9. API 예외 처리

#2.인강/5. 스프링 MVC 2/강의#

목차

API 예외 처리 - 시작

목표
API 예외 처리는 어떻게 해야할까?

HTML 페이지의 경우 지금까지 설명했던 것 처럼 4xx, 5xx와 같은 오류 페이지만 있으면 대부분의 문제를 해결할 수 있다.
그런데 API의 경우에는 생각할 내용이 더 많다. 오류 페이지는 단순히 고객에게 오류 화면을 보여주고 끝이지만, API는 각 오류 상황에 맞는 오류 응답 스펙을 정하고, JSON으로 데이터를 내려주어야 한다.

지금부터 API의 경우 어떻게 예외 처리를 하면 좋은지 알아보자.
API도 오류 페이지에서 설명했던 것 처럼 처음으로 돌아가서 서블릿 오류 페이지 방식을 사용해보자.

WebServerCustomizer 다시 동작

```
package hello.exception;

import org.springframework.boot.web.server.ConfigurableWebServerFactory;
import org.springframework.boot.web.server.ErrorPage;
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;
```

```
@Component
public class WebServerCustomizer implements
WebServerFactoryCustomizer<ConfigurableWebServerFactory> {
    @Override
    public void customize(ConfigurableWebServerFactory factory) {

        ErrorPage errorPage404 = new ErrorPage(HttpStatus.NOT_FOUND, "/error-
page/404");
        ErrorPage errorPage500 = new
ErrorPage(HttpStatus.INTERNAL_SERVER_ERROR, "/error-page/500");
        ErrorPage errorPageEx = new ErrorPage(RuntimeException.class, "/error-
page/500");
        factory.addErrorPages(errorPage404, errorPage500, errorPageEx);
    }
}
```

`WebServerCustomizer`가 다시 사용되도록 하기 위해 `@Component` 애노테이션에 있는 주석을 풀자
이제 WAS에 예외가 전달되거나, `response.sendError()` 가 호출되면 위에 등록한 예외 페이지 경로가 호출된다.

ApiExceptionController - API 예외 컨트롤러

```
package hello.exception.api;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
public class ApiExceptionController {

    @GetMapping("/api/members/{id}")
    public MemberDto getMember(@PathVariable("id") String id) {
```

```
        if (id.equals("ex")) {
            throw new RuntimeException("잘못된 사용자");
        }

        return new MemberDto(id, "hello " + id);
    }

    @Data
    @AllArgsConstructor
    static class MemberDto {
        private String memberId;
        private String name;
    }

}
```

단순히 회원을 조회하는 기능을 하나 만들었다. 예외 테스트를 위해 URL에 전달된 `id`의 값이 `ex`이면 예외가 발생하도록 코드를 심어두었다.

Postman으로 테스트

HTTP Header에 `Accept`가 `application/json`인 것을 꼭 확인하자.

정상 호출

`http://localhost:8080/api/members/spring`

```
{
    "memberId": "spring",
    "name": "hello spring"
}
```

예외 발생 호출

`http://localhost:8080/api/members/ex`

```
<!DOCTYPE HTML>
<html>
<head>
```

```
</head>
<body>
...
</body>
```

API를 요청했는데, 정상의 경우 API로 JSON 형식으로 데이터가 정상 반환된다. 그런데 오류가 발생하면 우리가 미리 만들어둔 오류 페이지 HTML이 반환된다. 이것은 기대하는 바가 아니다. 클라이언트는 정상 요청이든, 오류 요청이든 JSON이 반환되기를 기대한다. 웹 브라우저가 아닌 이상 HTML을 직접 받아서 할 수 있는 것은 별로 없다.

문제를 해결하려면 오류 페이지 컨트롤러도 JSON 응답을 할 수 있도록 수정해야 한다.

ErrorPageController - API 응답 추가

```
@RequestMapping(value = "/error-page/500", produces =
MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Map<String, Object>> errorPage500Api(HttpServletRequest
request, HttpServletResponse response) {
    log.info("API errorPage 500");

    Map<String, Object> result = new HashMap<>();
    Exception ex = (Exception) request.getAttribute(ERROR_EXCEPTION);
    result.put("status", request.getAttribute(ERROR_STATUS_CODE));
    result.put("message", ex.getMessage());

    Integer statusCode = (Integer)
request.getAttribute(RequestDispatcher.ERROR_STATUS_CODE);
    return new ResponseEntity(result, HttpStatus.valueOf(statusCode));
}
```

`produces = MediaType.APPLICATION_JSON_VALUE`의 뜻은 클라이언트가 요청하는 HTTP Header의 `Accept`의 값이 `application/json`일 때 해당 메서드가 호출된다는 것이다. 결국 클라어인트가 받고 싶은 미디어타입이 json이면 이 컨트롤러의 메서드가 호출된다.

응답 데이터를 위해서 `Map`을 만들고 `status`, `message`키에 값을 할당했다. Jackson 라이브러리는 `Map`을 JSON 구조로 변환할 수 있다.

`ResponseEntity`를 사용해서 응답하기 때문에 메시지 컨버터가 동작하면서 클라이언트에 JSON이 반환된다.

포스트맨을 통해서 다시 테스트 해보자.
HTTP Header에 `Accept`가 `application/json`인 것을 꼭 확인하자.

`http://localhost:8080/api/members/ex`

```
{
    "message": "잘못된 사용자",
    "status": 500
}
```

HTTP Header에 `Accept`가 `application/json`이 아니면, 기존 오류 응답인 HTML 응답이 출력되는 것을 확인할 수 있다.

API 예외 처리 - 스프링 부트 기본 오류 처리

API 예외 처리도 스프링 부트가 제공하는 기본 오류 방식을 사용할 수 있다.
스프링 부트가 제공하는 `BasicErrorController` 코드를 보자.

