● HTTP 요청 메세지 - 단순 텍스트

HTTP 메세지 바디에 데이터를 직접 담아서 요청하는 방식은 HTTP API 방식에서 주로 사용된다. 메세지 바디에는 JSON, XML, TEXT 등을 넣을 수 있으며, 그 중 JSON이 가장 인기가 많다. 이 상황에서 주로 사용되는 HTTP 메소드는 POST, PUT, PATCH이다.

HTTP 메세지 바디에 데이터를 직접 담아서 요청하는 방식은, 요청 파라미터를 받을 때와는 달리 @RequestParam, @ModelAttribute를 사용하여 데이터를 받아낼 수 없다. 메세지 바디를 조회하는 것과 요청 파라미터를 조회하는 것은 전혀 다른 개념이기 때문이다. (HTML form 형식으로 전달되는 경우는 예외.)

예제 코드를 통해 단순 텍스트를 받아들이는 방법을 알아보자.

```
@Slf4j
@Controller
public class RequestBodyStringController {
    @PostMapping("/request-body-string-v1")
    public void requestBodyString(HttpServletRequest request, HttpServletResponse response) throws IOException {
        // 요청 메세지 바디의 입력 스트림을 검색하여, 요청 데이터의 raw data를 읽는다.
        ServletInputStream inputStream = request.getInputStream();
        // messageBody에 검색한 입력 스트림을 카피에서 String 형태로 저장
        String messageBody = StreamUtils.copyToString(inputStream,
                StandardCharsets.UTF_8);   // String의 default는 byte 코드이기 때문에, 인코딩이 필요하면 추가로 옵션을 넣어줘야 함.

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

        response.getWriter().write("ok");   // 요청 메세지 바디에 직접 담아서 출력.
    }
}
```

Postman에서 Body > raw에서 text를 선택하고 아무거나 넣어서 입력하면, 콘솔에서 입력한대로 로그가 남는 것을 볼 수 있다.

이번에는 HttpServlet을 이용하는 것이 아닌, InputStream과 Writer를 통해 직접 메세지 바디에 접근해보도록 하자.

```
    @PostMapping("/request-body-string-v2")
    public void requestBodyStringV2(InputStream inputStream,   // HTTP 요청 메세지 바디의 내용을 직접 조회하게 해줌.
                                    Writer responseWriter)   // HTTP 응답 메세지 바디에 내용을 직접 작성하게 해줌.
            throws IOException {
        String messageBody = StreamUtils.copyToString(inputStream,
                StandardCharsets.UTF_8);
```

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

        responseWriter.write("ok");  // 요청 메세지 바디에 직접 담아서 출력.
    }
```

마찬가지로 Postman에서 Body > raw에서 text를 선택하고 아무거나 넣어서 입력하면, 콘솔에서 입력한대로 로그가 남는 것을 볼 수 있다.

이번에는 Spring에서 지원하는 HttpEntity를 활용해서 편리하게 개발해보도록 하자.

```
    @PostMapping("/request-body-string-v3")
    public HttpEntity<String> requestBodyStringV3(HttpEntity<String> httpEntity){
        String messageBody = httpEntity.getBody();  // Http 요청 메세지 바디의 내용을 가져옴.
        HttpHeaders headers = httpEntity.getHeaders();  // 헤더 정보도 가져올 수 있다.
        log.info("messageBody={}", messageBody);
        log.info("headers={}", headers);
        return new HttpEntity<>("ok");  // Http 응답 메세지 바디에 데이터를 채워서 전송. 뷰를 조회하지 않음.
    }
```

마찬가지로 Postman에서 Body > raw에서 text를 선택하고 아무거나 넣어서 입력하면, 콘솔에서 입력한대로 로그가 남는 것을 볼 수 있다. 이것이 가능한 이유는 Spring 내부에서 Http 메세지 바디를 읽어서 문자나 객체로 변환해서 전달하기 때문이다. 이에 대한 구체적인 원리는 추후 HttpMessageConverter 편에서 알아보도록 하자.

HttpEntity를 상속받는 RequestEntity와 ResponseEntity는 조금 더 부가적인 기능들을 제공한다.

