8. 예외 처리와 오류 페이지

#인강/5. 스프링 MVC 2/강의#

목차

프로젝트 생성

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.5.x
- Project Metadata
  - Group: hello
  - Artifact: exception
  - Name: exception
  - Package name: **hello.exception**
  - Packaging: **Jar**
  - Java: 11

주의: 강의 영상에서 package 선택시 War라고 잘못 이야기했는데, Jar가 맞습니다.

- Dependencies: **Spring Web**, **Lombok** , **Thymeleaf**, **Validation**

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.5.1'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}

repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
  implementation 'org.springframework.boot:spring-boot-starter-validation'
  implementation 'org.springframework.boot:spring-boot-starter-web'
  compileOnly 'org.projectlombok:lombok'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
  useJUnitPlatform()
}
```

- 동작 확인
  - 기본 메인 클래스 실행(`ExceptionApplication.main()`)
  - http://localhost:8080 호출해서 Whitelabel Error Page가 나오면 정상 동작

서블릿 예외 처리 - 시작

스프링이 아닌 순수 서블릿 컨테이너는 예외를 어떻게 처리하는지 알아보자.

서블릿은 다음 2가지 방식으로 예외 처리를 지원한다.

- `Exception`(예외)
- `response.sendError(HTTP 상태 코드, 오류 메시지)`

Exception(예외)

자바 직접 실행

자바의 메인 메서드를 직접 실행하는 경우 `main`이라는 이름의 쓰레드가 실행된다.
실행 도중에 예외를 잡지 못하고 처음 실행한 `main()` 메서드를 넘어서 예외가 던져지면, 예외 정보를 남기고 해당 쓰레드는 종료된다.

웹 애플리케이션

웹 애플리케이션은 사용자 요청별로 별도의 쓰레드가 할당되고, 서블릿 컨테이너 안에서 실행된다.
애플리케이션에서 예외가 발생했는데, 어디선가 try ~ catch로 예외를 잡아서 처리하면 아무런 문제가 없다. 그런데 만약에 애플리케이션에서 예외를 잡지 못하고, 서블릿 밖으로 까지 예외가 전달되면 어떻게 동작할까?

```
WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
```

결국 톰캣 같은 WAS 까지 예외가 전달된다. WAS는 예외가 올라오면 어떻게 처리해야 할까?
한번 테스트 해보자.

먼저 스프링 부트가 제공하는 기본 예외 페이지가 있는데 이건 꺼두자(뒤에서 다시 설명하겠다.)

application.properties

```
server.error.whitelabel.enabled=false
```

ServletExController - 서블릿 예외 컨트롤러

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Slf4j
@Controller
public class ServletExController {

    @GetMapping("/error-ex")
    public void errorEx() {
        throw new RuntimeException("예외 발생!");
    }

}
```

실행해보면 다음처럼 tomcat이 기본으로 제공하는 오류 화면을 볼 수 있다.

```
HTTP Status 500 – Internal Server Error
```

웹 브라우저에서 개발자 모드로 확인해보면 HTTP 상태 코드가 500으로 보인다.
`Exception`의 경우 서버 내부에서 처리할 수 없는 오류가 발생한 것으로 생각해서 HTTP 상태 코드 500을 반환한다.

이번에는 아무사이트나 호출해보자.
`http://localhost:8080/no-page`

```
HTTP Status 404 – Not Found
```

톰캣이 기본으로 제공하는 404 오류 화면을 볼 수 있다.

response.sendError(HTTP 상태 코드, 오류 메시지)

오류가 발생했을 때 `HttpServletResponse`가 제공하는 `sendError`라는 메서드를 사용해도 된다.
이것을 호출한다고 당장 예외가 발생하는 것은 아니지만, 서블릿 컨테이너에게 오류가 발생했다는 점을 전달할 수 있다.
이 메서드를 사용하면 HTTP 상태 코드와 오류 메시지도 추가할 수 있다.

