Validation Check

Validation Check

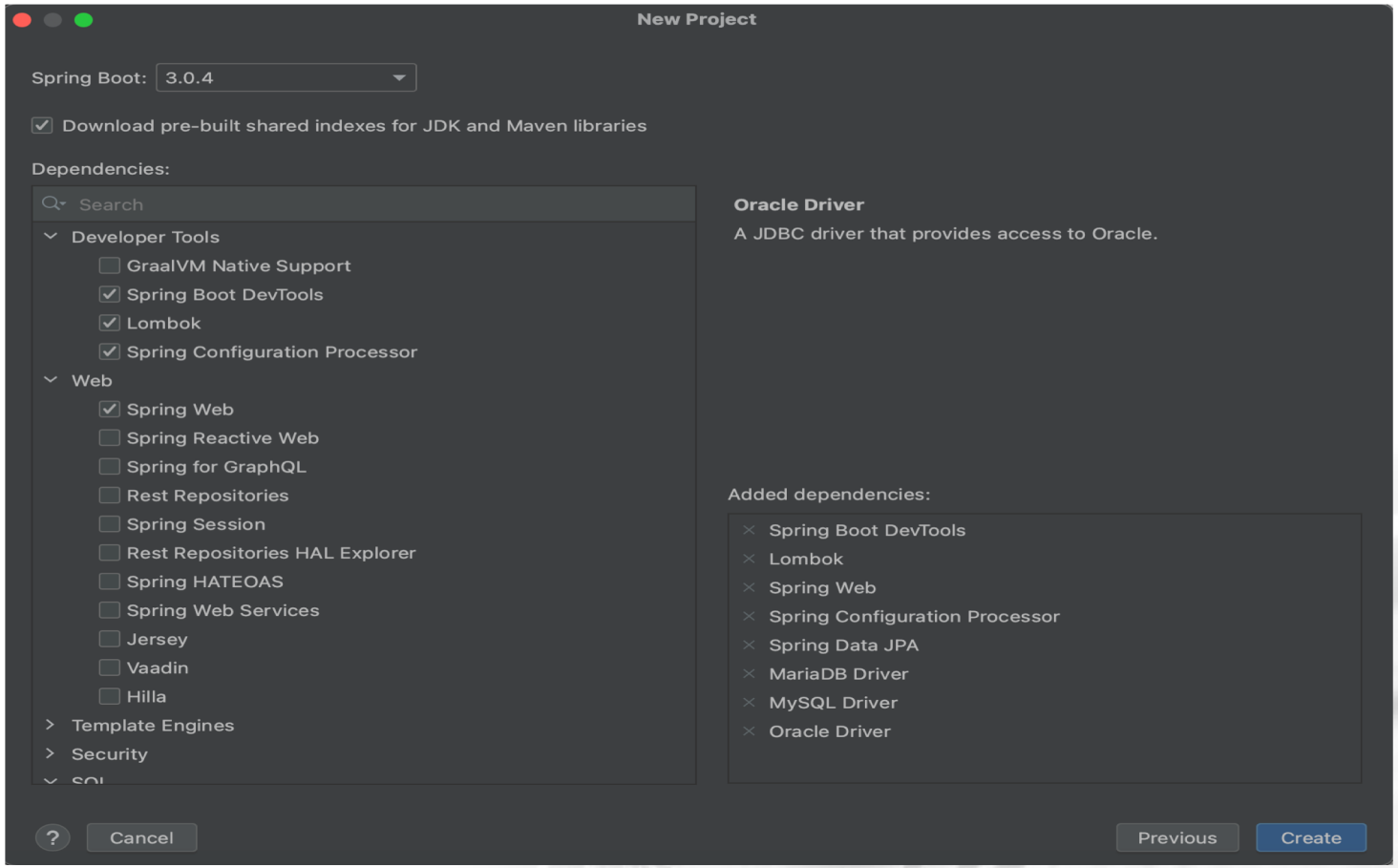
- 유효성 검사
  - ✓ 견고한 Server Application을 만들기 위해서는 클라이언트의 요청 값을 검증해야 함
  - ✓ 사용자가 전송하는 데이터는 개발자가 생각하는 것 보다 다양한 형태로 서버에 전달
  - ✓ 애플리케이션 서버를 공격하거나 데이터를 유출하기 위한 여러 형태로 서버에게 요청
  - ✓ 유효성 검사를 이용해서 NullPointerException 이나 예상하지 못한 오류를 사전에 피할 수 있음
  - ✓ 사용자가 요청하는 데이터를 검증하는 방법
    - 요청 데이터 자체의 포맷이나 무결성을 검증하는 방법 – Controller 검증 가능
    - 데이터 저장소에 데이터를 조회하여 데이터 유무를 검증하는 방법 – Service 나 Component 클래스에서 검증
  - ✓ 사용자 요청 검증 방법
    - JSR-303 Spec에서 제공하는 Annotation을 이용하여 검증하는 방법
    - Validator 구현 클래스 와 @InitBinder를 이용하여 사용자의 요청 데이터를 복합적으로 검증하는 방법

Validation Check

- Spring Framework에서의 데이터 검증
  - ✓ 일반적인 유효성 검사의 문제점
    - 계층 별로 진행하는 유효성 검사는 검증 로직이 각 클래스 별로 분산돼 있어 관리하기가 어렵고 검증 로직에 의외로 중복이 많아 여러 곳에 유사한 기능의 코드가 존재할 수 있으며 검증해야 할 값이 많다면 검증하는 코드가 길어지기 때문에 코드가 복잡해지고 가독성이 떨어짐
    - 자바 진영에서는 2009년부터 Bean Validation이라는 데이터 유효성 검사 프레임워크를 제공
    - Bean Validation은 어노테이션을 통해 다양한 데이터를 검증하는 기능을 제공
    - Bean Validation을 사용한다는 것은 유효성 검사를 위한 로직을 DTO 같은 도메인 모델과 묶어서 각 계층에서 사용하면서 검증 자체를 도메인 모델에 얹는 방식으로 수행한다는 의미
    - Bean Validation은 어노테이션을 사용한 검증 방식이기 때문에 코드의 간결함도 유지
  - ✓ Hibernate Validator
    - Hibernate Validator는 Bean Validation 명세의 구현체
    - 스프링 부트에서는 Hibernate Validator를 유효성 검사 표준으로 채택해서 사용
    - Hibernate Validator는 JSR-303 명세의 구현체로서 도메인 모델에서 어노테이션을 통한 필드 값 검증을 가능하게 도와줌

Validation Check

- Spring Framework에서의 데이터 검증
  - ✓ 프로젝트 생성



Validation Check

- Spring Framework에서의 데이터 검증
  - ✓ application.properties 파일을 삭제하고 application.yml 파일을 생성하고 작성

```
server:
  port: 80

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/adam
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: wnddkd

  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        format_sql: true
        show_sql: true

logging:
  level:
    org.hibernate.type.descriptor.sql: trace
```

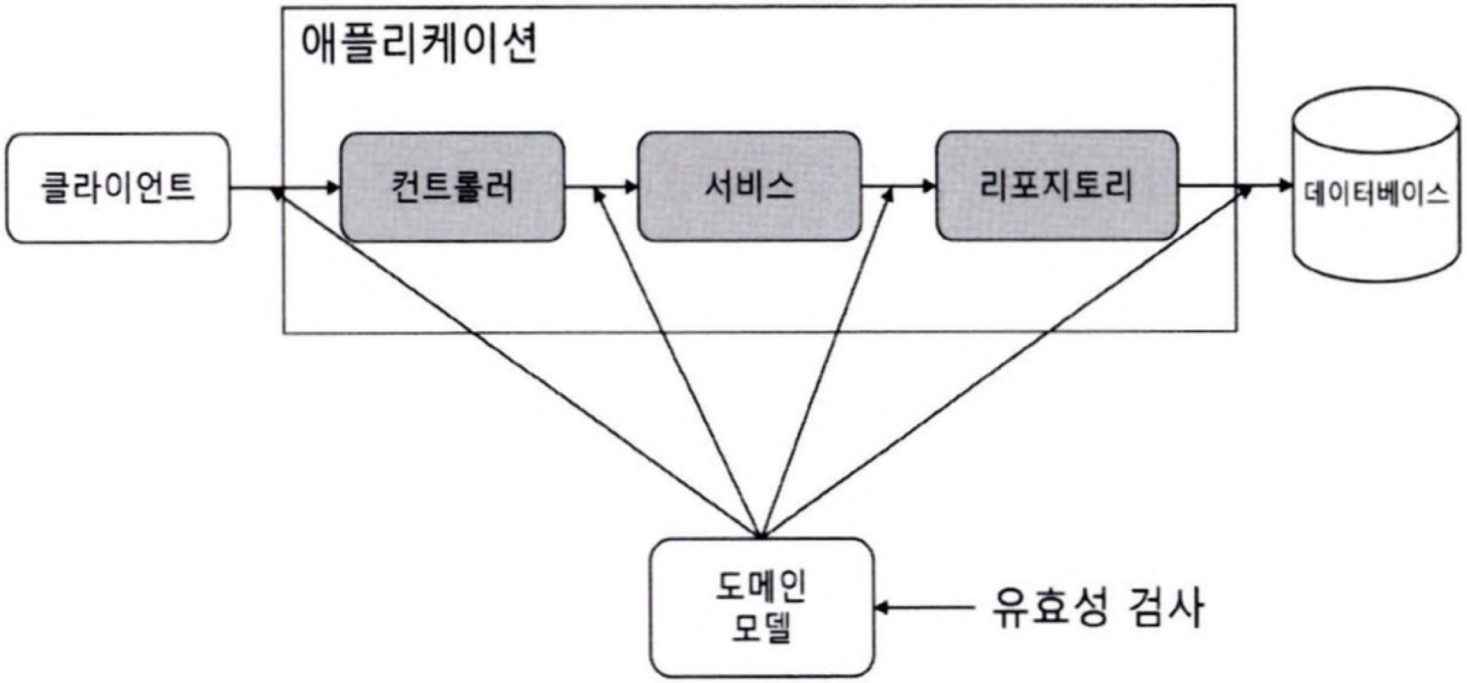
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ 유효성 검사를 위한 유효성 검사 라이브러리 의존성 추가

