에러와 예외 처리

주차	34주차
스터디 일자	@2024/07/12

프로그램 오류

- 프로그램이 실행 중 어떤 원인에 의해서 오작동을 하거나 비정상적으로 종료되는 경우
- 프로그램 오류에는 **에러와, 예외** 두가지로 구분할 수 있다.

에러

- 에러 (Error) 는 컴파일 시 문법적인 오류와 런타임 시 널포인트 참조 와 같은 오류
- 에러 (Error) 는 프로세스에 심각한 문제를 야기시켜 프로세스를 종료 시킬 수 있다.
- 비교적 심각한 오류

예외

- 예외 (Exception) 는 컴퓨터 시스템의 동작 도중 예기치 않았던 이상 상태가 발생하여 수행 중인 프로그램이 영향을 받는 것
- 연산 도중 넘침에 의한 끼어들기 등이 이에 해당된다.
- 비교적 덜 심각한 오류
- 프로그래머가 적절히 코드를 작성해주면 비정상적인 종료를 막을 수 있다.

💡 Error의 상황을 미리 미연에 방지하기 위해서 Exception 상황을 만들 수 있으며, java에서는 try-catch문으로 Exception handling을 할 수 있다.

Spring의 예외 처리 방법

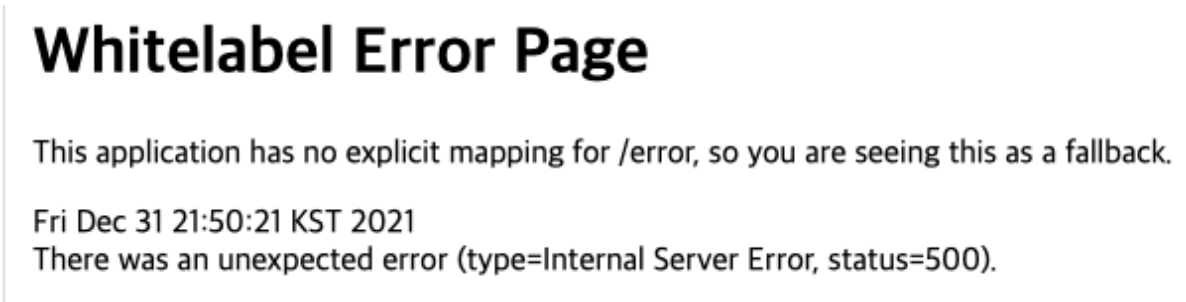
기본적인 예외 처리 방식 (SpringBoot)

```
@RestController
@RequiredArgsConstructor
public class ProductController {

    private final ProductService productService;

    @GetMapping("/product/{id}")
    public Response getProduct(@PathVariable String id){
        return productService.getProduct(id);
    }
}
```

- 해당 메서드에서 NoSuchElementFoundException 예외가 발생했다면 아래와 같은 에러 페이지를 반환받는다.



Spring은 만들어질 때부터 **에러 처리를 위한 BasicErrorController**를 구현해두었고, 스프링 부트는 예외가 발생하면 기본적으로 /error로 에러 요청을 다시 전달하도록 WAS 설정을 해두었다.

그래서 별도의 설정이 없다면 예외 발생 시에 아래의 과정을 거쳐서 BasicErrorController로 에러 처리 요청이 전달된다.

```
WAS(톰캣) -> 필터 -> 서블릿(디스패처 서블릿) -> 인터셉터 -> 컨트롤러
-> 컨트롤러(예외발생) -> 인터셉터 -> 서블릿(디스패처 서블릿) -> 필터 -> WA
-> WAS(톰캣) -> 필터 -> 서블릿(디스패처 서블릿) -> 인터셉터 -> 컨트롤러(B
```

BasicErrorController

- BasicErrorController는 accept 헤더에 따라 에러 페이지를 반환하거나 에러 메세지를 반환한다.

- 에러 경로는 기본적으로 /error로 정의되어 있으며, properties에서 server.error.path로 변경할 수 있다.

```
@Controller
@RequestMapping("${server.error.path:${error.path:/error}}")
public class BasicErrorController extends AbstractErrorControlle

    private final ErrorProperties errorProperties;
    ...

    @RequestMapping(produces = MediaType.TEXT_HTML_VALUE)
```

```
    public ModelAndView errorHtml(HttpServletRequest request, Ht
        ...
    }

    @RequestMapping
    public ResponseEntity<Map<String, Object>> error(HttpServlet
        ...
        return new ResponseEntity<>(body, status);
    }

    ...
}
```

- errorHtml()과 error()는 모두 getErrorAttributeOptions를 호출해 반환할 에러 속성을 얻는데, 기본적으로 **DefaultErrorAttributes로부터 반환할 정보를 가져온다.**
- DefaultErrorAttributes는 전체 항목들에서 설정에 맞게 불필요한 속성들을 제거한다.
  - timestamp: 에러가 발생한 시간
  - status: 에러의 Http 상태
  - error: 에러 코드
  - path: 에러가 발생한 URI
  - exception: 최상위 예외 클래스의 이름(설정 필요)
  - message: 에러에 대한 내용(설정 필요)
  - errors: BindingExecption에 의해 생긴 에러 목록(설정 필요)
  - trace: 에러 스택 트레이스(설정 필요)