- `response.sendError(HTTP 상태 코드)`
- `response.sendError(HTTP 상태 코드, 오류 메시지)`

ServletExController - 추가

```
@GetMapping("/error-404")
public void error404(HttpServletResponse response) throws IOException {
    response.sendError(404, "404 오류!");
}

@GetMapping("/error-500")
public void error500(HttpServletResponse response) throws IOException {
    response.sendError(500);
}
```

sendError 흐름

```
WAS(sendError 호출 기록 확인) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러
(response.sendError())
```

`response.sendError()`를 호출하면 `response` 내부에는 오류가 발생했다는 상태를 저장해둔다.
그리고 서블릿 컨테이너는 고객에게 응답 전에 `response`에 `sendError()`가 호출되었는지 확인한다.
그리고 호출되었다면 설정한 오류 코드에 맞추어 기본 오류 페이지를 보여준다.

실행해보면 다음처럼 서블릿 컨테이너가 기본으로 제공하는 오류 화면을 볼 수 있다.

- http://localhost:8080/error-ex
- http://localhost:8080/error-404

- http://localhost:8080/error-500

```
HTTP Status 404 – Bad Request
HTTP Status 500 – Internal Server Error
```

정리
서블릿 컨테이너가 제공하는 기본 예외 처리 화면은 사용자가 보기에 불편하다. 의미 있는 오류 화면을 제공해보자.

서블릿 예외 처리 - 오류 화면 제공

서블릿 컨테이너가 제공하는 기본 예외 처리 화면은 고객 친화적이지 않다. 서블릿이 제공하는 오류 화면 기능을 사용해보자.

서블릿은 `Exception`(예외)가 발생해서 서블릿 밖으로 전달되거나 또는 `response.sendError()`가 호출되었을 때 각각의 상황에 맞춘 오류 처리 기능을 제공한다.
이 기능을 사용하면 친절한 오류 처리 화면을 준비해서 고객에게 보여줄 수 있다.

과거에는 `web.xml`이라는 파일에 다음과 같이 오류 화면을 등록했다.

```
<web-app>
    <error-page>
      <error-code>404</error-code>
      <location>/error-page/404.html</location>
    </error-page>
    <error-page>
      <error-code>500</error-code>
      <location>/error-page/500.html</location>
    </error-page>
    <error-page>
      <exception-type>java.lang.RuntimeException</exception-type>
      <location>/error-page/500.html</location>
    </error-page>
</web-app>
```

지금은 스프링 부트를 통해서 서블릿 컨테이너를 실행하기 때문에, 스프링 부트가 제공하는 기능을 사용해서 서블릿 오류 페이지를 등록하면 된다.

서블릿 오류 페이지 등록

```
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);
    }
}
```

- `response.sendError(404)` : `errorPage404` 호출
- `response.sendError(500)` : `errorPage500` 호출
- `RuntimeException` 또는 그 자식 타입의 예외: `errorPageEx` 호출

500 예외가 서버 내부에서 발생한 오류라는 뜻을 포함하고 있기 때문에 여기서는 예외가 발생한 경우도 500 오류 화면으로 처리했다.

오류 페이지는 예외를 다룰 때 해당 예외와 그 자식 타입의 오류를 함께 처리한다. 예를 들어서 위의 경우 `RuntimeException`은 물론이고 `RuntimeException`의 자식도 함께 처리한다.

오류가 발생했을 때 처리할 수 있는 컨트롤러가 필요하다. 예를 들어서 `RuntimeException` 예외가 발생하면 `errorPageEx`에서 지정한 `/error-page/500`이 호출된다.

해당 오류를 처리할 컨트롤러가 필요하다.

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Slf4j
@Controller
public class ErrorPageController {

    @RequestMapping("/error-page/404")
    public String errorPage404(HttpServletRequest request, HttpServletResponse
response) {
        log.info("errorPage 404");
        return "error-page/404";
    }

    @RequestMapping("/error-page/500")
    public String errorPage500(HttpServletRequest request, HttpServletResponse
response) {
        log.info("errorPage 500");
        return "error-page/500";
    }

}
```

오류 처리 View

/templates/error-page/404.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>404 오류 화면</h2>
    </div>

    <div>
        <p>오류 화면 입니다.</p>
    </div>

    <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

/templates/error-page/500.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>500 오류 화면</h2>
```

```
    </div>

    <div>
        <p>오류 화면 입니다.</p>
    </div>

    <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

테스트 해보자.

- http://localhost:8080/error-ex
- http://localhost:8080/error-404
- http://localhost:8080/error-500

설정한 오류 페이지가 정상 노출되는 것을 확인할 수 있다.

서블릿 예외 처리 - 오류 페이지 작동 원리

서블릿은 `Exception`(예외)가 발생해서 서블릿 밖으로 전달되거나 또는 `response.sendError()`가 호출되었을 때 설정된 오류 페이지를 찾는다.

예외 발생 흐름