    implementation 'org.springframework.boot:spring-boot-starter-validation'

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ 유효성 검사는 각 계층으로 데이터가 넘어오는 시점에 해당 데이터에 대한 검사를 실시
  - ✓ 스프링 부트 프로젝트에서는 계층 간 데이터 전송에 대체로 DTO 객체를 활용하고 있기 때문에 유효성 검사를 DTO 객체를 대상으로 수행하는 것이 일반적



Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ 유효성 검사를 위한 어노테이션
    - 문자열 검증
      - @Null: null 값만 허용
      - @NotNull: null을 허용하지 않고 “”, “ “는 허용
      - @NotEmpty: null, “”을 허용하지 않지만 " "는 허용
      - @NotBlank: null, “”, “ “을 허용하지 않음
    - 최댓값/최솟값 검증
      - BigDecimal, Biginteger, int, long 등의 타입을 지원
      - @DemicalMax(value = "$numberString"): $numberString보다 작은 값을 허용
      - @DemicalMin(value = "$numberString"): $numberString보다 큰 값을 허용
      - @Min(value = $number): $number 이상의 값을 허용
      - @Max(value = $number): $number 이하의 값을 허용

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ 유효성 검사를 위한 어노테이션
    - 값의 범위 검증
      - BigDecimal, Biginteger, int, long 등의 타입을 지원
      - @Positive: 양수를 허용
      - @PositiveOrZero: 0을 포함한 양수를 허용
      - @Negative: 음수를 허용
      - @NegativeOrZero: 0을 포함한 음수를 허용
    - 시간에 대한 검증
      - Date, LocalDate, LocalDateTime 등의 타입을 지원
      - @Future: 현재보다 미래의 날짜를 허용
      - @FutureOrPresent: 현재를 포함한 미래의 날짜를 허용
      - @Past: 현재보다 과거의 날짜를 허용
      - @PastOrPresent: 현재를 포함한 과거의 날짜를 허용

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ 유효성 검사를 위한 어노테이션
    - 이메일 검증
      - @Email: 이메일 형식을 검사하는데 ""는 허용
    - 자릿수 범위 검증
      - BigDemical. Biginteger, int, long 등의 타입을 지원
      - @Digits(integer = $number1, fraction = $number2): $number1 의 정수 자릿수와 $number2의 소수 자릿수를 허용
    - Boolean 검증
      - @AssertTrue: true인지 체크하는데 null 값은 체크하지 않음
      - @AssertFalse: false인지 체크하는데 null 값은 체크하지 않음
    - 문자열 길이 검증
      - @Size(min = $number1, max = $number2): $number1 이상 $number2 이하의 범위를 허용
    - 정규식 검증
      - @Pattern (regexp = "$expression"): 정규식을 검사하는데 정규식은 자바의 java.util.regex.Pattern 패키지의 컨벤션을 따름

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ DTO 와 Controller 클래스 생성
    - 유효성 검사를 위한 DTO 클래스 생성 – dto.ValidRequestDTO

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Builder
public class ValidRequestDTO{

    @NotBlank
    private String name;

    @Email
    private String email;

    @Pattern(regexp = "01(?:0|1|[6-9])[.-]?(₩₩d{3}|₩₩d{4})[.-]?(₩₩d{4})$")
    private String phoneNumber;

    @Min(value = 20)
    @Max(value = 40)
    private int age;
```

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ DTO 와 Controller 클래스 생성
    - 유효성 검사를 위한 DTO 클래스 생성 – dto.ValidationRequestDTO

```
    @Size(min = 0, max = 40)
    private String description;

    @Positive
    private int count;

    @AssertTrue
    private boolean booleanCheck;

}
```

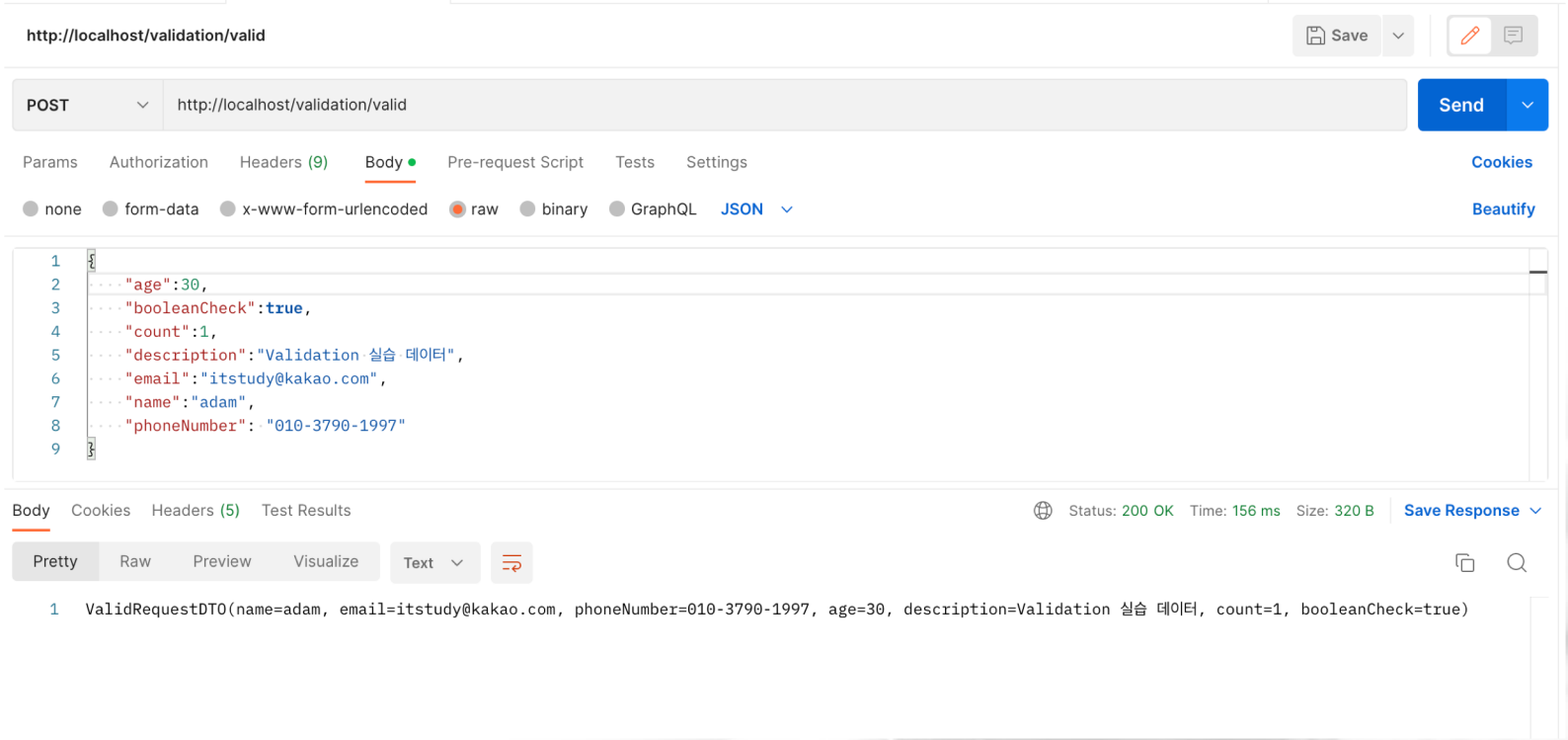
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ DTO 와 Controller 클래스 생성
    - 유효성 검사를 위한 Controller 클래스 생성 – controller.ValidationController