```
- RequestEntity : Http 메소드나 URL 정보를 추가할 수 있다.
- ResponseEntity : Http 상태코드를 설정할 수 있다.
```

어노테이션을 활용하면 위 코드를 더 간단히 표현할 수 있다. 실무에서 많이 쓰이는 방식이기도 하다.

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

헤더 정보가 필요하다면 @RequestHeader를 사용하면 된다.

이 장의 결론은 이것이다.

```
요청 파라미터 조회 : @RequestParam, @ModelAttribute 를 쓰자.
Http 요청 메세지 바디 조회 : @RequestBody 를 쓰자.
```

● HTTP 요청 메세지 - JSON

이번에는 주로 사용하는 JSON을 조회하는 방법을 알아보자.

먼저 처음에는 서블릿에서부터 시작해보자.

```
@Slf4j
@Controller
public class RequestBodyJsonController {
    private ObjectMapper objectMapper = new ObjectMapper();  // JSON 데이터를 받기 위한 객체

    @PostMapping("/request-body-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);
        // objectMapper를 통해 messageBody를 HelloData에 맞춰 자바 객체로 변환.
        HelloData helloData = objectMapper.readValue(messageBody, HelloData.class);
        log.info("username = {}, age = {}", helloData.getUsername(), helloData.getAge());

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

요청 메세지 바디에 { "username" : "hello", "age" : 20} 를 JSON 형식으로 넣어주고 보내면 데이터가 잘 보내진 것을 확인할 수 있다.

이번에는 @RequestBody를 사용하여 요청 메세지를 파싱해보자. 앞에 했던 거 그대로라 특이사항은 없다.

```
    @ResponseBody
    @PostMapping("/request-body-json-v2")
    public String requestBodyJsonV1(@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";
    }
```

근데 굳이 번거롭게 ObjectMapper를 써야 할까? 다행히도, @RequestBody는 내가 직접 만든 객체를 @RequestBody에 지정하는 것을 지원해준다.

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

이것도 메세지 컨버터 덕분에 가능한 일이다.

참고: @RequestBody를 생략하는 경우, 그 자리에 @RequestParam 또는 @ModelAttribute가 자동으로 들어간다. 그래서 의도치 않게 동작하게 되므로 절대 생략하지 말 것.

앞서 배운대로, `HttpEntity`를 활용하여 위와 동일한 코드를 만들 수 있다.

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

이번에는 응답 메세지 바디에도 내가 직접 만든 객체를 넣어서 반환해보자. 이건 @ResponseBody를 활용하면 된다.

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

● 응답 - 정적 리소스, 뷰 템플릿

스프링에서 응답 데이터를 만드는 방법은 크게 3가지이다.

⦿ 정적 리소스

웹 브라우저에 정적인 HTML, css, js를 제공할 때는 정적 리소스 방식으로 응답 데이터를 만든다. 스프링 부트는 클래스 패스의 시작 경로를 `/src/main/resources`로 지정하며, 이곳은 곧 리소스를 보관하는 곳이기도 하다. 스프링 부트는 다음 디렉토리에 있는 정적 리소스를 제공한다.

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

예를 들면, src/main/resources/static/basic/hello-form.html 경로에 파일이 저장되어 있으면, 웹 브라우저에서 http://localhost:8080/basic/hello-form.html 로 접속하면 스프링 서버에서는 해당 파일을 변경없이 그대로 리턴한다.

⦿ 뷰 템플릿

웹 브라우저에 동적인 HTML을 제공할 때는 뷰 템플릿을 사용한다. 뷰 템플릿을 거쳐서 HTML이 생성되며, 뷰가 응답을 받아서 클라이언트에게 전달한다. 뷰 템플릿은 HTML을 동적으로 생성하는데 주로 쓰인다. 스프링 부트의 기본 뷰 템플릿 경로는 아래와 같다.

src/main/resources/templates

그러면 `src/main/resources/templates/response/hello.html` 경로에 뷰 템플릿을 만들고, 이를 호출하여 결과를 확인해보자.

```
<!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 템플릿 호출 컨트롤러
@Controller
public class ResponseViewController {
  @RequestMapping("/response-view-v1")
  public ModelAndView responseViewV1() {
  ModelAndView mav = new ModelAndView("response/hello")   // 논리 viewName을 넣어줌.
  .addObject("data", "hello!");
```

```
  return mav;   // ModelAndView 객체 자체를 리턴
  }

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

스프링 부트가 자동으로 ThymeleafViewResolver와 필요한 스프링 빈들을 등록해준다.

참고: thymeleaf의 기본 경로를 재설정해줄 수 있다. application.properties에서 아래 코드를 추가한다.