```
WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
```

sendError 흐름

```
WAS(sendError 호출 기록 확인) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러
(response.sendError())
```

WAS는 해당 예외를 처리하는 오류 페이지 정보를 확인한다.

`new ErrorPage(RuntimeException.class, "/error-page/500")`

예를 들어서 `RuntimeException` 예외가 WAS까지 전달되면, WAS는 오류 페이지 정보를 확인한다. 확인해보니 `RuntimeException`의 오류 페이지로 `/error-page/500`이 지정되어 있다. WAS는 오류 페이지를 출력하기 위해 `/error-page/500`를 다시 요청한다.

오류 페이지 요청 흐름

```
WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(/error-page/500) -> View
```

예외 발생과 오류 페이지 요청 흐름

```
1. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
2. WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(/error-page/500) -> View
```

중요한 점은 웹 브라우저(클라이언트)는 서버 내부에서 이런 일이 일어나는지 전혀 모른다는 점이다. 오직 서버 내부에서 오류 페이지를 찾기 위해 추가적인 호출을 한다.

정리하면 다음과 같다.

1. 예외가 발생해서 WAS까지 전파된다.
2. WAS는 오류 페이지 경로를 찾아서 내부에서 오류 페이지를 호출한다. 이때 오류 페이지 경로로 필터, 서블릿, 인터셉터, 컨트롤러가 모두 다시 호출된다.

필터와 인터셉터가 다시 호출되는 부분은 조금 뒤에 자세히 설명하겠다.

오류 정보 추가

WAS는 오류 페이지를 단순히 다시 요청만 하는 것이 아니라, 오류 정보를 `request`의 `attribute`에 추가해서 넘겨준다.
필요하면 오류 페이지에서 이렇게 전달된 오류 정보를 사용할 수 있다.

ErrorPageController - 오류 출력

```
package hello.exception.servlet;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@Slf4j
@Controller
public class ErrorPageController {

    //RequestDispatcher 상수로 정의되어 있음
    public static final String ERROR_EXCEPTION =
"javax.servlet.error.exception";
    public static final String ERROR_EXCEPTION_TYPE =
"javax.servlet.error.exception_type";
    public static final String ERROR_MESSAGE = "javax.servlet.error.message";
    public static final String ERROR_REQUEST_URI =
"javax.servlet.error.request_uri";
    public static final String ERROR_SERVLET_NAME =
"javax.servlet.error.servlet_name";
    public static final String ERROR_STATUS_CODE =
"javax.servlet.error.status_code";

    @RequestMapping("/error-page/404")
    public String errorPage404(HttpServletRequest request, HttpServletResponse
response) {
        log.info("errorPage 404");
        printErrorInfo(request);
        return "error-page/404";
    }

    @RequestMapping("/error-page/500")
    public String errorPage500(HttpServletRequest request, HttpServletResponse
response) {
        log.info("errorPage 500");
        printErrorInfo(request);
```

```
        return "error-page/500";
    }

    private void printErrorInfo(HttpServletRequest request) {
        log.info("ERROR_EXCEPTION: ex=",
request.getAttribute(ERROR_EXCEPTION));
        log.info("ERROR_EXCEPTION_TYPE: {}",
request.getAttribute(ERROR_EXCEPTION_TYPE));
        log.info("ERROR_MESSAGE: {}", request.getAttribute(ERROR_MESSAGE)); //
ex의 경우 NestedServletException 스프링이 한번 감싸서 반환
        log.info("ERROR_REQUEST_URI: {}",
request.getAttribute(ERROR_REQUEST_URI));
        log.info("ERROR_SERVLET_NAME: {}",
request.getAttribute(ERROR_SERVLET_NAME));
        log.info("ERROR_STATUS_CODE: {}",
request.getAttribute(ERROR_STATUS_CODE));
        log.info("dispatchType={}", request.getDispatcherType());
    }