```
@RestController
@RequestMapping("/validation")
public class ValidationController {

    private final Logger LOGGER =
     LoggerFactory.getLogger(ValidationController.class);

    @PostMapping("/valid")
    public ResponseEntity<String> checkValidationByValid(
            @Valid @RequestBody ValidRequestDTO validRequestDTO) {
        LOGGER.info(validRequestDTO.toString());
        return
     ResponseEntity.status(HttpStatus.OK).body(validRequestDTO.toString());
    }
}
```

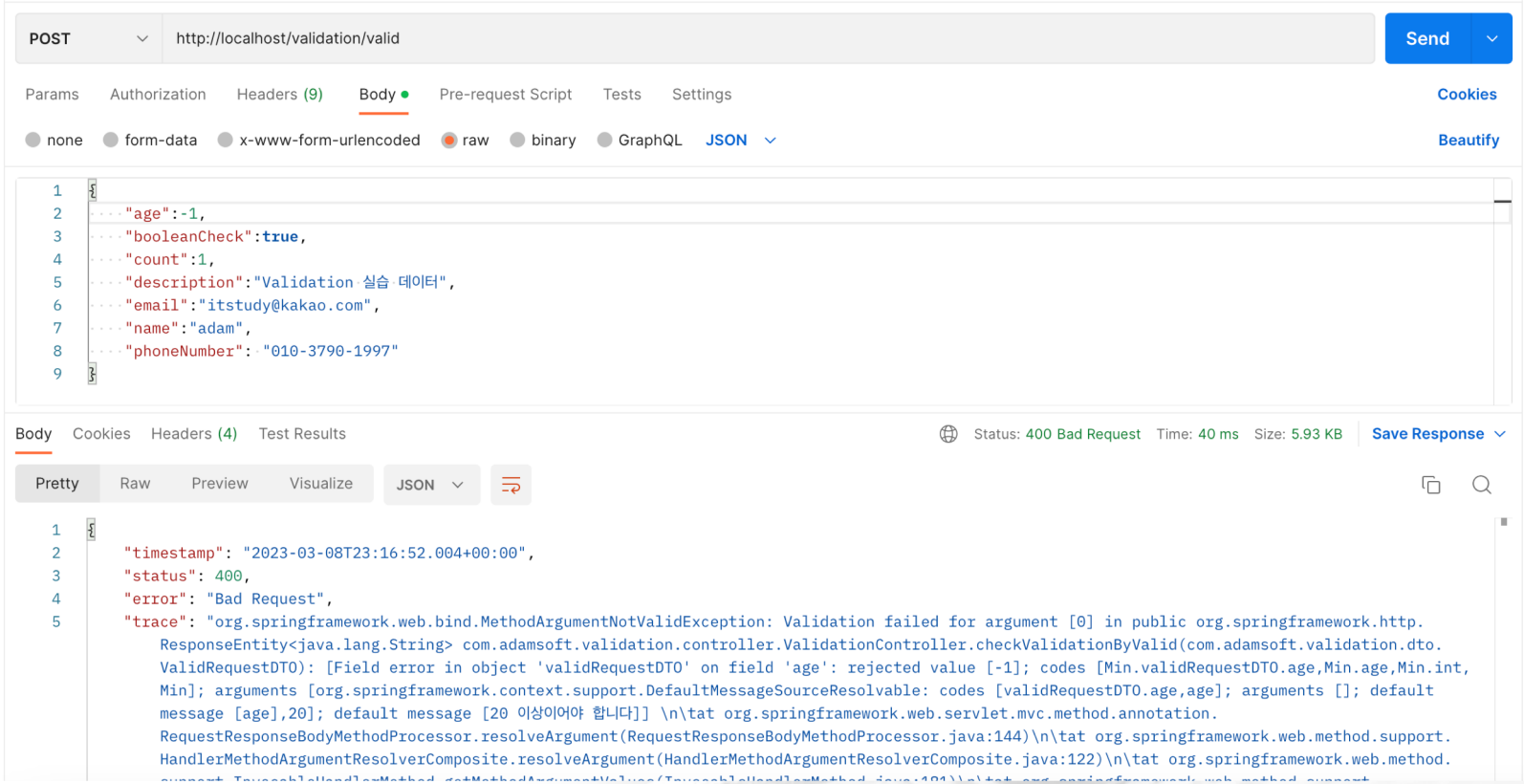
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ DTO 와 Controller 클래스 생성
    - 확인



Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ DTO 와 Controller 클래스 생성
    - 확인



Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - @Valid 어노테이션은 자바에서 지원하는 어노테이션이며 스프링도 @Validated라는 별도의 어노테이션으로 유효성 검사를 지원
    - @Validated은 @Valid 어노테이션의 기능을 포함하고 있기 때문에 @Validated로 변경할 수 있으며 @Validated는 유효성 검사를 그룹으로 묶어 대상을 특정할 수 있는 기능이 있dma

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - 그룹화를 위한 인터페이스 생성 – group.ValidationGroup1

      ```
      public interface ValidationGroup1 {

      }
      ```

    - 그룹화를 위한 인터페이스 생성 – group.ValidationGroup2

      ```
      public interface ValidationGroup2 {

      }
      ```

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - @Validated를 활용한 DTO 생성 – dto.ValidationGroup1

```
@Data
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Builder
public class ValidatedRequestDTO {

    @NotBlank
    private String name;

    @Email
    private String email;

    @Pattern(regexp = "01(?:0|1|[6-9])[.-]?(₩₩d{3}|₩₩d{4})[.-]?(₩₩d{4})$")
    private String phoneNumber;

    @Min(value = 20, groups = ValidationGroup1.class)
    @Max(value = 40, groups = ValidationGroup1.class)
    private int age;
```

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - @Validated를 활용한 DTO 생성 – dto.ValidationGroup1

```
    @Size(min = 0, max = 40)
    private String description;

    @Positive(groups = ValidationGroup2.class)
    private int count;

    @AssertTrue
    private boolean booleanCheck;
}
```

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - @Validated를 활용 위한 메서드를 ValidationController에 추가

```
@PostMapping("/validated")
public ResponseEntity<String> checkValidation(
        @Validated @RequestBody ValidatedRequestDTO validatedRequestDTO)
    {
        LOGGER.info(validatedRequestDTO.toString());
        return
     ResponseEntity.status(HttpStatus.OK).body(validatedRequestDTO.toString());
    }

@PostMapping("/validated/group1")
public ResponseEntity<String> checkValidation1(
        @Validated(ValidationGroup1.class) @RequestBody
     ValidatedRequestDTO validatedRequestDTO) {
        LOGGER.info(validatedRequestDTO.toString());
        return
     ResponseEntity.status(HttpStatus.OK).body(validatedRequestDTO.toString());
}
```

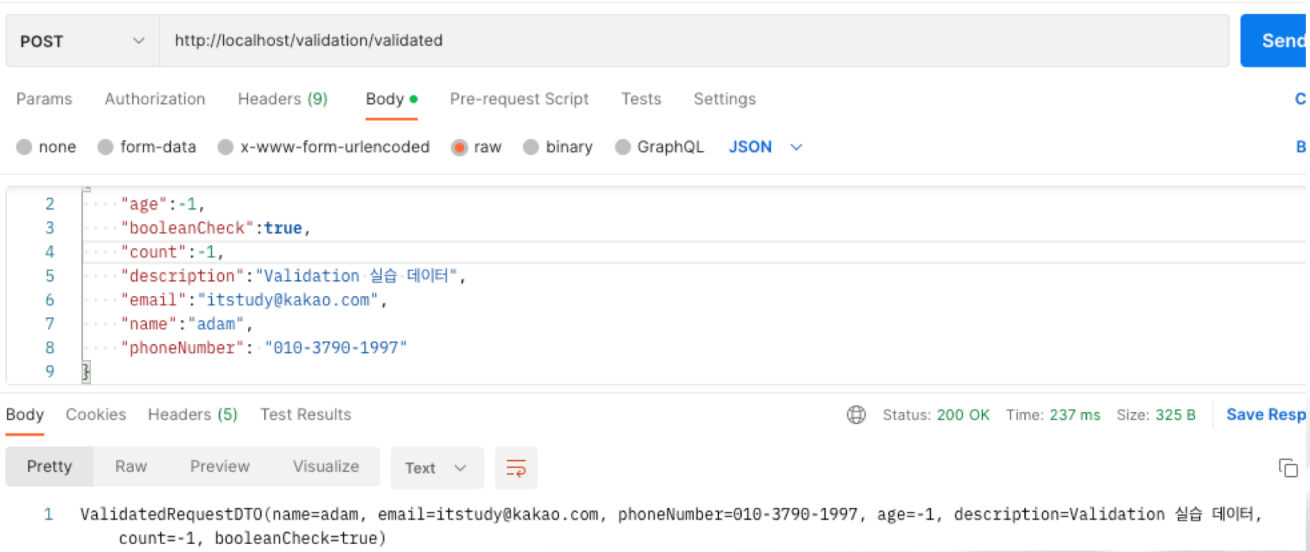
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - @Validated를 활용 위한 메서드를 ValidationController에 추가

