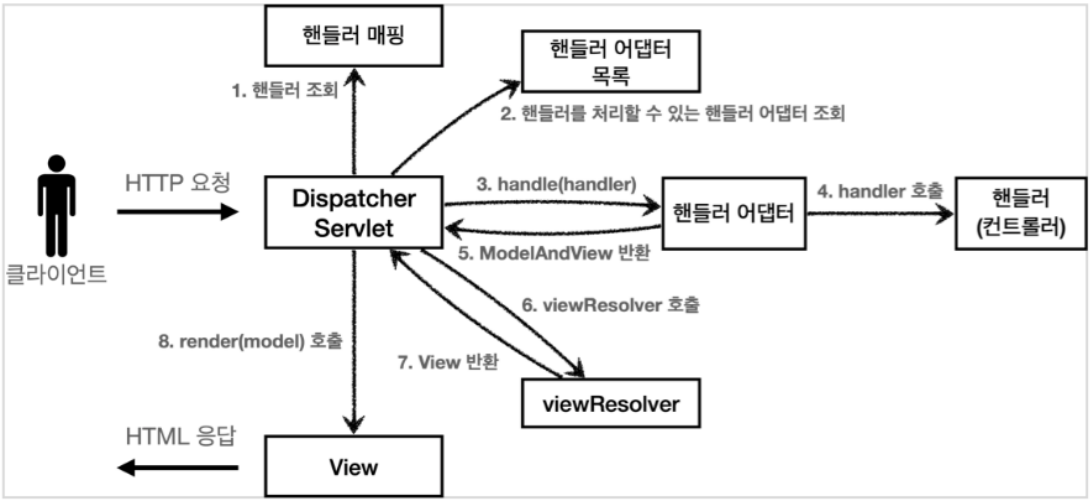
스프링 MVC - 구조 이해

스프링 MVC 전체 구조



스프링 MVC도 프론트 컨트롤러 패턴으로 구현되어 있다.

스프링 MVC 프론트 컨트롤러: 디스패처 서블릿(DispatcherServlet)

DispacherServlet 서블릿 등록

- DispatcherServlet도 부모 클래스에서 HttpServlet을 상속받아서 사용한다.
- 스프링 부트는 DispatcherServlet을 서블릿으로 자동으로 등록하면서 **모든경로**에 대해 매핑한다. (자세한 경로가 우선순위가 높다.)

요청흐름

- 서블릿이 호출되면 HttpServlet이 제공하는 service()가 호출된다.
- 스프링 MVC는 DispatcherServlet의 부모인 FrameworkServlet에서 service()를 오버라이드 해두었다.
- FranmeWorkServlet.service()를 시작으로 여러 메서드가 호출되면서

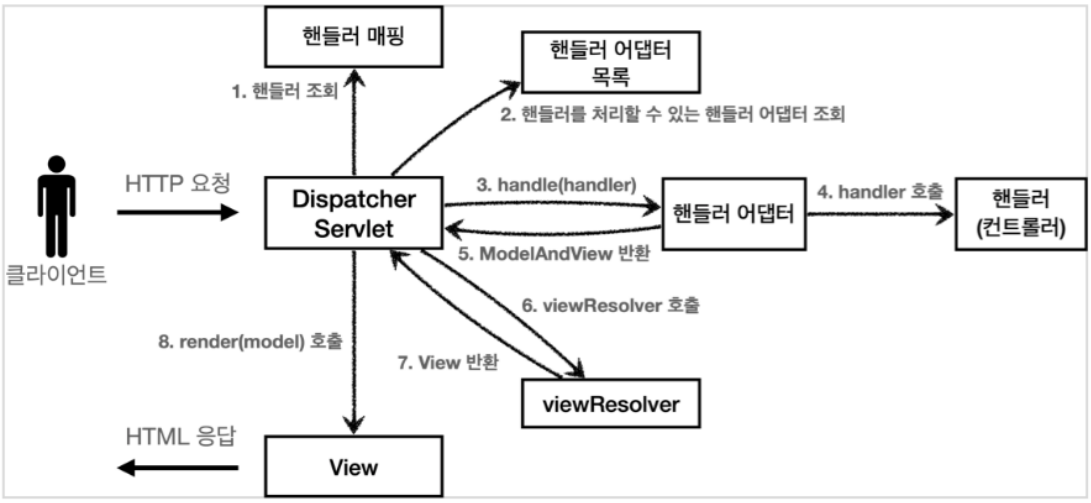
  DispatchServlet.doDispatch() 가 호출된다.