}
```

request.attribute에 서버가 담아준 정보

- `javax.servlet.error.exception` : 예외
- `javax.servlet.error.exception_type` : 예외 타입
- `javax.servlet.error.message` : 오류 메시지
- `javax.servlet.error.request_uri` : 클라이언트 요청 URI
- `javax.servlet.error.servlet_name` : 오류가 발생한 서블릿 이름
- `javax.servlet.error.status_code` : HTTP 상태 코드

서블릿 예외 처리 - 필터

목표

예외 처리에 따른 필터와 인터셉터 그리고 서블릿이 제공하는 `DispatchType` 이해하기

예외 발생과 오류 페이지 요청 흐름

```
1. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
2. WAS `/error-page/500` 다시 요청 -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러(/error-page/500) -> View
```

오류가 발생하면 오류 페이지를 출력하기 위해 WAS 내부에서 다시 한번 호출이 발생한다. 이때 필터, 서블릿, 인터셉터도 모두 다시 호출된다. 그런데 로그인 인증 체크 같은 경우를 생각해보면, 이미 한번 필터나, 인터셉터에서 로그인 체크를 완료했다. 따라서 서버 내부에서 오류 페이지를 호출한다고 해서 해당 필터나 인터셉트가 한번 더 호출되는 것은 매우 비효율적이다.
결국 클라이언트로 부터 발생한 정상 요청인지, 아니면 오류 페이지를 출력하기 위한 내부 요청인지 구분할 수 있어야 한다. 서블릿은 이런 문제를 해결하기 위해 `DispatcherType` 이라는 추가 정보를 제공한다.

DispatcherType

필터는 이런 경우를 위해서 `dispatcherTypes` 라는 옵션을 제공한다.
이전 강의의 마지막에 다음 로그를 추가했다.
`log.info("dispatchType={}", request.getDispatcherType())`
그리고 출력해보면 오류 페이지에서 `dispatchType=ERROR` 로 나오는 것을 확인할 수 있다.

고객이 처음 요청하면 `dispatcherType=REQUEST` 이다.
이렇듯 서블릿 스펙은 실제 고객이 요청한 것인지, 서버가 내부에서 오류 페이지를 요청하는 것인지 `DispatcherType` 으로 구분할 수 있는 방법을 제공한다.

`javax.servlet.DispatcherType`

```
public enum DispatcherType {
    FORWARD,
    INCLUDE,
    REQUEST,
    ASYNC,
    ERROR
}
```

DispatcherType

- `REQUEST` : 클라이언트 요청
- `ERROR` : 오류 요청

- `FORWARD` : MVC에서 배웠던 서블릿에서 다른 서블릿이나 JSP를 호출할 때 `RequestDispatcher.forward(request, response);`
- `INCLUDE` : 서블릿에서 다른 서블릿이나 JSP의 결과를 포함할 때 `RequestDispatcher.include(request, response);`
- `ASYNC` : 서블릿 비동기 호출

필터와 DispatcherType

필터와 DispatcherType이 어떻게 사용되는지 알아보자.

LogFilter - DispatcherType 로그 추가

```
package hello.exception.filter;

import lombok.extern.slf4j.Slf4j;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.util.UUID;

@Slf4j
public class LogFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        log.info("log filter init");
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain) throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        String requestURI = httpRequest.getRequestURI();

        String uuid = UUID.randomUUID().toString();
```

```
        try {
            log.info("REQUEST  [{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI);
            chain.doFilter(request, response);
        } catch (Exception e) {
            throw e;
        } finally {
            log.info("RESPONSE [{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI);
        }

    }

    @Override
    public void destroy() {
        log.info("log filter destroy");
    }
}
```

로그를 출력하는 부분에 `request.getDispatcherType()` 을 추가해두었다.

WebConfig

```
package hello.exception;

import hello.exception.filter.LogFilter;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import javax.servlet.DispatcherType;
import javax.servlet.Filter;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Bean
```

```
    public FilterRegistrationBean logFilter() {
        FilterRegistrationBean<Filter> filterRegistrationBean = new
FilterRegistrationBean<>();
        filterRegistrationBean.setFilter(new LogFilter());
        filterRegistrationBean.setOrder(1);
        filterRegistrationBean.addUrlPatterns("/*");
        filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST,
DispatcherType.ERROR);
        return filterRegistrationBean;
    }