```
@PostMapping("/validated/group2")
public ResponseEntity<String> checkValidation2(
        @Validated(ValidationGroup2.class) @RequestBody
 ValidatedRequestDTO validatedRequestDTO) {
    LOGGER.info(validatedRequestDTO.toString());
    return
 ResponseEntity.status(HttpStatus.OK).body(validatedRequestDTO.toString());
}

@PostMapping("/validated/all-group")
public ResponseEntity<String> checkValidation3(
        @Validated({ValidationGroup1.class,
                ValidationGroup2.class}) @RequestBody ValidatedRequestDTO
 validatedRequestDTO) {
    LOGGER.info(validatedRequestDTO.toString());
    return
 ResponseEntity.status(HttpStatus.OK).body(validatedRequestDTO.toString());
}
```

Validation Check

- ❖ Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - 확인 - validated

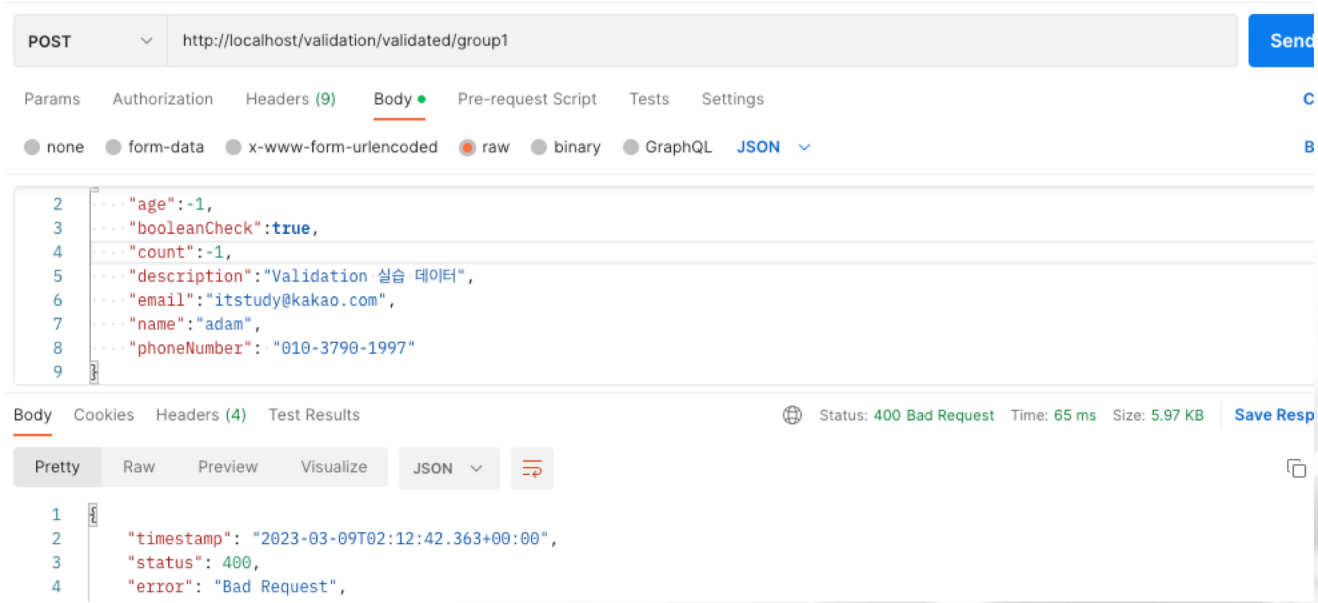


Validation Check

- ❖ Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - ● 확인 – validated
      - ◆ age와 count 변수에 대한 유효성 검사를 통과하지 못하는 데이터 지만 호출했을 경우 정상적으로 통과
      - ◆ @Validated 어노테이션에 특정 그룹을 지정하지 않는 경우에는 groups 속성을 설정하지 않은 필드에 대해서만 유효성 검사를 실시하기 때문

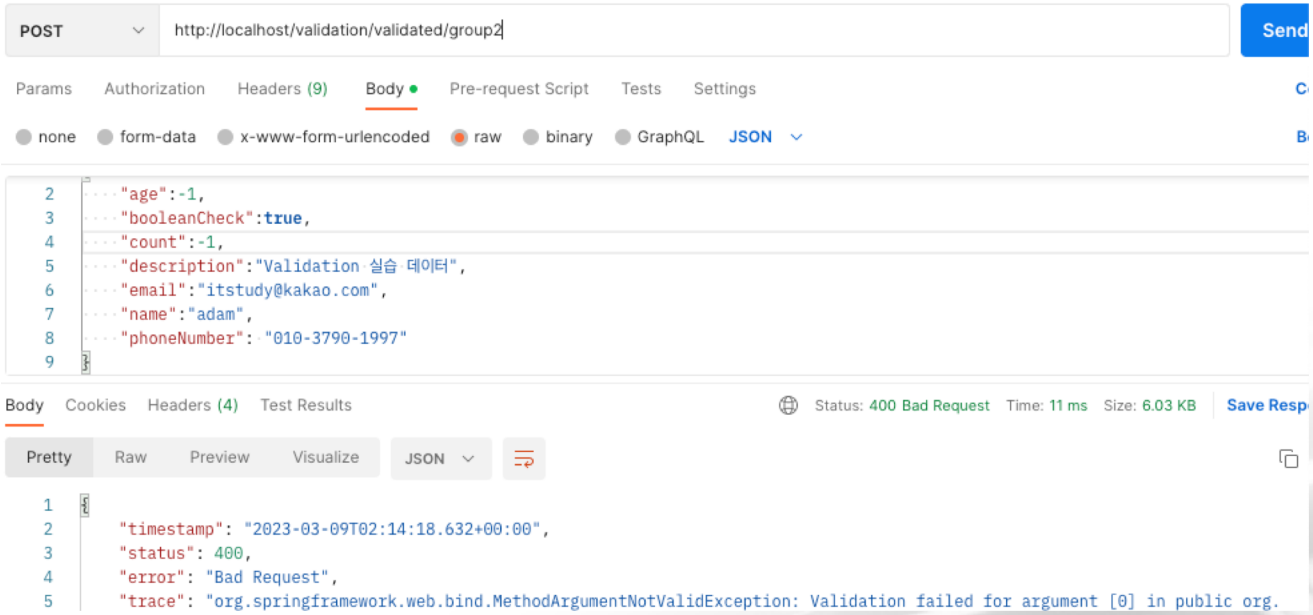
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - 확인 – validated/group1: age에 대해서 오류



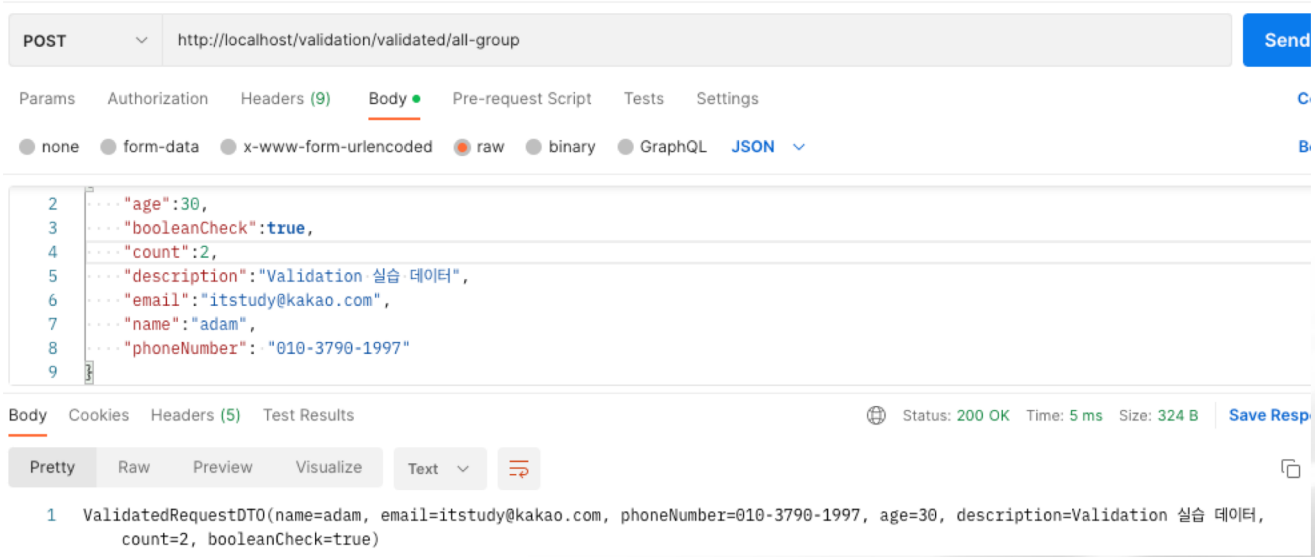
Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - 확인 – validated/group2: count에 대해서 오류



Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ @Validated 활용
    - 확인 – validated/all-group



Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - 실무에서는 유효성 검사를 실시할 때 자바 또는 스프링의 유효성 검사 어노테이션에서 제공하지 않는 기능을 써야 할 때도 있는데 이 경우 ConstraintValidator 와 커스텀 어노테이션을 조합해서 별도의 유효성 검사 어노테이션을 생성할 수 있음
    - 동일한 정규식을 계속 쓰는 @Pattern 어노테이션의 경우가 가장 흔한 사례

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - 휴대폰 번호를 위한 정규식 검사 클래스 – validator.TelephoneValidator