BasicErrorController 코드

```
@RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
public ModelAndView errorHtml(HttpServletRequest request, HttpServletResponse
response) {}

@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {}
```

`/error` 동일한 경로를 처리하는 `errorHtml()`, `error()` 두 메서드를 확인할 수 있다.

- `errorHtml()` : `produces = MediaType.TEXT_HTML_VALUE` : 클라이언트 요청의 Accept 해더 값이 `text/html`인 경우에는 `errorHtml()`을 호출해서 view를 제공한다.

- `error()` : 그외 경우에 호출되고 `ResponseEntity` 로 HTTP Body에 JSON 데이터를 반환한다.

스프링 부트의 예외 처리

앞서 학습했듯이 스프링 부트의 기본 설정은 오류 발생시 `/error` 를 오류 페이지로 요청한다.
`BasicErrorController` 는 이 경로를 기본으로 받는다. (`server.error.path` 로 수정 가능, 기본 경로 `/error`)

Postman으로 실행

GET http://localhost:8080/api/members/ex

주의

`BasicErrorController` 를 사용하도록 `WebServerCustomizer` 의 `@Component` 를 주석처리 하자.

```
{
    "timestamp": "2021-04-28T00:00:00.000+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "exception": "java.lang.RuntimeException",
    "trace": "java.lang.RuntimeException: 잘못된 사용자\n\tat
hello.exception.web.api.ApiExceptionController.getMember(ApiExceptionController
.java:19...,
    "message": "잘못된 사용자",
    "path": "/api/members/ex"
}
```

스프링 부트는 `BasicErrorController` 가 제공하는 기본 정보들을 활용해서 오류 API를 생성해준다.

다음 옵션들을 설정하면 더 자세한 오류 정보를 추가할 수 있다.

- server.error.include-binding-errors=always
- server.error.include-exception=true
- server.error.include-message=always
- server.error.include-stacktrace=always

물론 오류 메시지는 이렇게 막 추가하면 보안상 위험할 수 있다. 간결한 메시지만 노출하고, 로그를 통해서 확인하자.

Html 페이지 vs API 오류

`BasicErrorController`를 확장하면 JSON 메시지도 변경할 수 있다. 그런데 API 오류는 조금 뒤에 설명할 `@ExceptionHandler`가 제공하는 기능을 사용하는 것이 더 나은 방법이므로 지금은 `BasicErrorController`를 확장해서 JSON 오류 메시지를 변경할 수 있다 정도로만 이해해두자.

스프링 부트가 제공하는 `BasicErrorController`는 HTML 페이지를 제공하는 경우에는 매우 편리하다. 4xx, 5xx 등등 모두 잘 처리해준다. 그런데 API 오류 처리는 다른 차원의 이야기이다. API 마다, 각각의 컨트롤러나 예외마다 서로 다른 응답 결과를 출력해야 할 수도 있다. 예를 들어서 회원과 관련된 API에서 예외가 발생할 때 응답과, 상품과 관련된 API에서 발생하는 예외에 따라 그 결과가 달라질 수 있다.
결과적으로 매우 세밀하고 복잡하다. 따라서 이 방법은 HTML 화면을 처리할 때 사용하고, API 오류 처리는 뒤에서 설명할 `@ExceptionHandler`를 사용하자.

그렇다면 복잡한 API 오류는 어떻게 처리해야하는지 지금부터 하나씩 알아보자.

API 예외 처리 - HandlerExceptionResolver 시작

목표
예외가 발생해서 서블릿을 넘어 WAS까지 예외가 전달되면 HTTP 상태코드가 500으로 처리된다.
발생하는 예외에 따라서 400, 404 등등 다른 상태코드로 처리하고 싶다.
오류 메시지, 형식등을 API마다 다르게 처리하고 싶다.

상태코드 변환
예를 들어서 `IllegalArgumentException`을 처리하지 못해서 컨트롤러 밖으로 넘어가는 일이 발생하면 HTTP 상태코드를 400으로 처리하고 싶다. 어떻게 해야할까?

ApiExceptionController - 수정

```
@GetMapping("/api/members/{id}")
public MemberDto getMember(@PathVariable("id") String id) {

    if (id.equals("ex")) {
        throw new RuntimeException("잘못된 사용자");
    }
```

```
    if (id.equals("bad")) {
        throw new IllegalArgumentException("잘못된 입력 값");
    }

    return new MemberDto(id, "hello " + id);
}
```

http://localhost:8080/api/members/bad 라고 호출하면 `IllegalArgumentException` 이 발생하도록 했다.