}
```

`filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST, DispatcherType.ERROR);`
이렇게 두 가지를 모두 넣으면 클라이언트 요청은 물론이고, 오류 페이지 요청에서도 필터가 호출된다.
아무것도 넣지 않으면 기본 값이 `DispatcherType.REQUEST`이다. 즉 클라이언트의 요청이 있는 경우에만 필터가 적용된다. 특별히 오류 페이지 경로도 필터를 적용할 것이 아니면, 기본 값을 그대로 사용하면 된다.

물론 오류 페이지 요청 전용 필터를 적용하고 싶으면 `DispatcherType.ERROR`만 지정하면 된다.

서블릿 예외 처리 - 인터셉터

인터셉터 중복 호출 제거

LogInterceptor - DispatcherType 로그 추가

```
package hello.exception.interceptor;

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

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.util.UUID;
```

```
@Slf4j
public class LogInterceptor implements HandlerInterceptor {

    public static final String LOG_ID = "logId";

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler) throws Exception {

        String requestURI = request.getRequestURI();

        String uuid = UUID.randomUUID().toString();
        request.setAttribute(LOG_ID, uuid);

        log.info("REQUEST  [{}][{}][{}][{}]", uuid,
request.getDispatcherType(), requestURI, handler);
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
        log.info("postHandle [{}]", modelAndView);
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        String requestURI = request.getRequestURI();
        String logId = (String)request.getAttribute(LOG_ID);
        log.info("RESPONSE [{}][{}][{}]", logId, request.getDispatcherType(),
requestURI);
        if (ex != null) {
            log.error("afterCompletion error!!", ex);
        }
    }
}
```

앞서 필터의 경우에는 필터를 등록할 때 어떤 `DispatcherType`인 경우에 필터를 적용할 지 선택할 수 있었다. 그런데 인터셉터는 서블릿이 제공하는 기능이 아니라 스프링이 제공하는 기능이다. 따라서 `DispatcherType`과 무관하게 항상 호출된다.

대신에 인터셉터는 다음과 같이 요청 경로에 따라서 추가하거나 제외하기 쉽게 되어 있기 때문에, 이러한 설정을 사용해서 오류 페이지 경로를 `excludePathPatterns`를 사용해서 빼주면 된다.

```
package hello.exception;

import hello.exception.filter.LogFilter;
import hello.exception.interceptor.LogInterceptor;
import org.springframework.boot.web.servlet.FilterRegistrationBean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import javax.servlet.DispatcherType;
import javax.servlet.Filter;

@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LogInterceptor())
                .order(1)
                .addPathPatterns("/**")
                .excludePathPatterns(
                        "/css/**", "/*.ico"
                        , "/error", "/error-page/**" //오류 페이지 경로
                );
    }

    //@Bean
    public FilterRegistrationBean logFilter() {
        FilterRegistrationBean<Filter> filterRegistrationBean = new
FilterRegistrationBean<>();
```

```
        filterRegistrationBean.setFilter(new LogFilter());
        filterRegistrationBean.setOrder(1);
        filterRegistrationBean.addUrlPatterns("/*");
        filterRegistrationBean.setDispatcherTypes(DispatcherType.REQUEST,
DispatcherType.ERROR);
        return filterRegistrationBean;
    }

}
```

인터셉터와 중복으로 처리되지 않기 위해 앞의 `logFilter()` 의 `@Bean` 에 주석을 달아두자.
여기에서 `/error-page/**` 를 제거하면 `error-page/500` 같은 내부 호출의 경우에도 인터셉터가 호출된다.

전체 흐름 정리

`/hello` 정상 요청

```
WAS(/hello, dispatchType=REQUEST) -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러 -> View
```

`/error-ex` 오류 요청

- 필터는 `DispatchType` 으로 중복 호출 제거 (`dispatchType=REQUEST`)
- 인터셉터는 경로 정보로 중복 호출 제거(`excludePathPatterns("/error-page/**")`)