```
public class TelephoneValidator implements ConstraintValidator<Telephone,
    String> {

    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if(value==null){
            return false;
        }
        return value.matches("01(?:0|1|[6-9])[.-]?(₩₩d{3}|₩₩d{4})[.-]?(₩₩d{4})$");
    }
}
```

- ◆ TelephoneValidator 클래스를 Constraintvalidator 인터페이스의 구현체로 정의
- ◆ 인터페이스를 선언할 때는 어떤 어노테이션 인터페이스인지 타입을 지정해야 함
- ◆ ConstraintValidator 인터페이스는 invalid() 메서드를 정의하는데 구현하는 경우 직접 유효성 검사 로직을 작성해야 함
- ◆ null에 대한 허용 여부 로직을 추가하고 지정한 정규식과 비교해서 알맞은 형식을 띠고 있는지 검사하는데 false가 리턴되면 MethodArgumentNotValidException 예외 발생

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - 휴대폰 번호를 위한 어노테이션 인터페이스 – validator.Telephone

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = TelephoneValidator.class)
public @interface Telephone {
    String message() default "전화번호 형식이 일치하지 않습니다.";
    Class[] groups() default {};
    Class[] payload() default {};
}
```

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - ● 휴대폰 번호를 위한 어노테이션 인터페이스 – validator.Telephone
      - ◆ @Target: 어디서 선언할 수 있는지 정의
        - ElementType.PACKAGE
        - ElementType.TYPE
        - ElementType.CONSTRUCTOR
        - ElementType.FIELD
        - ElementType.METHOD
        - ElementType.ANNOTATION_TYPE
        - ElementType.LOCAL.VARIABLE
        - ElementType.PARAMETER
        - ElementType.TYPE.PARAMETER
        - ElementType.TYPE_USE

Validation Check

- ❖ Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - ● 휴대폰 번호를 위한 어노테이션 인터페이스 – validator.Telephone
      - ◆ @Retention: 적용되고 유지되는 범위
        - ▪ RetentionPolicy.RUNTIME: 컴파일 이후에도 JVM에 의해 계속 참조하는데 리플렉션이나 로깅에 많이 사용되는 정책
        - ▪ RetentionPolicy.CLASS: 컴파일러가 클래스를 참조할 때까지 유지
        - ▪ RetentionPolicy.SOURCE: 컴파일 전까지만 유지되고 컴파일 이후에는 사라짐
      - ◆ @Constraint 어노테이션을 활용해 TelephoneValidator와 매핑하는 작업을 수행
      - ◆ 속성
        - ▪ message(): 유효성 검사가 실패할 경우 반환되는 메시지를 의미
        - ▪ groups(): 유효성 검사를 사용하는 그룹으로 설정
        - ▪ payload(): 사용자가 추가 정보를 위해 전달하는 값

Validation Check

- Spring Boot 에서의 유효성 검사: JSR-303
  - ✓ Custom Validation
    - ● ValidatedRequestDTO 클래스의 phoneNumber 의 유효성 검사 수정

```
//@Pattern(regexp = "01(?:0|1|[6-9])[.-]?(₩₩d{3}|₩₩d{4})[.-]?(₩₩d{4})$")
@Telephone
private String phoneNumber;
```

Validation Check

- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 여러 조합으로 구성되는 유효성을 어노테이션 만으로 검증하기가 쉽지 않음
  - ✓ org.springframework.validator.Validator 인터페이스를 제공하는데 이 인터페이스에는 2개의 메서드가 존재
    - boolean supports(Class<?> clazz): 제공하는 클래스가 검증할 수 있는 클래스 인지 확인하는 메서드
    - void validate(Object target, Errors errors): 검증하고자 하는 객체는 target 으로 전달되는데 이를 캐스팅해서 검증하는 코드를 작성하는 메서드인데 검증 에러가 있다면 Errors errors 객체의 메서드를 사용하여 에러를 입력
  - ✓ 인터페이스의 구현체는 검증하고자 하는 데이터의 DTO 클래스 와 1:1로 매칭시키는 경우가 많음

Validation Check

- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 유효성 검사를 위한 DTO 클래스 생성 – dto.HotelRoomReserveRequestDTO

```
@Getter
@ToString
public class HotelRoomReserveRequestDTO {

    private LocalDate checkInDate;

    private LocalDate checkOutDate;

    private String name;
}
```

Validation Check

- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 유효성 검사를 위한 Validator 클래스 생성 – dto.HotelRoomReserveValidator

```
public class HotelRoomReserveValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return HotelRoomReserveRequestDTO.class.equals(clazz);
    }
```

Validation Check

- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 유효성 검사를 위한 Validator 클래스 생성 – validator.HotelRoomReserveValidator

```
@Override
public void validate(Object target, Errors errors) {
    HotelRoomReserveRequestDTO request =
HotelRoomReserveRequestDTO.class.cast(target);

    if (Objects.isNull(request.getCheckInDate())) {
        errors.rejectValue("checkInDate", "NotNull", "checkInDate is null");
        return;
    }
    if (Objects.isNull(request.getCheckOutDate())) {
        errors.rejectValue("checkOutDate", "NotNull", "checkOutDate is null");
        return;
    }
    if (request.getCheckInDate().compareTo(request.getCheckOutDate()) >= 0) {
        errors.rejectValue("checkOutDate", "Constraint Error", "checkOutDate is earlier
than checkInDate ");
        return;
    }
  }
}
```

Validation Check

- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 유효성 검사를 위한 Controller 클래스 생성 – controller.HotelRoomReserveController

```
@RestController
public class HotelRoomReserveController {

    @InitBinder
    void initBinder(WebDataBinder binder) {
        binder.addValidators(new HotelRoomReserveValidator());
    }
```

Validation Check

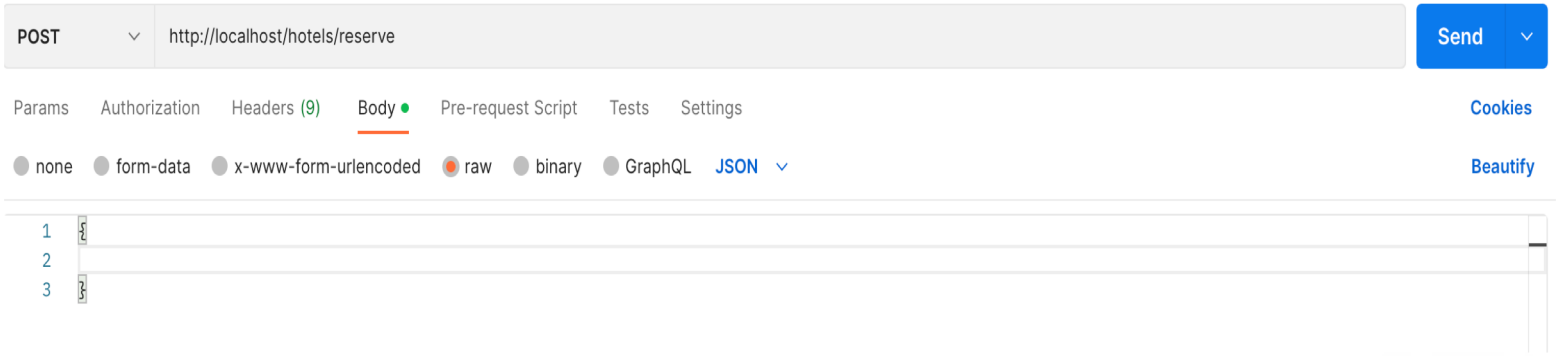
- Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증
  - ✓ 유효성 검사를 위한 Controller 클래스 생성 – controller.HotelRoomReserveController

```
@PostMapping(path = "/hotels/reserve")
public ResponseEntity<?> reserveHotelRoomByRoomNumber(
        @Valid @RequestBody HotelRoomReserveRequestDTO reserveRequest,
        BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        FieldError fieldError = bindingResult.getFieldError();
        String errorMessage = new
 StringBuilder(bindingResult.getFieldError().getCode())
                .append(" [").append(fieldError.getField()).append("] ")
                .append(fieldError.getDefaultMessage())
                .toString();

        System.out.println("error  : " + errorMessage);
        return ResponseEntity.badRequest().build();
    }
    System.out.println("유효성 검사 통과");
    return null;
}
}
```