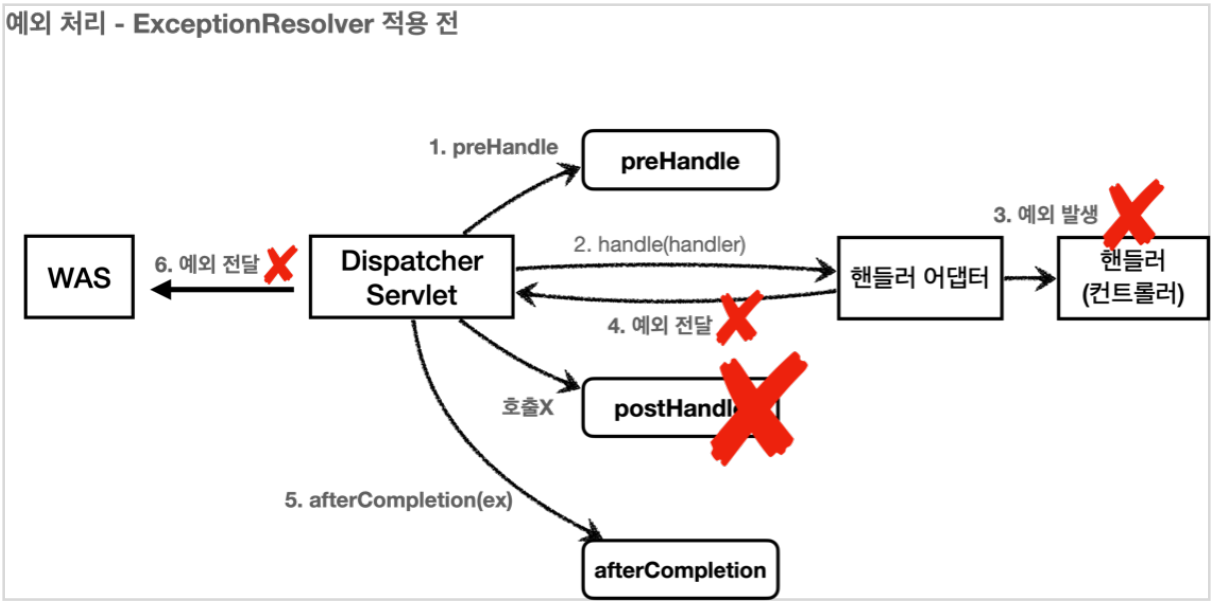
실행해보면 상태 코드가 500인 것을 확인할 수 있다.

```
{
    "status": 500,
    "error": "Internal Server Error",
    "exception": "java.lang.IllegalArgumentException",
    "path": "/api/members/bad"
}
```

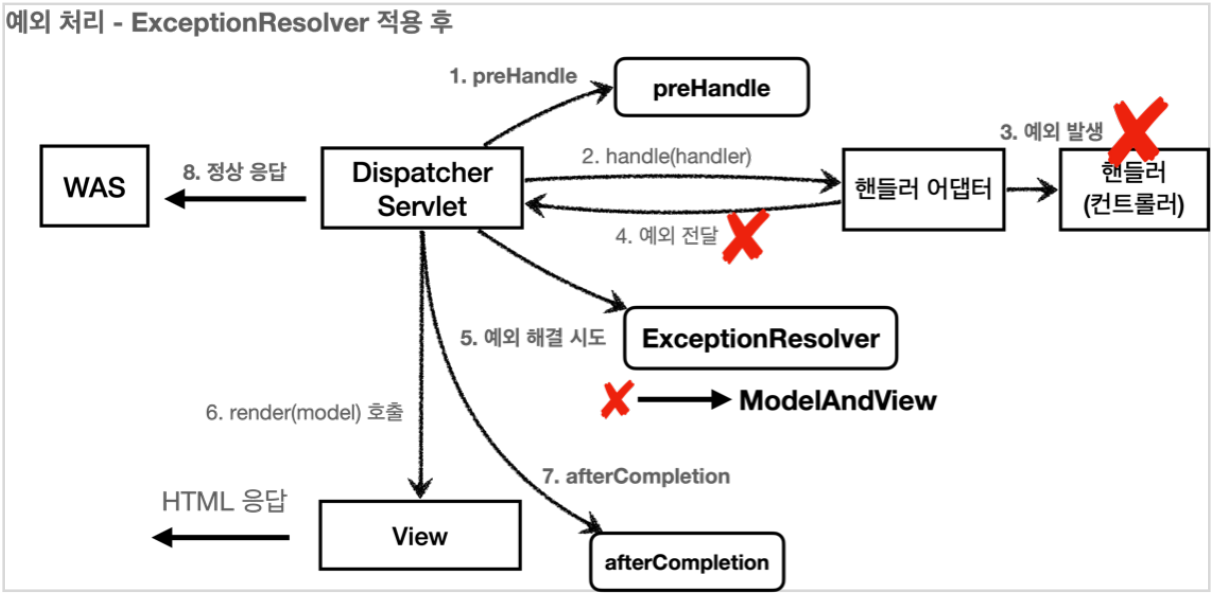
HandlerExceptionResolver

스프링 MVC는 컨트롤러(핸들러) 밖으로 예외가 던져진 경우 예외를 해결하고, 동작을 새로 정의할 수 있는 방법을 제공한다. 컨트롤러 밖으로 던져진 예외를 해결하고, 동작 방식을 변경하고 싶으면 `HandlerExceptionResolver` 를 사용하면 된다. 줄여서 `ExceptionResolver` 라 한다.

ExceptionResolver 적용 전



ExceptionResolver 적용 후



참고: `ExceptionResolver`로 예외를 해결해도 `postHandle()`은 호출되지 않는다.

HandlerExceptionResolver - 인터페이스

```
public interface HandlerExceptionResolver {

  ModelAndView resolveException(
    HttpServletRequest request, HttpServletResponse response,
    Object handler, Exception ex);
}
```

- `handler` : 핸들러(컨트롤러) 정보
- `Exception ex` : 핸들러(컨트롤러)에서 발생한 발생한 예외

MyHandlerExceptionResolver

```
package hello.exception.resolver;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@Slf4j
public class MyHandlerExceptionResolver implements HandlerExceptionResolver {

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {

        try {
            if (ex instanceof IllegalArgumentException) {
                log.info("IllegalArgumentException resolver to 400");
                response.sendError(HttpServletResponse.SC_BAD_REQUEST,
ex.getMessage());
                return new ModelAndView();
            }
        } catch (IOException e) {
            log.error("resolver ex", e);
        }

        return null;
    }
}
```

- `ExceptionResolver`가 `ModelAndView`를 반환하는 이유는 마치 try, catch를 하듯이, `Exception`을 처리해서 정상 흐름 처럼 변경하는 것이 목적이다. 이름 그대로 `Exception`을 Resolver(해결)하는 것이 목적이다.