  [DispatcherServlet.doDispatch()]

```
protected void doDispatch(HttpServeletRequest request, HttpServletResponse
response) throws Exceptions{

  HttpServletRequest processedRequest = request;
  HandlerExcutionCahin mappedHandler = null;
  ModelAndView mv = null;

  // 1. 핸들러 조회
  mappedHandler = getHandler(processedRequest);
  if (mappedHandler == null){
    noHnadelerFound(proceesedRequest, response);
    return;
  }

  // 2. 핸들러 어댑터 조회 - 핸들러를 처리할 수 있는 어댑터
```

```
  HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

  // 3. 핸들러 어댑터 실행 -> 4. 핸들러 어댑터를 통해 핸들러 실행 -> 5.
ModelAndView 반환
  mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

  processDispatchResult(processedRequest, response, mappedHandler, mv,
dispatchException);

}

private void processDispatchResult(HttpSerletRequest request,
HttpServletResponse response, HandlerExcutionChain mappedHanlder,
ModelAndView mv, Exception exception) thorws Exception{

  // 뷰 랜더링 호출
  render(mv, request, response);
}

protected void render(ModelAndView mv, HttpServletRequest request,
HttpServletResponse response) throws Exception{

  View view;
  String viewName = mv.getViewName();

  // 6. 뷰 리졸버를 통해서 뷰 찾기, 7. View 반환
  view = resolveViewName(viewName, mv.getModelInternal(), locale, request);

  // 8. 뷰 렌더링
  view.render(mv.getModerInternal(), request, response);
}
```

SpringMVC 동작순서



1. **핸들러 조회**: 핸들러 매핑을 통해 요청 URL에 매핑된 핸들러(컨트롤러)를 조회한다.
2. **핸들러 어댑터 조회**: 핸들러를 실행할 수 있는 핸들러 어댑터를 조회한다.
3. **핸들러 어댑터 실행**: 핸들러 어댑터를 실행한다.
4. **핸들러 실행**: 핸들러 어댑터가 실제 핸들러를 실행한다.
5. **ModelAndView반환**: 핸들러 어댑터는 핸들러가 반환하는 정보를 ModelAndView로 변환해서 반환한다.
6. **viewResolver호출**: 뷰 리졸버를 찾고 실행한다.

7. **View반환**: 뷰 리졸버는 뷰의 논리 이름을 물리 이름으로 바꾸고, 렌더링 역할을 담당하는 뷰 객체를 반환한다.
8. **뷰 랜더링**: 뷰를 통해서 뷰를 렌더링 한다.

핸들러 매핑과 핸들러 어댑터

스프링은 이미 필요한 핸들러 매핑과 핸들러 어댑터를 대부분 구현해두었다.

개발자가 직접 핸들러 매핑과 핸들러 어댑터를 만드는 일은 거의 없다.

스프링부트가 자동 등록하는 핸들러 매핑과 핸들러 어댑터

[HandlerMapping]

0 = RequestMappingHandlerMapping : 에노테이션 기반의 컨트롤러인 @RequestMapping 에서 사용

1 = BeanNameUrlHandlerMapping : 스프링 빈의 이름으로 핸들러를 찾는다.