Validation Check

❖ Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증

✓ 확인

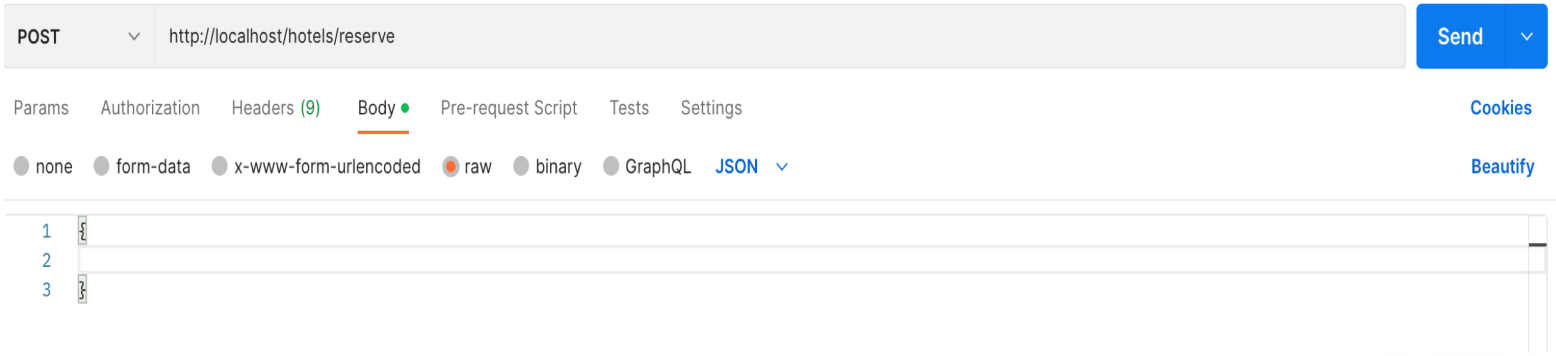


error : NotNull [checkInDate] checkInDate is null

Validation Check

❖ Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증

✓ 확인

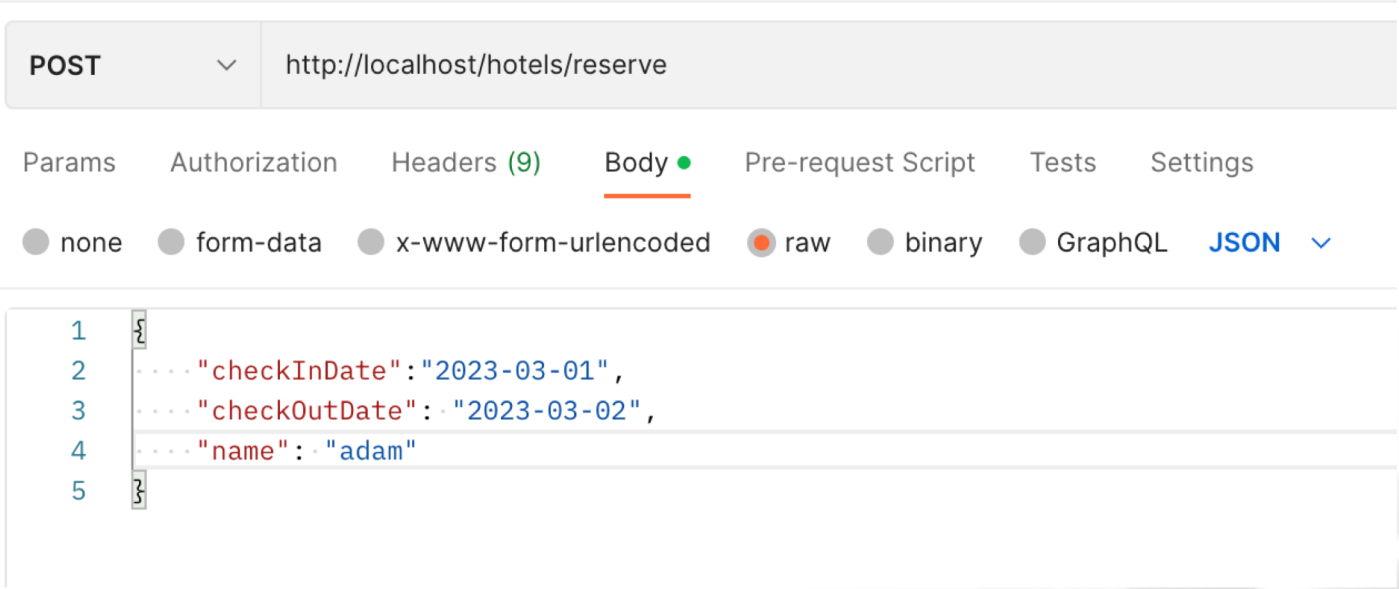


error : NotNull [checkInDate] checkInDate is null

Validation Check

❖ Spring Boot 에서의 유효성 검사: Validator 인터페이스를 이용한 검증

✓ 확인



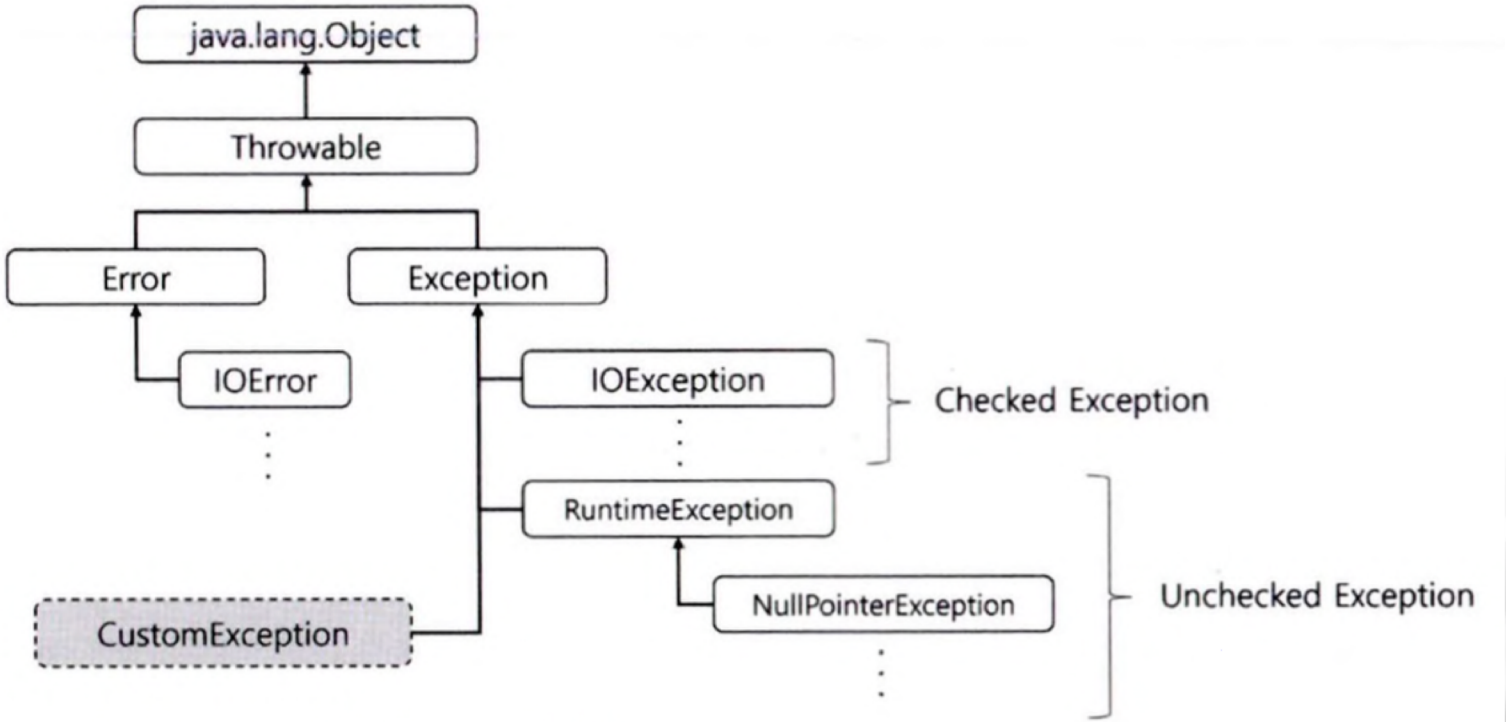
유효성 검사 통과

Exception Handling

- 예외 와 에러
  - ✓ 프로그래밍에서 예외(exception)란 입력 값의 처리가 불가능하거나 참조된 값이 잘못된 경우 등 애플리케이션이 정상적으로 동작하지 못하는 상황을 의미
  - ✓ 예외는 개발자가 직접 처리할 수 있는 것이므로 미리 코드 설계를 통해 처리할 수 있음
  - ✓ 에러는 주로 자바의 가상 머신에서 발생시키는 것으로서 예외와 달리 애플리케이션 코드에서 처리할 수 있는 것이 거의 없는데 대표적인 예로 메모리 부족(OutOfMemory), 스택 오버플로(StackOverFlow)이 있으며 이러한 에러는 발생 시점에 처리하는 것이 아니라 미리 애플리케이션의 코드를 살펴보면서 문제가 발생하지 않도록 예방해서 원천적으로 차단해야 함

