에러 핸들링 가이드

1 목표

AM 프로젝트의 백엔드 애플리케이션에서 에러를 핸들링할 때 필요한 점들을 숙지한다.

2 독자

AM 프로젝트에서 백엔드 애플리케이션 개발에 기여하고 있는 개발자

3 필수 조건

- SOLID
- 레이어드 아키텍처
- DIP와 DDD 원칙을 따르는 아키텍처 스타일

핵심은 포트-앤-어댑터 패턴에서 예외를 아래와 같이 처리한다는 것이다.

- 아웃바운드 어댑터: 발생한 예외를 도메인에 선언한 예외로 번역한다.
- 인바운드 어댑터: 도메인에 선언된 예외를 핸들링한다.

4 단계

4.1 비즈니스에서 필요한 예외를 도메인 영역에 선언한다.

```
public interface FoodRepository {

    void save(Food food);
}

public class FoodNameDuplicatedException extends RuntimeException {

    public FoodNameDuplicatedException(final Throwable cause) {
        super(cause);
    }

    public FoodNameDuplicatedException(
            final String message,
            final Throwable cause) {
        super(message, cause);
    }
}
```

예외가 발생한 곳에서 바로 핸들링 해야 한다면 Exception을,

예외가 명시적으로 핸들링하기 전까지 계속 전파되어야 한다면 RuntimeException을 상속한다.

예외를 다른 예외로 번역할 때는 반드시 생성자의 매개 변수로 Throwable을 받게 한다.

4.2 아웃바운드 어댑터에서 발생한 예외를 도메인의 예외로 번역한다.

```
@Entity
@Table(
        name = "FOOD",
        uniqueConstraints = @UniqueConstraint(columnNames = "NAME")
)
class FoodEntity {
    @Id
    private UUID id;
    @Column
    private String name;
    @Column
    private int calories;

    protected FoodEntity() {
    }

    FoodEntity(final UUID id, final String name, final int calories) {
        this.id = Objects.requireNonNull(id);
        this.name = Objects.requireNonNull(name);
        this.calories = calories;
    }

    UUID getId() {
        return id;
    }

    String getName() {
        return name;
    }

    int getCalories() {
        return calories;
    }
}

class FoodRepositoryImpl implements FoodRepository {
    private final FoodSpringDataRepository repository;

    public FoodRepositoryImpl(final FoodSpringDataRepository repository) {
        this.repository = Objects.requireNonNull(repository);
    }

    @Override
    public void save(final Food food) {
        final var entity = toEntity(food);

        try {
            repository.save(entity);
        } catch (DataIntegrityViolationException e) {
            throw new FoodNameDuplicatedException("original=" + food, e);
        }
    }
}
```

위는 Spring Data JPA를 활용한 예제로, Unique Constraint로 인해 기존에 저장된 이름을 사용해 새로운 레코드를 저장하려 할 때 무결성 예외(`DataIntegrityViolationException`)가 발생할 수 있다.

하지만 우리의 애플리케이션 코어 영역, 즉, 도메인 및 애플리케이션 영역은 순수한 자바로 작성되어 있기 때문에 Spring Data JPA에서 선언한 `DataIntegrityViolationException` 을 알 수 없고 부를 수도 없다. 따라서 아웃바운드 포트에 해당하는 `FoodRepository` 를 구현하는 `FoodRepositoryImp` 에서 이를 도메인에 선언된 예외로 번역해 주어야 한다.

```
record CookRequest(
        String name,
        int calories
) {
}

record CookResponse(
        UUID id
) {
}

class FoodClientService implements FoodService {
    private final WebClient webClient;

    FoodClientService(final WebClient webClient) {
        this.webClient = Objects.requireNonNull(webClient);
    }

    @Override
    public FoodId cook(final FoodDetails details) {
        final var body = new CookRequest(details.name(), details.calories().value());

        return webClient.post()
                .uri("/foods")
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON)
                .bodyValue(body)
                .retrieve()
                .bodyToMono(CookResponse.class)
                .map(it -> new FoodId(it.id()))
                .onErrorMap(WebClientResponseException.class, it -> {
                    if (it.getStatusCode().isSameCodeAs(HttpStatus.CONFLICT)) {
                        throw new FoodNameDuplicatedException(it);
                    }
                    throw it;
                })
                .block();
    }
}
```

클라이언트 서비스를 구현할 때 역시 마찬가지이다. 구현 라이브러리에 따라 다를 수 있으나, 위의 예제처럼 스프링의 WebClient를 이용하는 경우에는 `WebClientResponseException` 를 통해 실패한 요청의 상태 코드를 확인할 수 있다.