  여기서는 `IllegalArgumentException`이 발생하면 `response.sendError(400)`를 호출해서 HTTP 상태 코드를 400으로 지정하고, 빈 `ModelAndView`를 반환한다.

  반환 값에 따른 동작 방식

  `HandlerExceptionResolver`의 반환 값에 따른 `DispatcherServlet`의 동작 방식은 다음과 같다.

- **빈 ModelAndView**: `new ModelAndView()` 처럼 빈 `ModelAndView`를 반환하면 뷰를 렌더링 하지 않고, 정상 흐름으로 서블릿이 리턴된다.
- **ModelAndView 지정**: `ModelAndView`에 `View`, `Model` 등의 정보를 지정해서 반환하면 뷰를 렌더링 한다.
- **null**: `null`을 반환하면, 다음 `ExceptionResolver`를 찾아서 실행한다. 만약 처리할 수 있는 `ExceptionResolver`가 없으면 예외 처리가 안되고, 기존에 발생한 예외를 서블릿 밖으로 던진다.

  ExceptionResolver 활용

- 예외 상태 코드 변환
  - 예외를 `response.sendError(xxx)` 호출로 변경해서 서블릿에서 상태 코드에 따른 오류를 처리하도록 위임
  - 이후 WAS는 서블릿 오류 페이지를 찾아서 내부 호출, 예를 들어서 스프링 부트가 기본으로 설정한 `/error`가 호출됨
- 뷰 템플릿 처리
  - `ModelAndView`에 값을 채워서 예외에 따른 새로운 오류 화면 뷰 렌더링 해서 고객에게 제공
- API 응답 처리
  - `response.getWriter().println("hello");` 처럼 HTTP 응답 바디에 직접 데이터를 넣어주는 것도 가능하다. 여기에 JSON 으로 응답하면 API 응답 처리를 할 수 있다.

  WebConfig - 수정

  `WebMvcConfigurer`를 통해 등록

```
/**
 * 기본 설정을 유지하면서 추가
 */
@Override
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
```

```
        resolvers.add(new MyHandlerExceptionResolver());
    }
```

`configureHandlerExceptionResolvers(..)`를 사용하면 스프링이 기본으로 등록하는 `ExceptionResolver`가 제거되므로 주의, `extendHandlerExceptionResolvers`를 사용하자.

Postman으로 실행

- http://localhost:8080/api/members/ex ➡ HTTP 상태 코드 500
- http://localhost:8080/api/members/bad ➡ HTTP 상태 코드 400

API 예외 처리 - HandlerExceptionResolver 활용

예외를 여기서 마무리하기

예외가 발생하면 WAS까지 예외가 던져지고, WAS에서 오류 페이지 정보를 찾아서 다시 `/error`를 호출하는 과정은 생각해보면 너무 복잡하다. `ExceptionResolver`를 활용하면 예외가 발생했을 때 이런 복잡한 과정 없이 여기에서 문제를 깔끔하게 해결할 수 있다.

예제로 알아보자.

먼저 사용자 정의 예외를 하나 추가하자.

UserException

```
package hello.exception.exception;

public class UserException extends RuntimeException {

    public UserException() {
        super();
    }

    public UserException(String message) {
        super(message);
    }

    public UserException(String message, Throwable cause) {
        super(message, cause);
```

```
    }

    public UserException(Throwable cause) {
        super(cause);
    }

    protected UserException(String message, Throwable cause, boolean
enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

ApiExceptionController - 예외 추가

```
package hello.exception.api;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@Slf4j
@RestController
public class ApiExceptionController {

    @GetMapping("/api/members/{id}")
    public MemberDto getMember(@PathVariable("id") String id) {

        if (id.equals("ex")) {
            throw new RuntimeException("잘못된 사용자");
        }
        if (id.equals("bad")) {
            throw new IllegalArgumentException("잘못된 입력 값");
        }
```

```
        if (id.equals("user-ex")) {
            throw new UserException("사용자 오류");
        }

        return new MemberDto(id, "hello " + id);
    }

    @Data
    @AllArgsConstructor
    static class MemberDto {
        private String memberId;
        private String name;
    }

}
```

http://localhost:8080/api/members/user-ex 호출시 `UserException`이 발생하도록 해두었다.

이제 이 예외를 처리하는 `UserHandlerExceptionResolver`를 만들어보자.

UserHandlerExceptionResolver

```
package hello.exception.resolver;

import com.fasterxml.jackson.databind.ObjectMapper;
import hello.exception.exception.UserException;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.servlet.HandlerExceptionResolver;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

@Slf4j
public class UserHandlerExceptionResolver implements HandlerExceptionResolver {
```

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

    @Override
    public ModelAndView resolveException(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {

        try {
            if (ex instanceof UserException) {
                log.info("UserException resolver to 400");
                String acceptHeader = request.getHeader("accept");
                response.setStatus(HttpServletResponse.SC_BAD_REQUEST);

                if ("application/json".equals(acceptHeader)) {
                    Map<String, Object> errorResult = new HashMap<>();
                    errorResult.put("ex", ex.getClass());
                    errorResult.put("message", ex.getMessage());
                    String result =
objectMapper.writeValueAsString(errorResult);

                    response.setContentType("application/json");
                    response.setCharacterEncoding("utf-8");
                    response.getWriter().write(result);
                    return new ModelAndView();
                } else {
                    //TEXT/HTML
                    return new ModelAndView("error/500");
                }
            }
        } catch (IOException e) {
            log.error("resolver ex", e);
        }

        return null;
    }
}
```

HTTP 요청 해더의 `ACCEPT` 값이 `application/json` 이면 JSON으로 오류를 내려주고, 그 외 경우에는 error/500에 있는 HTML 오류 페이지를 보여준다.