```
1. WAS(/error-ex, dispatchType=REQUEST) -> 필터 -> 서블릿 -> 인터셉터 -> 컨트롤러
2. WAS(여기까지 전파) <- 필터 <- 서블릿 <- 인터셉터 <- 컨트롤러(예외발생)
3. WAS 오류 페이지 확인
4. WAS(/error-page/500, dispatchType=ERROR) -> 필터(x) -> 서블릿 -> 인터셉터(x) ->
컨트롤러(/error-page/500) -> View
```

스프링 부트 - 오류 페이지1

지금까지 예외 처리 페이지를 만들기 위해서 다음과 같은 복잡한 과정을 거쳤다.

- `WebServerCustomizer`를 만들고
- 예외 종류에 따라서 `ErrorPage`를 추가하고
- 예외 처리용 컨트롤러 `ErrorPageController`를 만듦

스프링 부트는 이런 과정을 모두 기본으로 제공한다.

- `ErrorPage`를 자동으로 등록한다. 이때 `/error`라는 경로로 기본 오류 페이지를 설정한다.
  - `new ErrorPage("/error")`, 상태코드와 예외를 설정하지 않으면 기본 오류 페이지로 사용된다.
  - 서블릿 밖으로 예외가 발생하거나, `response.sendError(...)`가 호출되면 모든 오류는 `/error`를 호출하게 된다.
- `BasicErrorController`라는 스프링 컨트롤러를 자동으로 등록한다.
  - `ErrorPage`에서 등록한 `/error`를 매핑해서 처리하는 컨트롤러다.

> **참고**
> `ErrorMvcAutoConfiguration`이라는 클래스가 오류 페이지를 자동으로 등록하는 역할을 한다.

주의

스프링 부트가 제공하는 기본 오류 메커니즘을 사용하도록 **WebServerCustomizer**에 있는 `@Component`를 주석 처리하자.

이제 오류가 발생했을 때 오류 페이지로 `/error`를 기본 요청한다. 스프링 부트가 자동 등록한 `BasicErrorController`는 이 경로를 기본으로 받는다.

개발자는 오류 페이지만 등록

`BasicErrorController`는 기본적인 로직이 모두 개발되어 있다.
개발자는 오류 페이지 화면만 `BasicErrorController`가 제공하는 룰과 우선순위에 따라서 등록하면 된다. 정적 HTML이면 정적 리소스, 뷰 템플릿을 사용해서 동적으로 오류 화면을 만들고 싶으면 뷰 템플릿 경로에 오류 페이지 파일을 만들어서 넣어두기만 하면 된다.

뷰 선택 우선순위

`BasicErrorController`의 처리 순서

1. 뷰 템플릿
   - `resources/templates/error/500.html`
   - `resources/templates/error/5xx.html`
2. 정적 리소스(`static` , `public`)
   - `resources/static/error/400.html`

- `resources/static/error/404.html`
- `resources/static/error/4xx.html`

3. 적용 대상이 없을 때 뷰 이름(`error`)
   - `resources/templates/error.html`

해당 경로 위치에 HTTP 상태 코드 이름의 뷰 파일을 넣어두면 된다.
뷰 템플릿이 정적 리소스보다 우선순위가 높고, 404, 500처럼 구체적인 것이 5xx처럼 덜 구체적인 것 보다 우선순위가 높다.
5xx, 4xx 라고 하면 500대, 400대 오류를 처리해준다.

오류 뷰 템플릿 추가

`resources/templates/error/4xx.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>4xx 오류 화면 스프링 부트 제공</h2>
    </div>

    <div>
        <p>오류 화면 입니다.</p>
    </div>

    <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

`resources/templates/error/404.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>404 오류 화면 스프링 부트 제공</h2>
    </div>

    <div>
        <p>오류 화면 입니다.</p>
    </div>

    <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

resources/templates/error/500.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>500 오류 화면 스프링 부트 제공</h2>
    </div>

    <div>
```

```
        <p>오류 화면 입니다.</p>
    </div>
    <hr class="my-4">

</div> <!-- /container -->

</body>
</html>
```

등록한 오류 페이지

`resources/templates/error/4xx.html`
`resources/templates/error/404.html`
`resources/templates/error/500.html`

테스트

- http://localhost:8080/error-404 ➡ `404.html`
- http://localhost:8080/error-400 ➡ `4xx.html` (400 오류 페이지가 없지만 4xx가 있음)
- http://localhost:8080/error-500 ➡ `500.html`
- http://localhost:8080/error-ex ➡ `500.html` (예외는 500으로 처리)