[HandlerAdapter]

0 = RequestMappingHandlerAdapter : 에노테이션 기반의 컨트롤러인 @RequestMapping에서 사용

1 = HttpRequestHandlerAdapter : HttpRequestHandler 처리

2 = SimpleControllerHanlerAdapter : Controller 인터페이스(에노테이션x , 과거에 사용)

스프링 MVC

스프링 MVC - 시작하기

@RequestMapping

스프링은 매우 유연하고, 실용적인 컨트롤러를 만들었는데 @RequestMapping 에노테이션을 사용하는 컨트롤러이다.

좀 더 세부화된 @PostMapping, @GetMapping은 @RequestMapping을 포함한다.

- RequestMappingHandlerMapping
- RequestMappingHandlerAdapter

[SpringmemberFormControllerV1] - 회원 등록 폼

```
@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 @Component가 있어 컴포넌트 스캔의 대상이 된다.)
  - 스프링 MVC에서 에노테이션 기반 컨트롤러로 인식된다.
- @RequestMapping

    - 요청 정보를 매핑한다.
    - 해당 URL이 호출되면 이 매서드가 호출된다.
- ModelAndView
    - 모델과 뷰 정보를 담아서 반환하면 된다.

[SpringMemberSaveControllerV1] - 회원 저장

```
@Controller
public class SpringMemberSaveControllerV1{

  private MemberRepository memberRepository = MemberRepository.getInstance();

  @RequestMapping("/springmvc/v1/members/save")
  public ModelAndView process(HttpServletRequest request, HttpServletResponse
response){

    String username = request.getParameter("username");
    int age = Integer.parseInt(request.getParameter("age"));

    Member member = new Member(username, age);

    System.out.println("member = " + member);
    memberRepository.save(member);

    ModelAndView mv = new ModelAndView("save-result");
    mv.addObject("member", member);
    return mv;
  }
}
```

- mv.addObject("member", member)
    - 스프링이 제공하는 ModelAndView 를 통해 Model 데이터를 추가할 때는 addObject를 사용하면 된다. 이 데이터는 뷰를 랜더링 할 때 사용된다.

[SpringMemberListControllerV1] - 회원목록

```
@Controller
public class SpringMemberListControllerV1{

  private MemberRepository memberRepository = MemberRepository.getInstance();

  @RequestMapping("/springmvc/v1/members")
  public ModelAndView process(){

    List<Member> members = memberRepository.findAll();

    ModelAndView mv = new ModelAndView("members");
    mv.getObject("members", members);
    return mv;
  }
}
```

스프링 MVC - 컨트롤러 통합

@RequestMapping은 클래스 단위가 아니라 메서드 단위에 적용된다.

따라서 컨트롤러 클래스를 유연하게 하나로 통합 할 수 있다.

[SpringMemberControllerV2]

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

  @RequestMapping("/new-form")
  public ModelAndView newForm(){
    return new ModelAndView("new-form");
  }

  @RequestMapping("/save")
  public ModelAndView save(HttpServletRequest request, HttpServletResponse
response){

    String username = request.getParameter("username");
    int age = Integer.parseInt(request.getParameter("age"));

    Member member = new Member(username, age);
    memberRepository.save(member);

    ModelAndView mav = new ModelAndView("save-result");
    mav.addObject("member", member);
    return mav;
  }

  @RequestMapping
  public ModelAndView members(){

    List<Member> members = memberRepository.findAll();

    ModelAndView mav = new ModelAndView("members");
    mav.addObject("members", members);
    return mav;
  }

}
```

/springmvc/v2/member라는 중복 부분을 위에 두면 밑에서 조합이 가능하다.

스프링MVC - 실용적인 방식(실제로 사용)

[SpringMemberControllerV3]

```
@Controller
@RequestMapping("/springmvc/v3/members")
public class SpringMemberControllerV3{
```

```
  private MemberRepository memberRepository = MemberRepository.getInstance();

  @GetMapping("/new-form")
  public Stirng newForm(){
    return "new-form";
  }

  @PostMapping("/save")
  public String save(
          @RequestParam("username") String username,
          @RequestParam("age") int age,
          Model model){

    Member member = new Member(username, age);
    memberRepository.save(member);

    model.addAtribute("member", member);
    return "save-result";
  }

  @GetMapping
  public String members(Model model){
    List<Member> members = memberRepository.findAll();
    model.addAttribute("members", members);
    return "members";
  }
}
```

Model 파라미터

- save(), members() 를 보면 Model을 파라미터로 받는것을 확인 할 수 있다.