WebConfig에 UserHandlerExceptionResolver 추가

```
@Override
public void extendHandlerExceptionResolvers(List<HandlerExceptionResolver>
resolvers) {
    resolvers.add(new MyHandlerExceptionResolver());
    resolvers.add(new UserHandlerExceptionResolver());
}
```

실행

POSTMAN 실행,

```
http://localhost:8080/api/members/user-ex
```

`ACCEPT` : `application/json`

```
{
    "ex": "hello.exception.exception.UserException",
    "message": "사용자 오류"
}
```

`ACCEPT` : `text/html`

```
<!DOCTYPE HTML>
<html>
...
</html>
```

정리

`ExceptionResolver` 를 사용하면 컨트롤러에서 예외가 발생해도 `ExceptionResolver` 에서 예외를 처리해버린다.

따라서 예외가 발생해도 서블릿 컨테이너까지 예외가 전달되지 않고, 스프링 MVC에서 예외 처리는 끝이

난다.
결과적으로 WAS 입장에서는 정상 처리가 된 것이다. 이렇게 예외를 이곳에서 모두 처리할 수 있다는 것이 핵심이다.

서블릿 컨테이너까지 예외가 올라가면 복잡하고 지저분하게 추가 프로세스가 실행된다. 반면에 `ExceptionResolver`를 사용하면 예외처리가 상당히 깔끔해진다.

그런데 직접 `ExceptionResolver`를 구현하려고 하니 상당히 복잡하다. 지금부터 스프링이 제공하는 `ExceptionResolver`들을 알아보자.

API 예외 처리 - 스프링이 제공하는 ExceptionResolver1

스프링 부트가 기본으로 제공하는 `ExceptionResolver`는 다음과 같다.
`HandlerExceptionResolverComposite`에 다음 순서로 등록

1. `ExceptionHandlerExceptionResolver`
2. `ResponseStatusExceptionResolver`
3. `DefaultHandlerExceptionResolver` ➜ 우선 순위가 가장 낮다.

ExceptionHandlerExceptionResolver
`@ExceptionHandler`을 처리한다. API 예외 처리는 대부분 이 기능으로 해결한다. 조금 뒤에 자세히 설명한다.

ResponseStatusExceptionResolver
HTTP 상태 코드를 지정해준다.
예) `@ResponseStatus(value = HttpStatus.NOT_FOUND)`

DefaultHandlerExceptionResolver
스프링 내부 기본 예외를 처리한다.

먼저 가장 쉬운 `ResponseStatusExceptionResolver` 부터 알아보자.

ResponseStatusExceptionResolver

`ResponseStatusExceptionResolver`는 예외에 따라서 HTTP 상태 코드를 지정해주는 역할을 한다.

다음 두 가지 경우를 처리한다.

- `@ResponseStatus`가 달려있는 예외
- `ResponseStatusException` 예외

하나씩 확인해보자.

예외에 다음과 같이 `@ResponseStatus` 애노테이션을 적용하면 HTTP 상태 코드를 변경해준다.

```
package hello.exception.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "잘못된 요청 오류")
public class BadRequestException extends RuntimeException {
}
```

`BadRequestException` 예외가 컨트롤러 밖으로 넘어가면 `ResponseStatusExceptionResolver` 예외가 해당 애노테이션을 확인해서 오류 코드를 `HttpStatus.BAD_REQUEST`(400)으로 변경하고, 메시지도 담는다.

`ResponseStatusExceptionResolver` 코드를 확인해보면 결국 `response.sendError(statusCode, resolvedReason)`를 호출하는 것을 확인할 수 있다.
`sendError(400)`를 호출했기 때문에 WAS에서 다시 오류 페이지(`/error`)를 내부 요청한다.

ApiExceptionController - 추가

```
@GetMapping("/api/response-status-ex1")
public String responseStatusEx1() {
    throw new BadRequestException();
}
```

실행
http://localhost:8080/api/response-status-ex1?message=

```
{
    "status": 400,
```

```
    "error": "Bad Request",
    "exception": "hello.exception.exception.BadRequestException",
    "message": "잘못된 요청 오류",
    "path": "/api/response-status-ex1"
}
```

메시지 기능

`reason`을 `MessageSource`에서 찾는 기능도 제공한다. `reason = "error.bad"`

```
package hello.exception.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

//@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "잘못된 요청 오류")
@ResponseStatus(code = HttpStatus.BAD_REQUEST, reason = "error.bad")
public class BadRequestException extends RuntimeException {
}
```

messages.properties

```
error.bad=잘못된 요청 오류입니다. 메시지 사용
```

메시지 사용 결과

```
{
    "status": 400,
    "error": "Bad Request",
    "exception": "hello.exception.exception.BadRequestException",
    "message": "잘못된 요청 오류입니다. 메시지 사용",
    "path": "/api/response-status-ex1"
}
```

ResponseStatusException

`@ResponseStatus`는 개발자가 직접 변경할 수 없는 예외에는 적용할 수 없다. (애노테이션을 직접 넣어야 하는데, 내가 코드를 수정할 수 없는 라이브러리의 예외 코드 같은 곳에는 적용할 수 없다.)
추가로 애노테이션을 사용하기 때문에 조건에 따라 동적으로 변경하는 것도 어렵다. 이때는 `ResponseStatusException` 예외를 사용하면 된다.