스프링 부트 - 오류 페이지2

BasicErrorController가 제공하는 기본 정보들

`BasicErrorController` 컨트롤러는 다음 정보를 model에 담아서 뷰에 전달한다. 뷰 템플릿은 이 값을 활용해서 출력할 수 있다.

```
* timestamp: Fri Feb 05 00:00:00 KST 2021
* status: 400
* error: Bad Request
* exception: org.springframework.validation.BindException
* trace: 예외 trace
* message: Validation failed for object='data'. Error count: 1
* errors: Errors(BindingResult)
* path: 클라이언트 요청 경로 (`/hello`)
```

오류 정보 추가 - `resources/templates/error/500.html`

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="utf-8">
</head>
<body>

<div class="container" style="max-width: 600px">
    <div class="py-5 text-center">
        <h2>500 오류 화면 스프링 부트 제공</h2>
    </div>

    <div>
        <p>오류 화면 입니다.</p>
    </div>

    <ul>
        <li>오류 정보</li>
        <ul>
            <li th:text="|timestamp: ${timestamp}|"></li>
            <li th:text="|path: ${path}|"></li>
            <li th:text="|status: ${status}|"></li>
            <li th:text="|message: ${message}|"></li>
            <li th:text="|error: ${error}|"></li>
            <li th:text="|exception: ${exception}|"></li>
            <li th:text="|errors: ${errors}|"></li>
            <li th:text="|trace: ${trace}|"></li>
        </ul>
        </li>
    </ul>

    <hr class="my-4">

</div> <!-- /container -->
```

```
</body>
</html>
```

오류 관련 내부 정보들을 고객에게 노출하는 것은 좋지 않다. 고객이 해당 정보를 읽어도 혼란만 더해지고, 보안상 문제가 될 수도 있다.
그래서 `BasicErrorController` 오류 컨트롤러에서 다음 오류 정보를 `model` 에 포함할지 여부 선택할 수 있다.

`application.properties`

- `server.error.include-exception=false` : `exception` 포함 여부(`true` , `false`)
- `server.error.include-message=never` : `message` 포함 여부
- `server.error.include-stacktrace=never` : `trace` 포함 여부
- `server.error.include-binding-errors=never` : `errors` 포함 여부

`application.properties`

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

기본 값이 `never` 인 부분은 다음 3가지 옵션을 사용할 수 있다.
`never, always, on_param`

- `never` : 사용하지 않음
- `always` :항상 사용
- `on_param` : 파라미터가 있을 때 사용

`on_param` 은 파라미터가 있으면 해당 정보를 노출한다. 디버그 시 문제를 확인하기 위해 사용할 수 있다. 그런데 이 부분도 개발 서버에서 사용할 수 있지만, 운영 서버에서는 권장하지 않는다.
`on_param` 으로 설정하고 다음과 같이 HTTP 요청시 파라미터를 전달하면 해당 정보들이 `model` 에 담겨서 뷰 템플릿에서 출력된다.
`message=&errors=&trace=`

테스트
http://localhost:8080/error-ex?message=&errors=&trace=

실무에서는 이것들을 노출하면 안된다! 사용자에게는 이쁜 오류 화면과 고객이 이해할 수 있는 간단한 오류 메시지를 보여주고 오류는 서버에 로그로 남겨서 로그로 확인해야 한다.

스프링 부트 오류 관련 옵션

- `server.error.whitelabel.enabled=true` : 오류 처리 화면을 못 찾을 시, 스프링 whitelabel 오류 페이지 적용
- `server.error.path=/error` : 오류 페이지 경로, 스프링이 자동 등록하는 서블릿 글로벌 오류 페이지 경로와 `BasicErrorController` 오류 컨트롤러 경로에 함께 사용된다.

확장 포인트

에러 공통 처리 컨트롤러의 기능을 변경하고 싶으면 `ErrorController` 인터페이스를 상속 받아서 구현하거나 `BasicErrorController` 상속 받아서 기능을 추가하면 된다.

정리

스프링 부트가 기본으로 제공하는 오류 페이지를 활용하면 오류 페이지와 관련된 대부분의 문제는 손쉽게 해결할 수 있다.

정리