  스프링이 제공해준다.

ViewName 직접 반환

- 뷰의 논리 이름을 반환할 수 있다.

@RequestParam 사용

- 스프링은 HTTP 요청 파라미터를 @RequestParam으로 받을 수 있다.
- @RequestParam("username") 은 request.getParameter("username")과 동일

@RequestMapping -> @GetMapping, @PostMapping

- @RequestMapping(value = "/new-form", method = RequestMethod.Get)

  과 같이 Method도 구분할 수 있다.

스프링 MVC - 기본 기능

로깅 알아보기

SLF4J 라이브러리를 사용 - 그 구현체로 Logback과 같은 로그 라이브러리를 선택

로그 선언

- private Logger log = LoggerFactory.getLogger(getClass());
- private static final Logger log = LoggerFactory.getLogger(xxx.class)

- **@Slf4j: 롬복 사용 가능**

로그 호출

- log.info("hello")

요청 매핑

기본

```
@RestController
public class MappingController{

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

  @GetMapping("/mapping-get-v2")
  public String mappingGetV2(){
    log.info("mapping-get-v2");
    return "ok"
  }
}
```

@RestController

- @Controller는 반환값이 String 이면 뷰 이름으로 인식된다. -> **뷰를 찾고 뷰가 랜더링**
- @RestController는 반환 값으로 뷰를 찾는 것이 아니라, **HTTP 메시지 바디에 바로 입력 한다.** -> 따라서 바로 ok 메시지를 받을 수 있다.

PathVariable(경로 변수) 사용

```
@GetMapping("/mapping/{userId}")
public String mappingPath(@PathVariable("userId") String data){
  log.info("mappingPath userid={}", data);
  return "ok";
}
```

PathVariable사용 - 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";
}
```

최근 HTTP API는 다음과 같이 리소스 경로에 식별자를 넣는 스타일을 선호한다.

- /mapping/userA
- /users/1
- @RequestMapping은 URL경로를 템플릿화 할 수 있는데, @PathVariable을 사용하면 매칭되는 부분을 편하게 조회 할 수 있다.
- @PathVariable의 이름과 파라미터 이름이 같으면 생략 할 수 있다.

요청 매핑 - API 예시

```
@RestController
@RequestMapping("mapping/users")
public class MappingClassController{

  // GET 회원 목록 조회
  @GetMapping
  public String users(){
    return "get users";
  }

  // POST 회원 등록
  @PostMapping
  public String addUser(){
    return "post user";
  }

  // GET 회원 조회
  @GetMapping("/{userId}")
  public String findUser(@PathVariable String userId){
    return "get userId=" + userId;
  }

  // PATCH 회원 수정
  @PatchMapping("/{userId}")
  public String updateUser(@PathVariable String userId){
    return "update userId=" + userId;
  }

  // DELETE 회원 삭제
  @DeleMapping("/{userId}")
  public String deleteUser(@PathVairable String userId){
    return "delete userId=" + userId;
  }
}
```

HTTP 요청 파라미터 - 쿼리 파라미터, HTML Form

클라이언트에서 서버로 요청 데이터를 전달하는 방법

1. **GET - 쿼리 파라미터**
    - /url**?username=hello?age=20**
    - 메시지 바디없이, URL의 쿼리 파라미터에 데이터를 포함해서 전달
    - 검색,필터,페이징등에서 많이 사용
2. **POST - HTML Form**
    - content-type : application/x-www-form-urlencoded
    - 메시지 바디에 쿼리 파라미터 형식으로 전달 username=hello&age=20
    - 회원 가입, 상품 주문, HTML Form 사용
3. **HTTP message body**에 데이터를 직접 담아서 요청
    - HTTP API에서 주로 사용 -JSON, XML, TXET
    - 데이터 형식은 주로 JSON 사용

- POST, PUT, PATCH

GET 쿼리 파라미터 전송방식이든, POST HTML Form 전송 방식이든 둘다 형식이 같으므로 구분없이

요청 파라미터(request parameter) 조회를 통해 조회할 수 있다.

[RequestParamController]