```
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
```

스프링 부트의 타임리프 관련 추가 설정은 다음 공식 사이트를 참고하자. (페이지 안에서 thymeleaf 검색)
https://docs.spring.io/spring-boot/docs/2.4.3/reference/html/appendix-applicationproperties.html#common-application-properties-templating

⦿ HTTP 메세지 사용

HTTP API (또는 REST API)를 제공하는 경우에는 HTTP 메세지 바디에 JSON 같은 형식으로 데이터를 담아 전달한다. `@ResponseBody`나 `HttpEntity`를 사용하여 Http 응답 메세지 바디에 데이터를 넣어 전달한다. 뷰 템플릿은 필요없다.

● HTTP 응답 - HTTP API, 메세지 바디에 직접 입력

HTML이나 뷰 템플릿을 사용해도 HTTP 응답 메세지 바디에 HTML 데이터가 담겨서 전달된다. 하지만 관련 내용은 앞에서 설명했으니, 이번에는 HTTP 응답 메세지를 직접 전달하는 방법에 대해 알아보자. 허나 이 내용들도 앞에서 설명했던 내용들이라 복습한다는 느낌으로 보자.

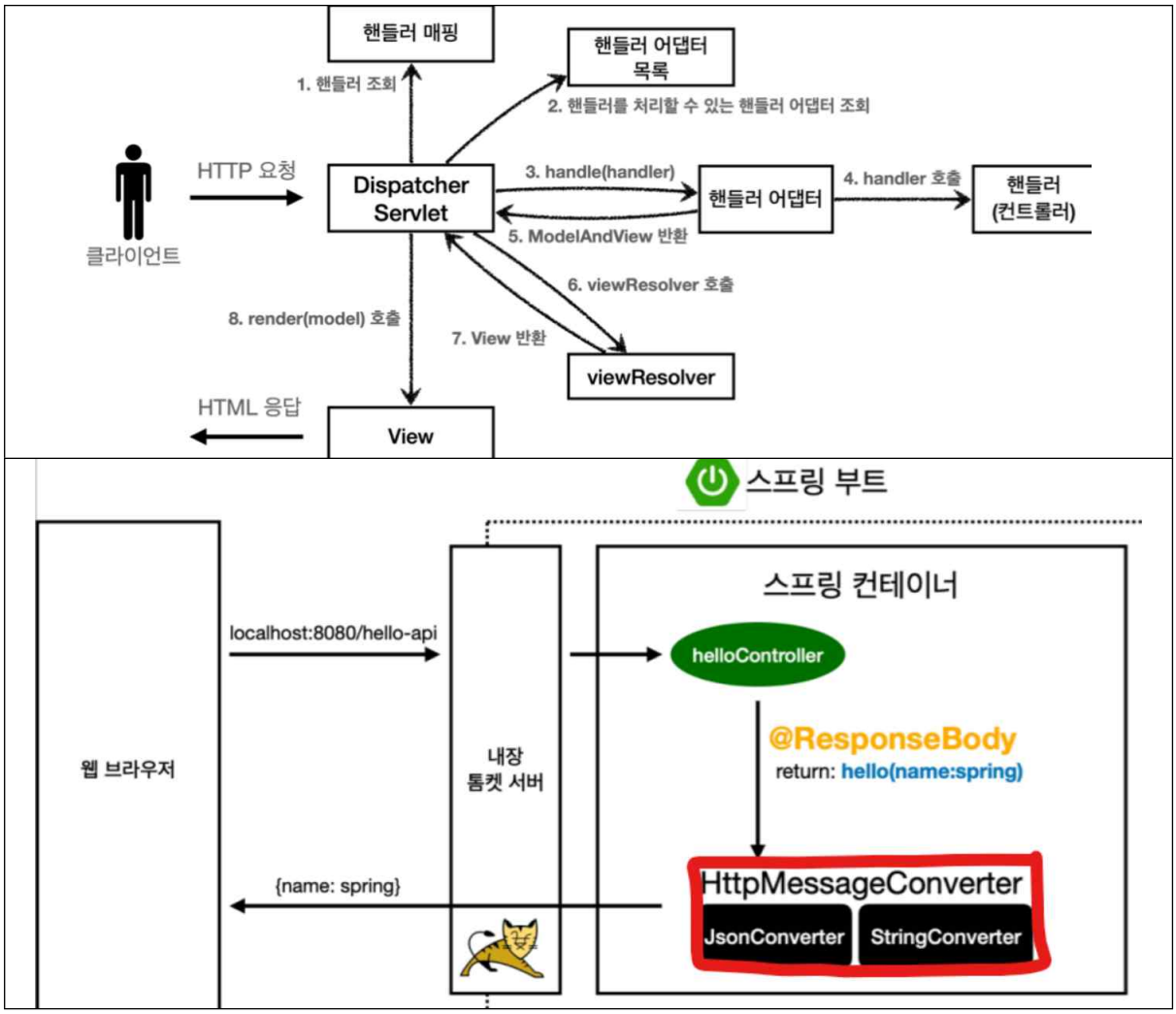
```
@Slf4j
@Controller
// @ResponseBody를 클래스 레벨에서도 사용할 수 있음. 모든 메소드에 @ResponseBody 적용됨.
//@RestController = @Controller + @ResponseBody
public class ResponseBodyController {
    // HttpServlet을 활용한 Text 응답 메세지 바디 전달 방식
    @GetMapping("/response-body-string-v1")
    public void responseBodyV1(HttpServletResponse response) throws IOException
    {
        response.getWriter().write("ok");
    }