- 기본 설정으로 받는 에러 응답

```
{
    "timestamp": "2021-12-31T03:35:44.675+00:00",
    "status": 500,
```

```
    "error": "Internal Server Error",
    "path": "/product/5000"
}
```

스프링이 제공하는 다양한 예외처리 방법

@ResponseStatus

- 에러 HTTP 상태를 변경하도록 도와주는 어노테이션

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class NoSuchElementFoundException extends RuntimeExceptio
  ...
}
```

- 이렇게 응답 상태를 지정해줄 수 있다.

- 결과

```
{
    "timestamp": "2021-12-31T03:35:44.675+00:00",
    "status": 404,
    "error": "Not Found",
    "path": "/product/5000"
}
```

한계점

- 예와 클래스와 강하게 결합되어 **같은 예외는 같은 상태와 에러 메세지를 반환함**
- 별도의 응답 상태가 필요하다면 예외 클래스를 추가해야 됨
- 외부에서 정의한 Exception 클래스에는 @ResponseStatus를 붙여줄 수 없음

@ResponseStatusException

- HttpStatus와 함께 선택적으로 이유와 원인을 추가할 수 있고, 언체크 예외을 상속받고 있어 명시적으로 에러를 처리해주지 않아도 된다.

```
@GetMapping("/product/{id}")
public ResponseEntity<Product> getProduct(@PathVariable String i
    try {
        return ResponseEntity.ok(productService.getProduct(id))
    } catch (NoSuchElementFoundException e) {
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,
    }
}
```

장점

- 기본적인 예외 처리를 빠르게 적용할 수 있으므로 손쉽게 프로토타이핑할 수 있음
- HttpStatus를 직접 설정하여 예외 클래스와의 결합도를 낮출 수 있음
- 불필요하게 많은 별도의 예외 클래스를 만들지 않아도 됨

한계점

- 직접 예외 처리를 프로그래밍하므로 **일관된 예외 처리가 어려움**
- 예외 처리 코드가 **중복될 수 있음**

@ExceptionHandler

- 매우 유연하게 에러를 처리할 수 있는 방법을 제공하는 기능

```
@RestController
@RequiredArgsConstructor
public class ProductController {

  private final ProductService productService;

  @GetMapping("/product/{id}")
  public Response getProduct(@PathVariable String id){
    return productService.getProduct(id);
  }

  @ExceptionHandler(NoSuchElementFoundException.class)
  public ResponseEntity<String> handleNoSuchElementFoundExceptio
    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(exce
  }
}
```

- @ExceptionHandler를 사용 시에 주의할 점은 @ExceptionHandler에 등록된 예외 클래스와 파라미터로 받는 예와 클래스가 동일해야 한다는 것이다.
  - 만약 값이 다르다면 스프링은 컴파일 시점에 에러를 내지 않다가 런타임 시점에 에러를 발생시킨다.

→ @ExceptionHandler는 컨트롤러에 구현하므로 특정 컨트롤러에서만 발생하는 예외만 처리된다.

하지만 컨트롤러에 에러 처리 코드가 섞이며, 에러 처리 코드가 중복될 가능성이 높다.

그래서 스프링은 전역적으로 예외를 처리할 수 있는 좋은 기술을 제공해준다.

@RestControllerAdvice

- 전역적으로 @ExceptionHandler를 적용할 수 있는 어노테이션

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(NoSuchElementFoundException.class)
    protected ResponseEntity<?> handleNoSuchElementFoundExceptio
        final ErrorResponse errorResponse = ErrorResponse.builde
                .code("Item Not Found")
                .message(e.getMessage()).build();

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(
    }
}
```

- 여러 컨트롤러에 대해 전역적으로 ExceptionHandler 를 적용해 주어, 하나의 예외 핸들링 클래스를 만들어서 에러 처리를 위임할 수 있다.

장점

- 하나의 클래스로 모든 컨트롤러에 대해 전역적으로 예외 처리가 가능함
- 직접 정의한 에러 응답을 일관성있게 클라이언트에게 내려줄 수 있음
- 별도의 try-catch문이 없어 코드의 가독성이 높아짐

주의할 점

- 한 프로젝트당 하나의 ControllerAdvice만 관리하는 것이 좋다.

- 만약 여러 ControllerAdvice가 필요하다면 basePackages나 annotations 등을 지정해야 한다.
- 직접 구현한 Exception 클래스들은 한 공간에서 관리한다.

출처

https://gyoogle.dev/blog/computer-language/Java/Error & Exception.html

https://mangkyu.tistory.com/204

https://jojoldu.tistory.com/734

https://choiblack.tistory.com/39

https://toneyparky.tistory.com/40