```
@Slf4j
@Controller
public class RequestParamController{

    @RequestMapping("/request-param-v1")
    public void requestParamV1(HttpServletRequest request, HttpServletResponse
response) throws IOException{

        String username = request.getParameter("username");
        int age = Integer.parseInt(request.getParameter("age"));
        log.info("username={}, age={}", username, age);

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

}
```

변환 타입이 없으면서 응답에 값을 직접 집어넣으면, view 조회 x

HTTP 요청 파라미터 - @RequestParam

스프링에서 제공하는 @RequestParam을 사용하면 요청 파라미터를 매우 편리하게 사용할 수 있다.

[requestParamV2]

```
@ResponseBody
@RequestMapping("/request-param-v2")
public String requestParamV2(
        @RequestParam("username") String memberName,
        @RequestParam("age") int memberAge){

    log.info("username={}, age={}", memberName, memberAge);
    return "ok";
}
```

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

@RequestParam의 name(value) 속성이 파라미터 이름으로 사용된다.

- @RequestParam("**username**") String **memberName** -> request.getParameter("**username**")

[requestParamV3] ☆

```
@ResponseBody
@RequestMapping("/request-param-v3")
public String requestParamV3(
        @RequestParam String username,
        @RequestParam int age){
    log.info("username={}, age={}", username, age);
    return "ok";
}
```

- HTTP 파라미터 이름이 변수 이름과 같으면 @RequestParam(name="xx") 생략이 가능하다.

[requestParamV4]

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

- String, int, Integer등의 단순 타입이면 @RequestParam도 생략이 가능하다.
- 생략을 하게 되면 MVC내부에서 required = false를 적용한다.

파라미터 필수 여부 - requestParamRequired

```
@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";
}
```

- @RequestParam.required
  - 파라미터 필수 여부
  - 기본값이 파라미터 필수(true) 이다.
- /request-param 요청
  - username이 없으므로 400 예외가 발생한다.
  - /request-param?username= 파라미터 이름만 있고 값이 없는 경우 -> 빈문자로 통과한다.
- null을 int에 입력하는 것이 불가능(500예외가 발생) -> null을 받을 수 있는 Integer로 변경하거나, defaultValue 사용

기본 값 적용 - requestParamDefault

```
@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";
}
```

파라미터에 값이 없는 경우 defaultValue를 사용하면 기본 값을 적용할 수 있다.

파라미터를 Map으로 조회하기 - requestParamMap

```
@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";
}
```

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 사용해도 괜찮지만, 그렇지 않다면 MultiValueMap을 사용

HTTP 요청 파라미터 - @ModelAttribute

실제 개발을 하면 요청 파라미터를 받아서 필요한 객체를 만들고 그 객체에 값을 넣어주어야 한다.

@ModelAttribute적용 - modelAttributeV1

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

- HelloData 객체가 생성되고, 요청 파라미터의 값도 모두 들어가 있다.

@ModelAttribute의 작용

1. HelloData 객체를 생성한다.
2. 요청 파라미터의 이름으로 HelloData 객체의 프로퍼티(getter, setter...)를 찾는다.
3. 해당 프로퍼티의 setter를 호출해서 파라미터의 값을 입력(바인딩)한다.

ex) 파라미터 이름이 username이면 setUsername()메서드를 찾아 호출하면서 값을 입력한다.

@ModerAttriubte생략 - modelAttributeV2

```
@ResponseBody
@RequestMapping("/model-attribute-v2")
public String modelAttributeV2(HelloData helloData){

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

- @ModelAttribute는 생략할 수 있다.
- @RequestParam도 생략 가능하므로 혼란이 발생할 수 있다.

