸들러 어댑터 구조



메시지 컨버터는 어디에서 호출되는 지

- `@RequestMapping` 을 처리하는 핸들러 어댑터인 `ReqeustMappingHandlerAdapter`

ReqeustMappingHandlerAdapter 동작 방식 ← 파라미터 처리



💡 **※ ArgumentResolver ※**

다양한 파라미터를 처리하는 인터페이스

1.

`boolean supportsParameter(MethodParameter parameter);` 처리 가능 파라미터인지 확인

2. 처리 가능 파라미터인지 확인되면

`resolveArgument()` 호출하여 실제 객체 반환

```
@Nullable
Object resolveArgument(MethodParameter parameter, @Nullabl
```

💡 **※ ReturnValueHandler ※**

-

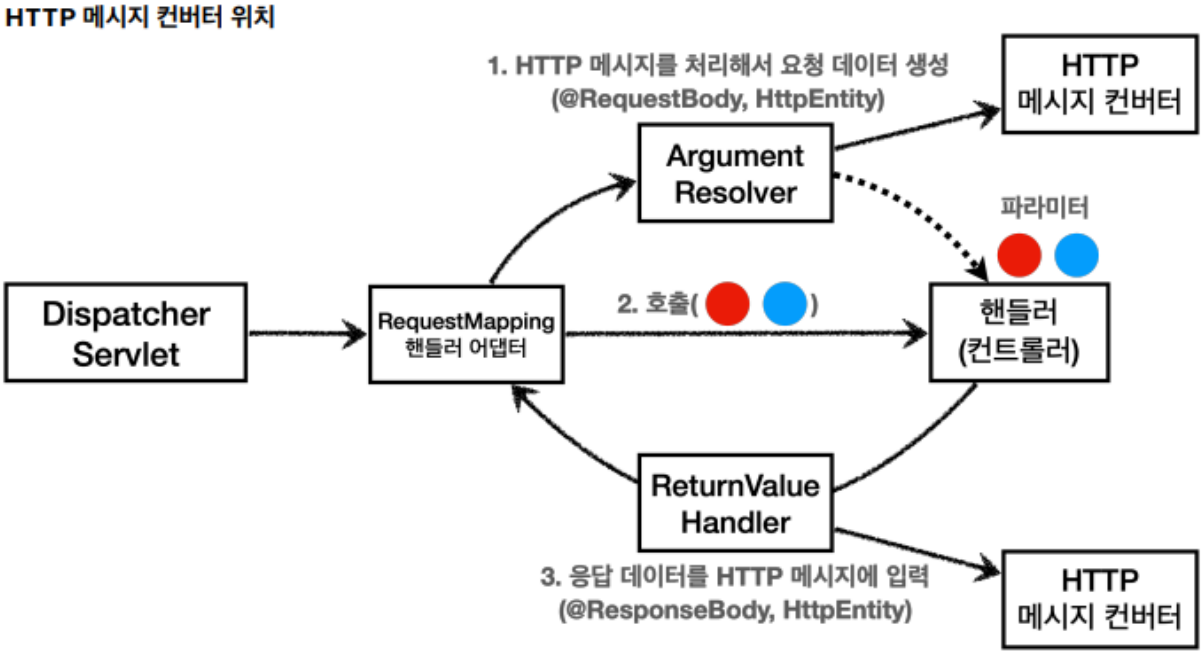
응답 값을 변환하고 처리한다.

→ 컨트롤러에서 String 뷰 이름을 반환해도, 동작하는 이유

-

`ModelAndView` , `@ResponseBody` , `HttpEntity` , `String` 등 지원

HTTP 메시지 컨버터



요청의 경우(read 호출)

- **ArgumentResolver**
  - HttpEntity 처리
    1. `supportsParameter()` 호출
    2. `resolveArgument()` 호출
       - `MessageConverters` 호출 → Body에 Message가 들어간다.
    3. `HttpEntity<>` 반환

응답의 경우(write 호출)

- **ReturnValueHandle**

스프링 MVC는 @RequestBody, @ResponseBody 가 있으면

RequestResponseBodyMethodProcessor((통합)ArgumentResolver) 를 사용한다.

스프링은 다음을 모두 인터페이로 제공하므로 언제든지 기능을
확장할 수 있다.

- HandlerMethodArgumentResolver
- HandlerMethodReturnValueHandler
- HttpMessageConverter

확장 시 @WebMvcConfigurer 사용