Exception Handling

❖ 예외 클래스



Exception Handling

❖ 예외 클래스

✓ Exception 분류

	Checked Exception	Unchecked Exception
처리 여부	반드시 예외 처리 필요	명시적 처리를 강제하지 않음
확인 시점	컴파일 단계	실행 중 단계
대표적인 예외 클래스	IOException SQLException	RuntimeException NullPointerException IllegalArgumentException IndexOutOfBoundException SystemException

- Checked Exception은 컴파일 단계에서 확인 가능한 예외 상황으로 IDE에서 캐치해서 반드시 예외 처리를 할 수 있게 표시해주는 반면 Unchecked Exception은 런타임 단계에서 확인되는 예외 상황으로 문법상 문제는 없지만 프로그램이 동작하는 도중 예기치 않은 상황이 생겨 발생하는 예외를 의미
- RuntimeException을 상속받는 Exception 클래스는 Unchecked Exception이고 그렇지 않은 Exception 클래스는 Checked Exception

Exception Handling

- ❖ 예외 처리 방법
  - ✓ 처리 방법
    - ● 예외 복구
      - ◆ try ~ catch ~ finally를 이용해서 예외가 발생했을 때 처리할 내용을 작성
    - ● 예외 처리 회피
      - ◆ throws를 이용해서 예외가 발생하면 호출하는 곳에 예외를 전달
    - ● 예외 전환
      - ◆ 예외가 발생했을 때 어떤 예외가 발생했느냐에 따라 호출부로 예외 내용을 전달하면서 좀 더 적합한 예외 타 입으로 전달할 필요가 있고 애플리케이션에서 예외 처리를 좀 더 단순하게 하기 위해 래핑(wrapping)해야 하는 경우도 있는데 이런 경우에는 try/catch 방식을 사용하면서 catch 블록에서 throw 키워드를 사용해 다른 예외 타입으로 전달
      - ◆ 커스텀 예외를 만드는 과정에서 사용되는 방법

Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ 웹 서비스 애플리케이션에서는 외부에서 들어오는 요청에 담긴 데이터를 처리하는 경우가 많은데 그 과정에서 예외가 발생하면 예외를 복구해서 정상으로 처리하기보다는 요청을 보낸 클라이언트에 어떤 문제가 발생했는지 상황을 전달하는 경우가 많음
  - ✓ 예외가 발생했을 때 클라이언트에 오류 메시지를 전달하려면 각 레이어에서 발생한 예외를 엔드 포인트 레벨인 컨트롤러로 전달해야 함
  - ✓ 스프링 부트에서 처리하는 방식
    - (Rest)ControllerAdvice 와 @ExceptionHandler를 통해 모든 컨트롤러의 예외를 처리
    - @ExceptionHandler를 통해 특정 컨트롤러의 예외를 처리

Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ 웹 서비스 애플리케이션에서는 외부에서 들어오는 요청에 담긴 데이터를 처리하는 경우가 많은데 그 과정에서 예외가 발생하면 예외를 복구해서 정상으로 처리하기보다는 요청을 보낸 클라이언트에 어떤 문제가 발생했는지 상황을 전달하는 경우가 많음
  - ✓ 예외가 발생했을 때 클라이언트에 오류 메시지를 전달하려면 각 레이어에서 발생한 예외를 엔드 포인트 레벨인 컨트롤러로 전달해야 함
  - ✓ 스프링 부트에서 처리하는 방식
    - (Rest)ControllerAdvice 와 @ExceptionHandler를 통해 모든 컨트롤러의 예외를 처리
    - @ExceptionHandler를 통해 특정 컨트롤러의 예외를 처리

Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ RestControllerAdvice를 이용한 예외 처리
    - ● 예외 처리 클래스를 생성 – exception.CustomExceptionHandler

```
@RestControllerAdvice
public class CustomExceptionHandler {

    private final Logger LOGGER =
     LoggerFactory.getLogger(CustomExceptionHandler.class);
```

Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ RestControllerAdvice를 이용한 예외 처리
    - 예외 처리 클래스를 생성 – exception.CustomExceptionHandler

```
//처리할 예외 클래스를 지정해서 Controller에서 예외가 발생했을 때 호출
@ExceptionHandler(value = RuntimeException.class)
public ResponseEntity<Map<String, String>>
 handleException(RuntimeException e,
    HttpServletRequest request) {
    HttpHeaders responseHeaders = new HttpHeaders();
    HttpStatus httpStatus = HttpStatus.BAD_REQUEST;

    LOGGER.error("Advice 내 exceptionHandler 호출, {}, {}",
 request.getRequestURI(),
        e.getMessage());

    Map<String, String> map = new HashMap<>();
    map.put("error type", httpStatus.getReasonPhrase());
    map.put("code", "400");
    map.put("message", e.getMessage());

    return new ResponseEntity<>(map, responseHeaders, httpStatus);
  }
}
```

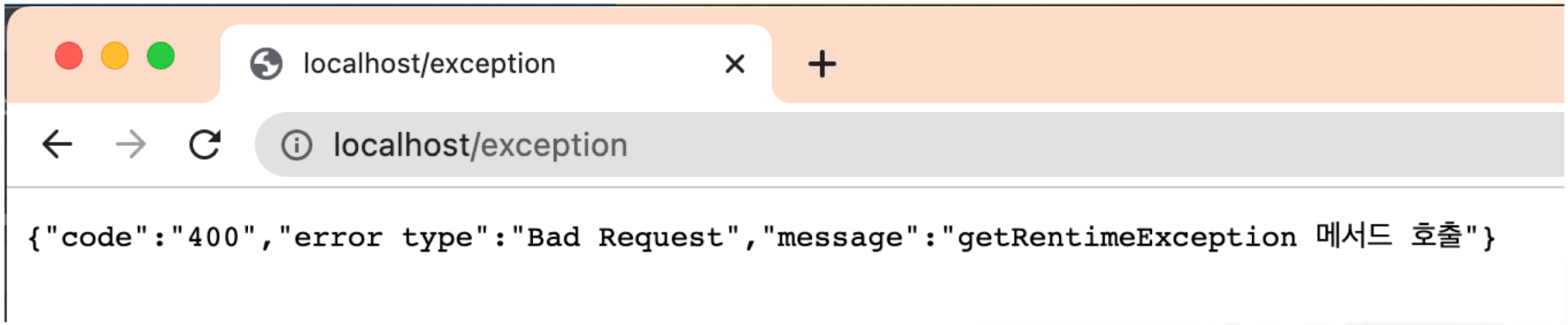
Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ RestControllerAdvice를 이용한 예외 처리
    - Controller 클래스를 생성 – controller. ExceptionController

```
@RestController
@RequestMapping("/exception")
public class ExceptionController {
    @GetMapping
    public void getRuntimeException(){
        throw new RuntimeException("getRentimeException 메서드 호출");
    }
}
```