    // HttpEntity를 상속받은 ResponseEntity를 활용한 Text 응답 메세지 바디 전달 방식
    @GetMapping("/response-body-string-v2")
    public ResponseEntity<String> responseBodyV2() {
        return new ResponseEntity<>("ok", HttpStatus.OK);
    }

    // @ResponseBody를 활용한 Text 응답 메세지 바디 전달 방식
    @ResponseBody
    @GetMapping("/response-body-string-v3")
    public String responseBodyV3() {
        // @ResponseBody를 적용하면 리턴값이 논리 viewName이 아닌 Text로 응답 메세지 바디에
담김.
        return "ok";
    }

    // HttpEntity를 활용한 Json 응답 메세지 바디 전달 방식 (상속받은 ResponseEntity를 활용)
    @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);
    }

    // @ResponseBody를 활용한 Json 응답 메세지 바디 전달 방식
    @ResponseStatus(HttpStatus.OK)   // @ResponseBody의 문제점인 HttpStatus 설정 불가 해결.
    @ResponseBody
```

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

● HTTP 메세지 컨버터

앞서 정적 리소스와 뷰 템플릿을 통해 HTTP 응답 메세지를 전송하는 방법에 대해 알아보았다. 또한 HTTP API (REST API) 방식으로 데이터를 주고 받을 때는 `**HTTP 메세지 컨버터**`가 작동한다고 언급만 했었는데, 이제 이에 대해 알아보도록 하자.

먼저 `@ResponseBody`의 작동 메커니즘을 다시 리마인드해보자.



`@ResponseBody`를 사용하면, 기존에 `viewResolver`를 동작시키는 방식에서 `HttpMessageConverter`가 대신 동작하게 된다. 그래서 HTTP 응답 메세지 바디에 데이터를 직접 넣어서 클라이언트에게 반환하게 된다. 이외에도 HTTP 응답에서 HttpEntity(ResponseEntity)를 사용하거나, HTTP 요청에서 @RequestBody 또는 HttpEntity(RequestEntity)를 사용할 때에도 HttpMessageConverter가 동작하게 된다. 즉, **HttpMessageConverter는 HTTP 요청할 때와 응답할 때 모두 사용된다는 것이다.**

`**HttpMessageConverter**`의 종류는 다양하게 존재한다. 대표적으로 아래 세 가지에 대해서 알아보자. 아래에서 소개되는 순서대로 메세지 컨버터를 조회하게 된다. `클래스 타입`과 `미디어 타입`의 차이점에 대해 집중해서 보자.

⦿ ByteArrayHttpMessageConverter

`byte[]` 형태의 데이터를 처리한다. 클래스 타입은 byte[]이어야 한다. 읽기 미디어 타입은 모두 허용된다.

요청 예시 : @RequestBody byte[] requestData
응답 예시 : @ResponseBody return byte[] 쓰기 미디어 타입은 `application/octet-stream`이어야 한다.

⦿ StringHttpMessageConverter

String 타입으로 데이터를 처리한다. 클래스 타입은 String이어야 한다. 읽기 미디어 타입은 모두 허용된다.

요청 예시 : @RequestBody String requestData
응답 예시 : @ResponseBody return "ok" 쓰기 미디어 타입은 `text/plain`이어야 한다.

⦿ MappingJackson2HttpMessageConverter