ApiExceptionController - 추가

```
@GetMapping("/api/response-status-ex2")
public String responseStatusEx2() {
    throw new ResponseStatusException(HttpStatus.NOT_FOUND, "error.bad", new
IllegalArgumentException());
}
```

http://localhost:8080/api/response-status-ex2

```
{
    "status": 404,
    "error": "Not Found",
    "exception": "org.springframework.web.server.ResponseStatusException",
    "message": "잘못된 요청 오류입니다. 메시지 사용",
    "path": "/api/response-status-ex2"
}
```

API 예외 처리 - 스프링이 제공하는 ExceptionResolver2

이번에는 `DefaultHandlerExceptionResolver`를 살펴보자.

`DefaultHandlerExceptionResolver`는 스프링 내부에서 발생하는 스프링 예외를 해결한다.
대표적으로 파라미터 바인딩 시점에 타입이 맞지 않으면 내부에서 `TypeMismatchException`이 발생하는데, 이 경우 예외가 발생했기 때문에 그냥 두면 서블릿 컨테이너까지 오류가 올라가고, 결과적으로 500 오류가 발생한다.
그런데 파라미터 바인딩은 대부분 클라이언트가 HTTP 요청 정보를 잘못 호출해서 발생하는 문제이다.
HTTP 에서는 이런 경우 HTTP 상태 코드 400을 사용하도록 되어 있다.
`DefaultHandlerExceptionResolver`는 이것을 500 오류가 아니라 HTTP 상태 코드 400 오류로

변경한다.
스프링 내부 오류를 어떻게 처리할지 수 많은 내용이 정의되어 있다.

코드 확인

`DefaultHandlerExceptionResolver.handleTypeMismatch`를 보면 다음과 같은 코드를 확인할 수 있다.
`response.sendError(HttpServletResponse.SC_BAD_REQUEST)` (400)
결국 `response.sendError()`를 통해서 문제를 해결한다.

`sendError(400)`를 호출했기 때문에 WAS에서 다시 오류 페이지(`/error`)를 내부 요청한다.

ApiExceptionController - 추가

```
@GetMapping("/api/default-handler-ex")
public String defaultException(@RequestParam Integer data) {
    return "ok";
}
```

`Integer data`에 문자를 입력하면 내부에서 `TypeMismatchException`이 발생한다.

실행
http://localhost:8080/api/default-handler-ex?data=hello&message=

```
{
    "status": 400,
    "error": "Bad Request",
    "exception":
"org.springframework.web.method.annotation.MethodArgumentTypeMismatchException"
,
    "message": "Failed to convert value of type 'java.lang.String' to required
type 'java.lang.Integer'; nested exception is java.lang.NumberFormatException:
For input string: \"hello\"",
    "path": "/api/default-handler-ex"
}
```

실행 결과를 보면 HTTP 상태 코드가 400인 것을 확인할 수 있다.

정리

지금까지 다음 `ExceptionResolver` 들에 대해 알아보았다.

1. `ExceptionHandlerExceptionResolver` ➜ 다음 시간에
2. `ResponseStatusExceptionResolver` ➜ HTTP 응답 코드 변경
3. `DefaultHandlerExceptionResolver` ➜ 스프링 내부 예외 처리

지금까지 HTTP 상태 코드를 변경하고, 스프링 내부 예외의 상태코드를 변경하는 기능도 알아보았다.
그런데 `HandlerExceptionResolver` 를 직접 사용하기는 복잡하다. API 오류 응답의 경우 `response` 에 직접 데이터를 넣어야 해서 매우 불편하고 번거롭다. `ModelAndView` 를 반환해야 하는 것도 API에는 잘 맞지 않는다.
스프링은 이 문제를 해결하기 위해 `@ExceptionHandler` 라는 매우 혁신적인 예외 처리 기능을 제공한다.
그것이 아직 소개하지 않은 `ExceptionHandlerExceptionResolver` 이다.

API 예외 처리 - @ExceptionHandler

HTML 화면 오류 vs API 오류

웹 브라우저에 HTML 화면을 제공할 때는 오류가 발생하면 `BasicErrorController` 를 사용하는게 편하다.
이때는 단순히 5xx, 4xx 관련된 오류 화면을 보여주면 된다. `BasicErrorController` 는 이런 메커니즘을 모두 구현해두었다.

그런데 API는 각 시스템 마다 응답의 모양도 다르고, 스펙도 모두 다르다. 예외 상황에 단순히 오류 화면을 보여주는 것이 아니라, 예외에 따라서 각각 다른 데이터를 출력해야 할 수도 있다. 그리고 같은 예외라고 해도 어떤 컨트롤러에서 발생했는가에 따라서 다른 예외 응답을 내려주어야 할 수 있다. 한마디로 매우 세밀한 제어가 필요하다.
앞서 이야기했지만, 예를 들어서 상품 API와 주문 API는 오류가 발생했을 때 응답의 모양이 완전히 다를 수 있다.

결국 지금까지 살펴본 `BasicErrorController` 를 사용하거나 `HandlerExceptionResolver` 를 직접 구현하는 방식으로 API 예외를 다루기는 쉽지 않다.

API 예외처리의 어려운 점

- `HandlerExceptionResolver` 를 떠올려 보면 `ModelAndView` 를 반환해야 했다. 이것은 API 응답에는 필요하지 않다.
- API 응답을 위해서 `HttpServletResponse` 에 직접 응답 데이터를 넣어주었다. 이것은 매우 불편하다. 스프링 컨트롤러에 비유하면 마치 과거 서블릿을 사용하던 시절로 돌아간 것 같다.

- 특정 컨트롤러에서만 발생하는 예외를 별도로 처리하기 어렵다. 예를 들어서 회원을 처리하는 컨트롤러에서 발생하는 `RuntimeException` 예외와 상품을 관리하는 컨트롤러에서 발생하는 동일한 `RuntimeException` 예외를 서로 다른 방식으로 처리하고 싶다면 어떻게 해야할까?