스프링의 규칙

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

HTTP 요청 메시지 - 단순 텍스트

HttpEntity - requestBodyStringV3

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

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

스프링 MVC는 다음 파라미터를 지원한다.

HttpEntity: HTTP header, body 정보를 편리하게 조회

- 메시지 바디 정보를 직접 조회
- 요청 파라미터를 조회하는 기능과 관계 없음 @RequestParam X, @ModelAttribute X

응답에도 사용가능 한 HttpEntity

- 메시지 바디 정보 직접 반환
- 헤더 정보 포함 가능
- view 조회 X

HttpEntitiy를 상속 받은 객체들

- **RequestEntity**
  - HttpMethod, url 정보가 추가, **요청에서** 사용한다.
- **ResponseEnetiy**
  - HTTP상태 코드 설정 가능, **응답에서** 사용한다.
  - return new ResponseEntity("Hello World", responseHeaders, HttpStatus.CREATED)

@RequestBody - requestBodyStringV4

```
@ResponseBody
@PostMapping("/request-body-string-v4")
public String requestBodyStringV4(@RequestBody String messageBody){
    log.info("messageBody={}", messageBody);
    return "ok";
}
```

@RequestBody

HTTP 메시지 바디 정보를 편리하게 조회할 수 있다.

만약, 헤더 정보가 필요하면 HttpEntity를 사용하거나, @RequestHeader를 사용하면 된다.

@ResponseBody

HTTP 메시지 바디에 직접 담아서 전달할 수 있다. - 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

HTTP API에서 주로 사용

requestBodyJsonV3 - @RequestBody 객체 변환

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

- HttpEntity, @RequestBody를 사용하면 HTTP 메시지 컨버터가 HTTP메시지 바디의 내용을 우리가 원하는 문자나 객체등으로 반환
- @RequestBody는 생략 불가
  - 생략하면 @ModelAttribute가 적용되어 HTTP바디가 아닌 요청 파라미터를 처리하게 된다.

requestBodyJsonV4 - HttpEntity

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

requestBodyJsonV5 - 객체를 HTTP 메시지 바디에 직접 반환

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

HTTP응답 - 정적 리소스, 뷰 템플릿

스프링(서버)에서 응답 데이터를 만드는 방법은 크게 3가지

1. 정적 리소스
    - 웹 브라우저에 정적인 HTML, css, js을 제공할 때, **정적 리소스**를 사용
2. 뷰 템플릿 사용
    - 웹 브라우저에 동적인 HTML을 제공할 때 뷰 템플릿을 사용한다.
3. HTTP 메시지 사용
    - HTTP API를 제공하는 경우에는 HTML이 아니라 데이터를 전달해야 하므로, HTTP 메시지 바디에 JSON 같은 형식으로 데이터를 실어 보낸다.

1. 정적 리소스

스프링 부트는 클래스패스의 다음 디렉토리에 있는 정적 리소스를 제공한다.

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

/src/main/resources는 리소스를 보관하는 곳이고, 또 클래스패스의 시작 경로이다.