객체나 HashMap 타입으로 데이터를 처리한다. 내가 직접 만든 객체여도 상관없다. 클래스 타입은 객체나 HashMap이어야 하고, 읽기 미디어 타입은 반드시 `application/json` 관련된 것이어야 한다.

요청 예시 : @RequestBody HelloData helloData
응답 예시 : @ResponseBody return helloData 쓰기 미디어 타입은 `application/json` 관련된 것이어야 한다.

이번에는 HTTP 요청 데이터를 읽고, HTTP 응답 데이터를 쓰는 방식에 대해 알아보자.

⦿ HTTP 요청 데이터 읽기

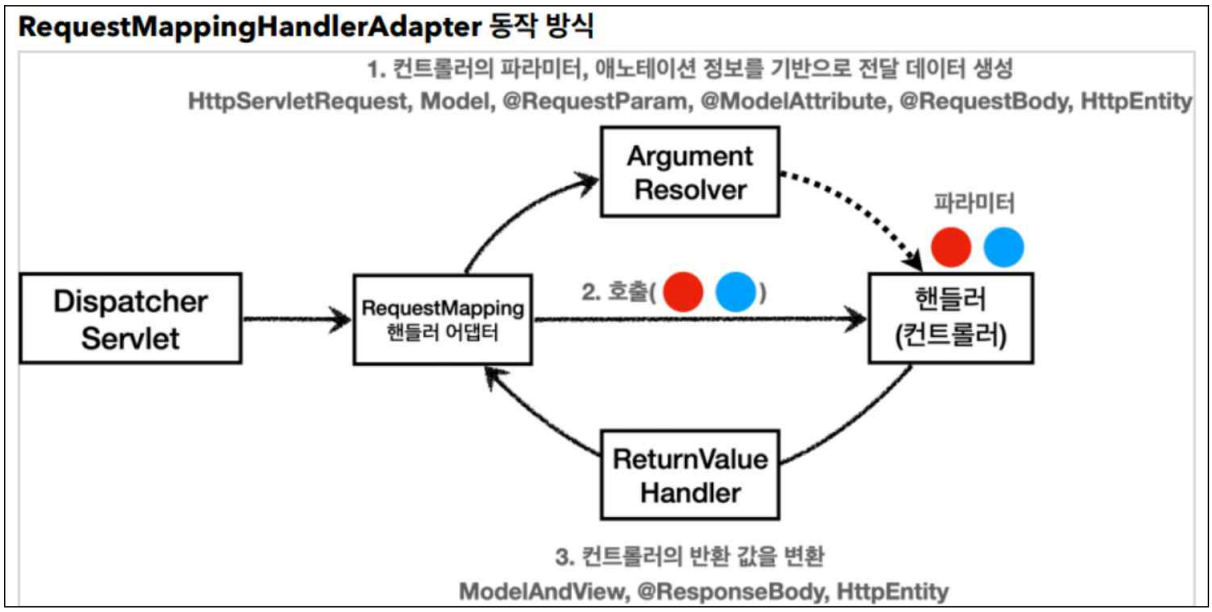
HTTP 요청이 들어오면, 컨트롤러에서는 @RequestBody 또는 HttpEntity 파라미터를 사용하여 요청 메세지를 읽고자 한다. 그러면 `HttpMessageConverter`가 메세지 바디 내용을 어떻게 읽어야 하는지 확인하기 위해 canRead() 메소드를 호출한다. canRead()에서는 대상 클래스 타입을 지원하는지(byte[], String, 커스텀 객체 등), 그리고 Content-Type 미디어 타입을 지원하는지(*/*, text/plain, application/json 등)를 확인한다. 만약 canRead() 조건을 만족한다면, read()를 호출하여 객체를 생성하고 반환한다.

⦿ HTTP 응답 데이터 쓰기

HTTP 응답 메세지를 작성하기 위해서는, 컨트롤러에서 @ResponseBody 또는 HttpEntity를 사용하여 값을 반환해야 한다. 응답 메세지 작성을 try하게 되면, 이번에는 HttpMessageConverter가 메세지 바디에 작성할 수 있는 내용인지 확인하기 위해 canWrite()를 호출한다. canWrite()에서는 대상 클래스 타입을 지원하는지(byte[], String, 커스텀 객체 등), 그리고 Http 요청 메세지에서 받은 Accept 미디어 타입을 지원하는지(*/*, text/plain, application/json 등)를 확인한다. 만약 canWrite() 조건을 만족한다면, write()를 호출하여 HTTP 응답 메세지 바디에 내용을 작성하고 반환하게 된다.

● 요청 매핑 핸들러 어댑터 구조

`HTTP 메세지 컨버터`는 `@RequestMapping`을 처리하는 `**RequestMappingHandlerAdapter(요청 매핑 헨들러 어뎁터)**`에 의해 처리된다. Spring MVC 모델에서는 아래 과정처럼 동작한다.



`**RequestMappingHandlerAdapter**`의 동작 방식에서 주목해야 할 요소는 `**ArgumentResolver(정확히는 HandlerMethodArgumentResolver)**`와 `**ReturnValueHandler(정확히는 HandlerMethodReturnValueHandler)**`이다.

⦿ ArgumentResolver (HandlerMethodArgumentResolver)

`**ArgumentResolver**`에 의해, 어노테이션 기반으로 구성된 컨트롤러는 매우 다양한 파라미터들을 사용할 수 있다. HTTP 요청 메세지의 파라미터를 처리하기 위한 HttpServletRequest, Model, @RequestParam, @ModelAttribute, 그리고 HTTP API 메세지 바디를 처리하는 @RequestBody, HttpEntity까지... 이렇게 수많은 컨트롤러의 파라미터를 유연하게 처리할 수 있는 이유가 전부 `**ArgumentResolver**` 덕분이라는 것이다.

`**RequestMappingHandlerAdapter**`는 `**ArgumentResolver**`를 호출하여 컨트롤러(또는 핸들러)가 필요로 하는 여러가지 파라미터의 값(또는 객체)를 생성하여, 어노테이션 기반의 컨트롤러를 처리한다. Spring에서는 30개가 넘는 `**ArgumentResolver**`들을 기본으로 제공하고 있다.

`**ArgumentResolver (HandlerMethodArgumentResolver)**`의 동작 방식을 코드로 살펴보자.