@ExceptionHandler

스프링은 API 예외 처리 문제를 해결하기 위해 `@ExceptionHandler` 라는 애노테이션을 사용하는 매우 편리한 예외 처리 기능을 제공하는데, 이것이 바로 `ExceptionHandlerExceptionResolver` 이다.
스프링은 `ExceptionHandlerExceptionResolver` 를 기본으로 제공하고, 기본으로 제공하는 `ExceptionResolver` 중에 우선순위도 가장 높다. 실무에서 API 예외 처리는 대부분 이 기능을 사용한다.

먼저 예제로 알아보자.

ErrorResult

```
package hello.exception.exhandler;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class ErrorResult {
    private String code;
    private String message;
}
```

예외가 발생했을 때 API 응답으로 사용하는 객체를 정의했다.

ApiExceptionV2Controller

```
package hello.exception.exhandler;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
```

```java
@Slf4j
@RestController
public class ApiExceptionV2Controller {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(IllegalArgumentException.class)
    public ErrorResult illegalExHandle(IllegalArgumentException e) {
        log.error("[exceptionHandle] ex", e);
        return new ErrorResult("BAD", e.getMessage());
    }

    @ExceptionHandler
    public ResponseEntity<ErrorResult> userExHandle(UserException e) {
        log.error("[exceptionHandle] ex", e);
        ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
        return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
    }

    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ExceptionHandler
    public ErrorResult exHandle(Exception e) {
        log.error("[exceptionHandle] ex", e);
        return new ErrorResult("EX", "내부 오류");
    }

    @GetMapping("/api2/members/{id}")
    public MemberDto getMember(@PathVariable("id") String id) {

        if (id.equals("ex")) {
            throw new RuntimeException("잘못된 사용자");
        }
        if (id.equals("bad")) {
            throw new IllegalArgumentException("잘못된 입력 값");
        }
        if (id.equals("user-ex")) {
            throw new UserException("사용자 오류");
        }
```

```
        return new MemberDto(id, "hello " + id);
    }

    @Data
    @AllArgsConstructor
    static class MemberDto {
        private String memberId;
        private String name;
    }

}
```

@ExceptionHandler 예외 처리 방법

`@ExceptionHandler` 애노테이션을 선언하고, 해당 컨트롤러에서 처리하고 싶은 예외를 지정해주면 된다. 해당 컨트롤러에서 예외가 발생하면 이 메서드가 호출된다. 참고로 지정한 예외 또는 그 예외의 자식 클래스는 모두 잡을 수 있다.

다음 예제는 `IllegalArgumentException` 또는 그 하위 자식 클래스를 모두 처리할 수 있다.

```
@ExceptionHandler(IllegalArgumentException.class)
public ErrorResult illegalExHandle(IllegalArgumentException e) {
    log.error("[exceptionHandle] ex", e);
    return new ErrorResult("BAD", e.getMessage());
}
```

우선순위

스프링의 우선순위는 항상 자세한 것이 우선권을 가진다. 예를 들어서 부모, 자식 클래스가 있고 다음과 같이 예외가 처리된다.

```
@ExceptionHandler(부모예외.class)
public String 부모예외처리()(부모예외 e) {}

@ExceptionHandler(자식예외.class)
public String 자식예외처리()(자식예외 e) {}
```

`@ExceptionHandler`에 지정한 부모 클래스는 자식 클래스까지 처리할 수 있다. 따라서 `자식예외`가

발생하면 `부모예외처리()`, `자식예외처리()` 둘다 호출 대상이 된다. 그런데 둘 중 더 자세한 것이 우선권을 가지므로 `자식예외처리()` 가 호출된다. 물론 `부모예외` 가 호출되면 `부모예외처리()` 만 호출 대상이 되므로 `부모예외처리()` 가 호출된다.

다양한 예외

다음과 같이 다양한 예외를 한번에 처리할 수 있다.

```
@ExceptionHandler({AException.class, BException.class})
public String ex(Exception e) {
    log.info("exception e", e);
}
```

예외 생략

`@ExceptionHandler` 에 예외를 생략할 수 있다. 생략하면 메서드 파라미터의 예외가 지정된다.

```
@ExceptionHandler
public ResponseEntity<ErrorResult> userExHandle(UserException e) {}
```

파리미터와 응답

`@ExceptionHandler` 에는 마치 스프링의 컨트롤러의 파라미터 응답처럼 다양한 파라미터와 응답을 지정할 수 있다.

자세한 파라미터와 응답은 다음 공식 메뉴얼을 참고하자.

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-exceptionhandler-args

Postman 실행

http://localhost:8080/api2/members/bad

IllegalArgumentException 처리

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(IllegalArgumentException.class)
public ErrorResult illegalExHandle(IllegalArgumentException e) {
    log.error("[exceptionHandle] ex", e);
```

```
    return new ErrorResult("BAD", e.getMessage());
}
```

실행 흐름

- 컨트롤러를 호출한 결과 `IllegalArgumentException` 예외가 컨트롤러 밖으로 던져진다.
- 예외가 발생했으로 `ExceptionResolver` 가 작동한다. 가장 우선순위가 높은 `ExceptionHandlerExceptionResolver` 가 실행된다.
- `ExceptionHandlerExceptionResolver` 는 해당 컨트롤러에 `IllegalArgumentException` 을 처리할 수 있는 `@ExceptionHandler` 가 있는지 확인한다.
- `illegalExHandle()` 를 실행한다. `@RestController` 이므로 `illegalExHandle()` 에도 `@ResponseBody` 가 적용된다. 따라서 HTTP 컨버터가 사용되고, 응답이 다음과 같은 JSON으로 반환된다.
- `@ResponseStatus(HttpStatus.BAD_REQUEST)` 를 지정했으므로 HTTP 상태 코드 400으로 응답한다.

결과