정적 리소스 경로

src/main/resources/static

src/main/resouces/static/basic/hello-form.html

----> 웹브라우저: http://localhost:8080/basic/hello-form.htm 서비스

2. 뷰 템플릿

- 뷰 템플릿을 거쳐서 HTML이 생성되고, 뷰가 응답을 만들어서 전달한다.
- 컨트롤러로 뷰 템플릿을 호출

뷰 템플릿 경로

src/main/resources/templates

뷰 템플릿 생성 src/main/resources/templates/response/hello.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<p th:text="${data}">empty</p>
</body>
</html>
```

뷰 템플릿을 호출하는 컨트롤러

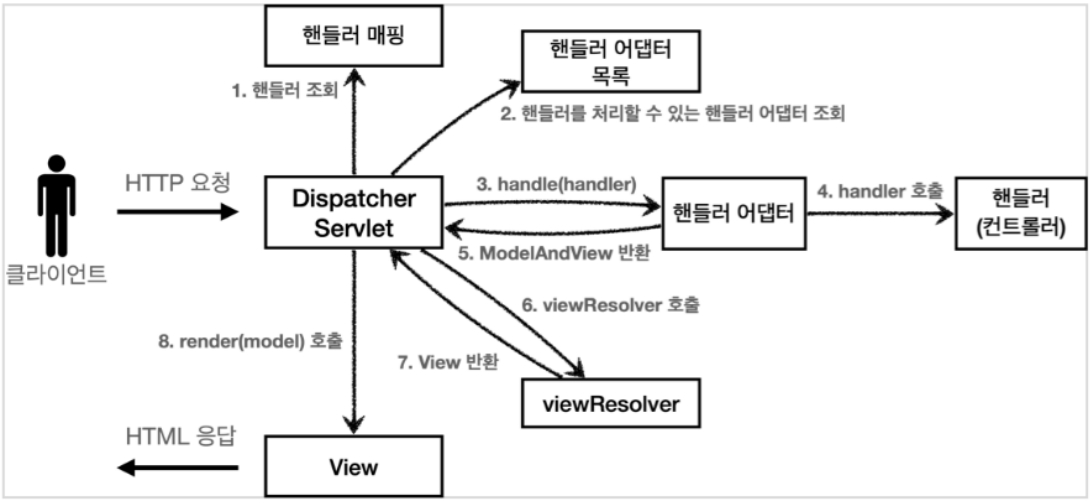
```
@Controller
public class ResponseViewController{

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

뷰의 논리 이름인 response/hello를 반환하면 **templates/response/hello.html** 의 뷰 템플릿이 렌더링

요청 매핑 헨들러 어뎁터 구조

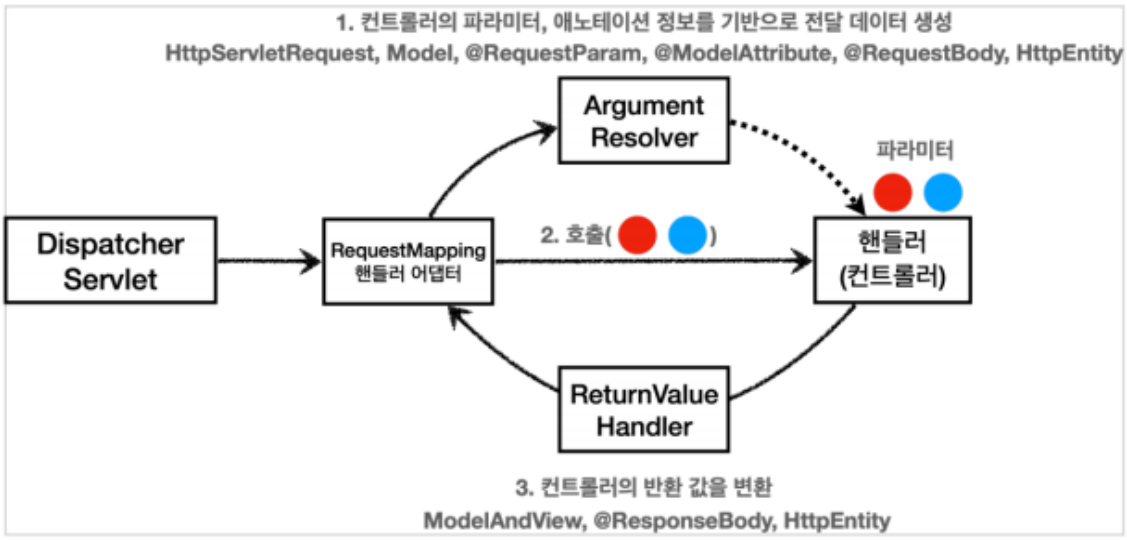
HTTP 메시지 컨버터는 스프링 MVC 어디쯤에 사용되는 것일까?