```
public interface HandlerMethodArgumentResolver {
   boolean supportsParameter(MethodParameter parameter);

   @Nullable
   Object resolveArgument(MethodParameter parameter,
   @Nullable ModelAndViewContainer mavContainer,
```

```
    NativeWebRequest webRequest,
    @Nullable WebDataBinderFactory
    binderFactory) throws Exception;
}
```

`**ArgumentResolver**`가 호출되면, 먼저 `supportsParameter()`를 호출하여 컨트롤러에서 사용하려는 파라미터를 지원하는지 체크한다. 만약 해당 파라미터를 지원한다면, `resolveArgument()`를 호출하여 그 파라미터의 실제 객체를 생성한다. 이렇게 생성된 파라미터 객체가 컨트롤러가 호출되면 넘어가게 되어, 컨트롤러에서 아무 문제 없이 사용할 수 있게 되는 것이다.

참고: 가능한 파라미터 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-annarguments

⦿ ReturnValueHandler (HandlerMethodReturnValueHandler)

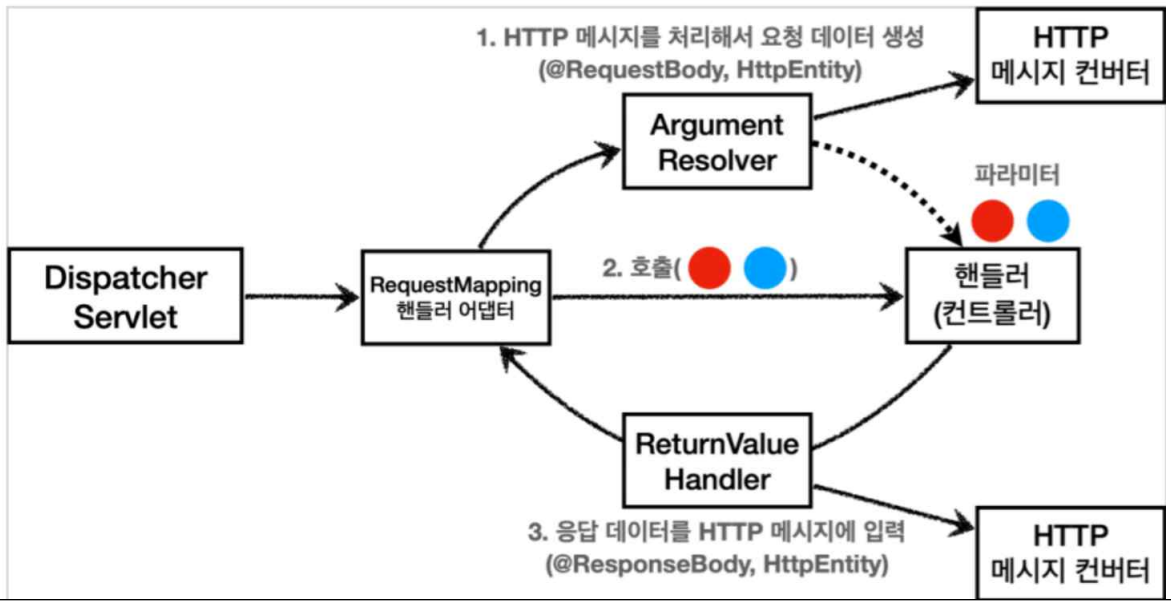
컨트롤러에서 String으로 뷰 이름을 return해도 정상적으로 동작하는 이유가 이 핸들러 덕분이다. 동작 방식은 `**ArgumentResolver**`와 비슷하게, 컨트롤러에서 사용하려는 파라미터가 지원하는지 체크되면 그 파라미터의 실제 객체를 생성하는 방식으로 동작한다.

Spring에서는 10개가 넘는 `**ReturnValueHandler**`를 지원한다. 예를 들어 ModelAndView, @ResponseBody, HttpEntity, String 등이 전부 `**ReturnValueHandler**`라고 할 수 있다.

참고: 가능한 응답 값 목록은 다음 공식 메뉴얼에서 확인할 수 있다.
https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-annreturn-types

⦿ HTTP 메시지 컨버터

그렇다면, HTTP 메시지 컨버터는 Spring MVC 과정 중 어디에서 일어나는 걸까?



요청의 경우 : HTTP 요청 메세지 바디를 처리하는 @RequestBody와 HttpEntity 각각의 ArgumentResolver가 존재한다. 이 ArgumentResolver들이 Http 메세지 컨버터를 사용하여, 컨트롤러에 필요한 객체를 생성하게 된다.

응답의 경우 : HTTP 응답 메세지 바디를 처리하는 @ResponseBody와 HttpEntity 각각의 ReturnValueResolver가 존재한다. ReturnValueResolver가 HTTP 메세지 컨버터를 호출하여 응답 결과를 만든다.