```
{
    "code": "BAD",
    "message": "잘못된 입력 값"
}
```

Postman 실행

http://localhost:8080/api2/members/user-ex

UserException 처리

```
@ExceptionHandler
public ResponseEntity<ErrorResult> userExHandle(UserException e) {
    log.error("[exceptionHandle] ex", e);
    ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
    return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
}
```

- `@ExceptionHandler` 에 예외를 지정하지 않으면 해당 메서드 파라미터 예외를 사용한다. 여기서는 `UserException` 을 사용한다.
- `ResponseEntity` 를 사용해서 HTTP 메시지 바디에 직접 응답한다. 물론 HTTP 컨버터가 사용된다.

`ResponseEntity`를 사용하면 HTTP 응답 코드를 프로그래밍해서 동적으로 변경할 수 있다. 앞서 살펴본 `@ResponseStatus`는 애노테이션이므로 HTTP 응답 코드를 동적으로 변경할 수 없다.

Postman 실행

- http://localhost:8080/api2/members/ex

Exception

```
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
@ExceptionHandler
public ErrorResult exHandle(Exception e) {
    log.error("[exceptionHandle] ex", e);
    return new ErrorResult("EX", "내부 오류");
}
```

- `throw new RuntimeException("잘못된 사용자")` 이 코드가 실행되면서, 컨트롤러 밖으로 `RuntimeException`이 던져진다.
- `RuntimeException`은 `Exception`의 자식 클래스이다. 따라서 이 메서드가 호출된다.
- `@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)`로 HTTP 상태 코드를 500으로 응답한다.

기타

HTML 오류 화면

다음과 같이 `ModelAndView`를 사용해서 오류 화면(HTML)을 응답하는데 사용할 수도 있다.

```
@ExceptionHandler(ViewException.class)
public ModelAndView ex(ViewException e) {
    log.info("exception e", e);
    return new ModelAndView("error");
}
```

API 예외 처리 - @ControllerAdvice

`@ExceptionHandler`를 사용해서 예외를 깔끔하게 처리할 수 있게 되었지만, 정상 코드와 예외 처리 코드가 하나의 컨트롤러에 섞여 있다. `@ControllerAdvice` 또는 `@RestControllerAdvice`를 사용하면 둘을 분리할 수 있다.

ExControllerAdvice

```
package hello.exception.exhandler.advice;

import hello.exception.exception.UserException;
import hello.exception.exhandler.ErrorResult;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

@Slf4j
@RestControllerAdvice
public class ExControllerAdvice {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(IllegalArgumentException.class)
    public ErrorResult illegalExHandle(IllegalArgumentException e) {
        log.error("[exceptionHandle] ex", e);
        return new ErrorResult("BAD", e.getMessage());
    }

    @ExceptionHandler
    public ResponseEntity<ErrorResult> userExHandle(UserException e) {
        log.error("[exceptionHandle] ex", e);
        ErrorResult errorResult = new ErrorResult("USER-EX", e.getMessage());
        return new ResponseEntity<>(errorResult, HttpStatus.BAD_REQUEST);
    }

    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    @ExceptionHandler
    public ErrorResult exHandle(Exception e) {
        log.error("[exceptionHandle] ex", e);
```

```
        return new ErrorResult("EX", "내부 오류");
    }

}
```

ApiExceptionV2Controller 코드에 있는 @ExceptionHandler 모두 제거

```
package hello.exception.exhandler;

import hello.exception.exception.UserException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import org.springframework.web.bind.annotation.*;

@Slf4j
@RestController
public class ApiExceptionV2Controller {

    @GetMapping("/api2/members/{id}")
    public MemberDto getMember(@PathVariable("id") String id) {

        if (id.equals("ex")) {
            throw new RuntimeException("잘못된 사용자");
        }
        if (id.equals("bad")) {
            throw new IllegalArgumentException("잘못된 입력 값");
        }
        if (id.equals("user-ex")) {
            throw new UserException("사용자 오류");
        }

        return new MemberDto(id, "hello " + id);
    }

    @Data
    @AllArgsConstructor
    static class MemberDto {
```

```
        private String memberId;
        private String name;
    }

}
```

Postman 실행

- http://localhost:8080/api2/members/bad
- http://localhost:8080/api2/members/user-ex
- http://localhost:8080/api2/members/ex

@ControllerAdvice

- `@ControllerAdvice`는 대상으로 지정한 여러 컨트롤러에 `@ExceptionHandler`, `@InitBinder` 기능을 부여해주는 역할을 한다.
- `@ControllerAdvice`에 대상을 지정하지 않으면 모든 컨트롤러에 적용된다. (글로벌 적용)
- `@RestControllerAdvice`는 `@ControllerAdvice`와 같고, `@ResponseBody`가 추가되어 있다. `@Controller`, `@RestController`의 차이와 같다.

대상 컨트롤러 지정 방법

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class ExampleAdvice1 {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class ExampleAdvice2 {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class,
AbstractController.class})
public class ExampleAdvice3 {}
```

https://docs.spring.io/spring-framework/docs/current/reference/html/web.html#mvc-ann-controller-advice (스프링 공식 문서 참고)

스프링 공식 문서 예제에서 보는 것 처럼 특정 애노테이션이 있는 컨트롤러를 지정할 수 있고, 특정 패키지를 직접 지정할 수도 있다. 패키지 지정의 경우 해당 패키지와 그 하위에 있는 컨트롤러가 대상이 된다. 그리고

특정 클래스를 지정할 수도 있다.
대상 컨트롤러 지정을 생략하면 모든 컨트롤러에 적용된다.

정리
`@ExceptionHandler`와 `@ControllerAdvice`를 조합하면 예외를 깔끔하게 해결할 수 있다.

정리