SpringMVC 구조



- 에노테이션 기반의 컨트롤러 : @RequestMapping을 처리하는 핸들러 어댑터인 RequestMappingHandlerAdapter(요청 매핑 핸들러 어뎁터)를 확인

RequestMappingHandlerAdapter 동작 방식



ArgumentResolver

에노테이션 기반의 컨트롤러는 매우 다양한 파라미터를 사용할 수 있다.

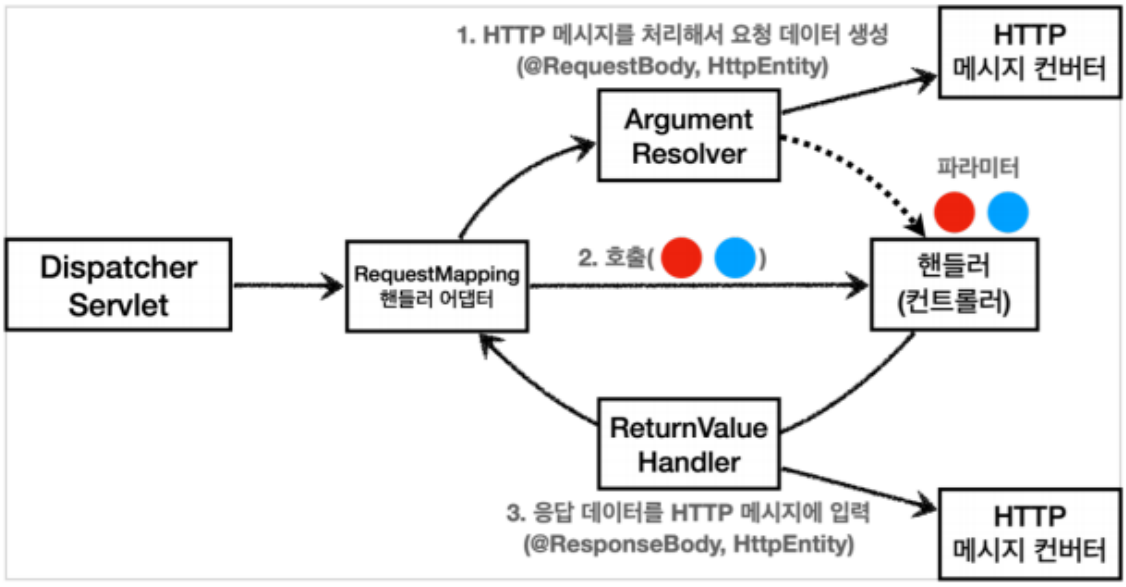
HttpServletRequest, Model은 물론 @RequestParam, @ModelAttribute같은 에노테이션 그리고

@RequestBodyt, HttpEntity 같은 HTTP메시지 처리하는 부분까지 매우 큰 유연함을 가지고 있다.

RequestMappingHandlerAdapter 는 **ArgumentResolver**를 호출해서 컨트롤러(핸들러)가 필요하는 다양한 파라미터의 값(객체)를 생성한다.

-> 파라미터의 값이 준비되면 컨트롤러를 호출하면서 값을 넘겨줌

HTTP 메시지 컨버터 위치



- HTTP 메시지 컨버터를 사용하는 @RequestBody도 컨트롤러가 필요하는 파랕미터의 값에 이용
- @ResponseBody의 경우도 컨트롤러의 반환 값을 이용

요청

- @RequestBody를 처리하는 ArgumentResolver가 있고, HttpEntity를 처리하는 ArgumentResolver가 있다.
- ArgumentResolver들이 HTTP 메시지 컨버터를 사용해서 필요한 객체를 생성

응답

- @ResponseBody와 HttpEntity를 처리하는 ReturnValueHandler가 있다.
- ReturnValueHandler에서 HTTP 메시지 컨버터를 호출해서 응답결과를 만든다.