4.3 인바운드 어댑터에서는 도메인에서 선언한 예외만 핸들링한다.

```
@RestController
@RequestMapping("/foods")
class FoodController {
    private static final Logger logger = LoggerFactory.getLogger(FoodController.class);

    private final FoodService foodService;

    FoodController(final FoodService foodService) {
        this.foodService = Objects.requireNonNull(foodService);
    }

    //
}
```

컨트롤러 내부에서만 동작하는 한정된 예외 처리 로직은 아래와 같이 구현할 수 있다.

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
CookResponse cook(@RequestBody CookRequest request) {
    final var details = new FoodDetails(request.name(), new Calories(request.calories()));

    final FoodId foodId;
    try {
        foodId = foodService.cook(details);
    } catch (FoodNameDuplicatedException e) {
        logger.info("food name was duplicated.", e);
        throw new ResponseStatusException(HttpStatus.CONFLICT, null, e);
    }

    return new CookResponse(foodId.value());
}
```

스프링의 컨트롤러의 핸들러 메소드에서 try~catch 구문을 구현할 수 있다. 이 경우 반환될 HTTP 상태 코드를 직접 제어하기 위해 위와 같이 spring-web에서 제공하는 `ResponseStatusException` 와 같은 예외를 사용할 수 있다.

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
CookResponse cook(@RequestBody CookRequest request) {
    final var details = new FoodDetails(request.name(), new Calories(request.calories()));

    final var foodId = foodService.cook(details);

    return new CookResponse(foodId.value());
}

@ExceptionHandler(FoodNameDuplicatedException.class)
@ResponseStatus(HttpStatus.CONFLICT)
void handleFoodNameDuplicatedException(final FoodNameDuplicatedException e) {
    logger.info("food name was duplicated.", e);
```

다른 방식으로는 컨트롤러에서 @ExceptionHandler를 가지는 예외 핸들러를 작성할 수 있다.

다만 @ExceptionHandler는 같은 컨트롤러 내 다른 핸들러 메소드에도 적용되기 때문에 주의가 필요하다.

4.4 HTTP API를 서비스하는 경우 처리되지 않은 예외를 위한 ControllerAdvice를 작성한다.

```
@RestControllerAdvice
class GlobalControllerAdvice {
    private static final Logger logger = LoggerFactory.getLogger(GlobalControllerAdvice.class);

    @ExceptionHandler(Throwable.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    void handleThrowable(final Throwable t) {
        logger.warn("Unhandled exception was thrown", t);
    }
}
```

`@ControllerAdvice` 를 열어 보면 아래와 같은 문구가 있다.

> By default, the methods in an {@code @ControllerAdvice} apply globally to all controllers.

따라서 위와 같이 아무런 설정을 추가하지 않으면 모든 컨트롤러를 대상으로 동작하는 예외 처리 로직을 구현할 수 있다.

`@ControllerAdvice` 는 WebFlux에서도 지원하는 어노테이션 기반의 선언법이며,

스프링의 주요 테스트 프레임워크에서도 지원하기 때문에 활용도가 높다.

4.4.1 ProblemDetails 도입

IETF(Internet Engineering Task Force)가 HTTP API의 에러 메시지 표준화를 시도한 바 있다. (RFC 7807 Problem Details for HTTP APIs)

스프링도 이를 지원하고 있으며, 해당 기능을 활성화하는 옵션은 아래와 같다.

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnProperty(prefix = "spring.mvc.problemdetails", name = "enabled", havingValue = "true")
static class ProblemDetailsErrorHandlingConfiguration {

    @Bean
    @ConditionalOnMissingBean(ResponseEntityExceptionHandler.class)
    ProblemDetailsExceptionHandler problemDetailsExceptionHandler() { return new ProblemDetailsExceptionHandler(); }

}
```

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnProperty(prefix = "spring.webflux.problemdetails", name = "enabled", havingValue = "true")
static class ProblemDetailsErrorHandlingConfiguration {

    @Bean
    @ConditionalOnMissingBean(ResponseEntityExceptionHandler.class)
    ProblemDetailsExceptionHandler problemDetailsExceptionHandler() {
        return new ProblemDetailsExceptionHandler();
    }

}
```

- spring.mvc.problemdetails.enabled=true
- spring.webflux.problemdetails.enabled=true

4.4.2 Spring Boot ErrorProperties 설정

- server.error.include-exception=true
- server.error.include-message=ALWAYS
- server.error.include-stacktrace=ALWAYS

위는 Spring Boot가 기본적으로 에러 핸들링을 할 때 사용하는 옵션이다.

운영 환경이 아닌, 로컬 또는 개발 환경에서 손쉬운 디버깅을 위해 위와 같은 옵션을 고려할 수도 있다.

단, 앞 항목에서 언급된 ProblemDetails 기본 핸들러를 활성화하면 이 옵션은 의미가 없어진다.

5 참고 문헌

- https://datatracker.ietf.org/doc/html/rfc7807