Exception Handling

- ❖ Spring Boot 의 예외 처리 방식
  - ✓ RestControllerAdvice를 이용한 예외 처리
    - ● 테스트



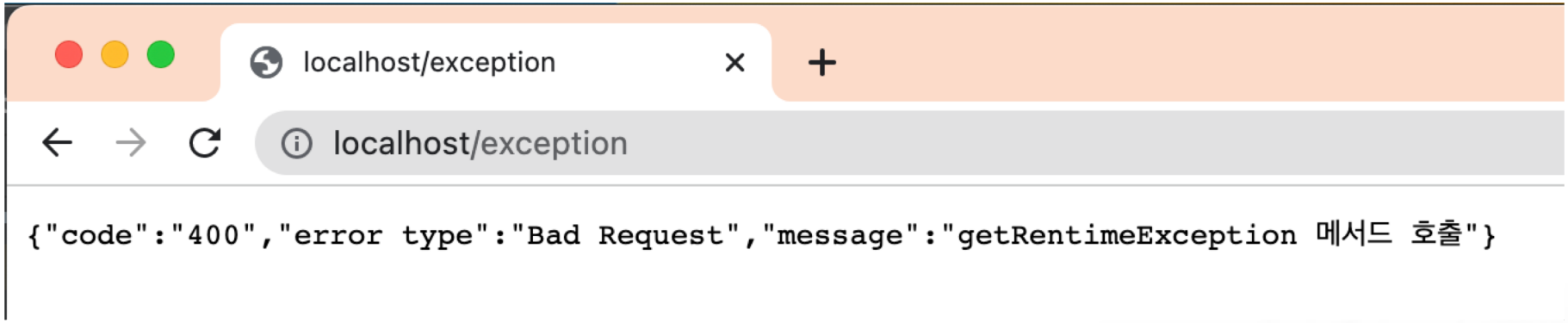
Exception Handling

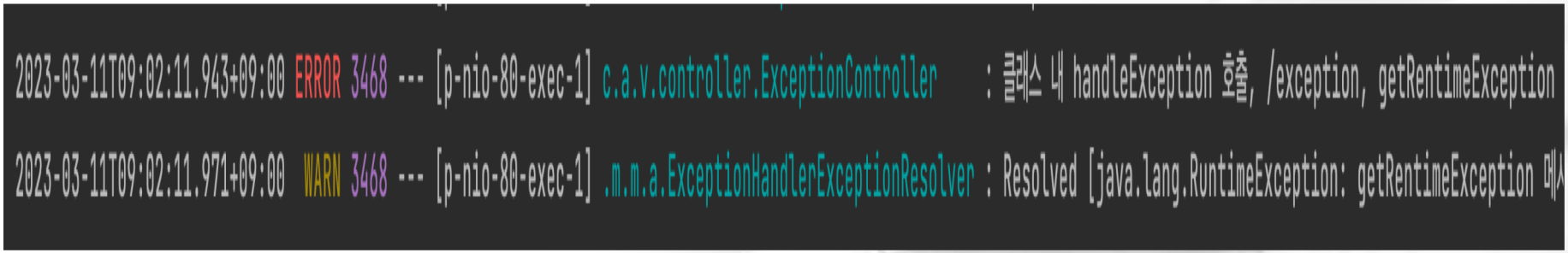
- Spring Boot 의 예외 처리 방식
  - ✓ Controller 내에서의 예외 처리
    - ExceptionController에 코드 추가

```
@ExceptionHandler(value = RuntimeException.class)
public ResponseEntity<Map<String, String>> handleException(RuntimeException
    e,
                                                    HttpServletRequest request) {
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setContentType(MediaType.APPLICATION_JSON);
    HttpStatus httpStatus = HttpStatus.BAD_REQUEST;

    LOGGER.error("클래스 내 handleException 호출, {}, {}",
    request.getRequestURI(),
            e.getMessage());

    Map<String, String> map = new HashMap<>();
    map.put("error type", httpStatus.getReasonPhrase());
    map.put("code", "400");
    map.put("message", e.getMessage());

    return new ResponseEntity<>(map, responseHeaders, httpStatus);
}
```

Exception Handling

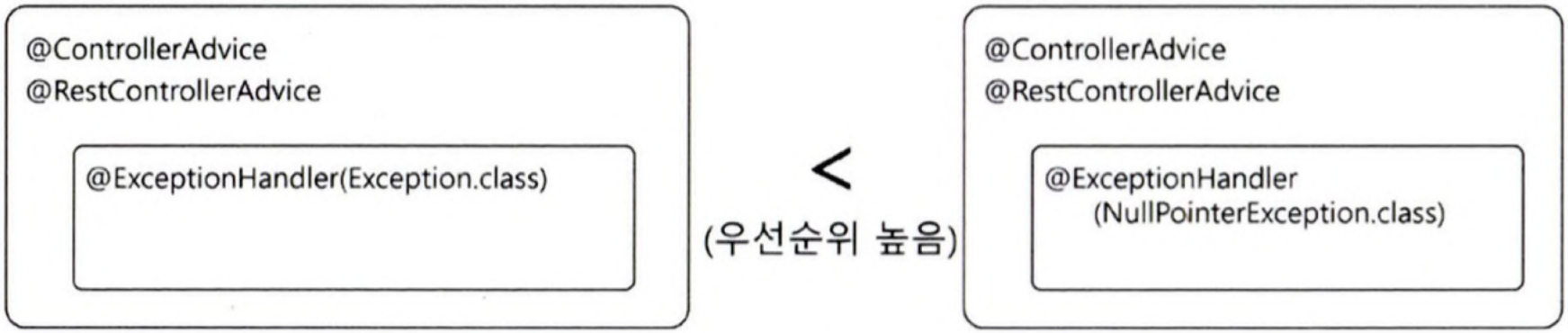
- ❖ Spring Boot 의 예외 처리 방식
  - ✓ RestControllerAdvice를 이용한 예외 처리
    - 테스트





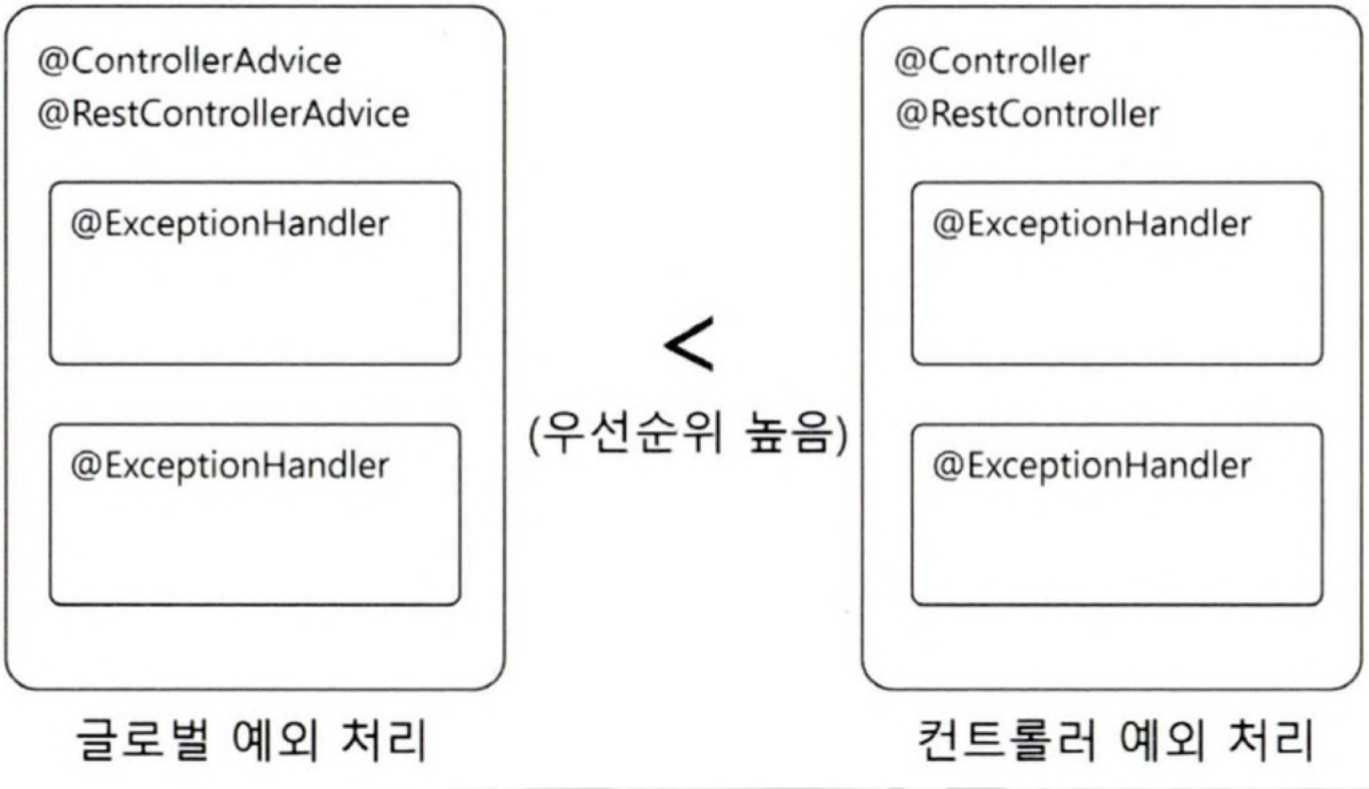
Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ 예외 처리의 우선 순위
    - 예외 타입 레벨에 따른 예외 처리 우선 순위



Exception Handling

- Spring Boot 의 예외 처리 방식
  - ✓ 예외 처리의 우선 순위
    - 핸들러 위치에 따른 예외 처리 우선 순위



Exception Handling

- ❖ Custom 예외
  - ✓ Custom 예외를 사용하는 이유
    - 애플리케이션을 개발하다 보면 점점 예외로 처리할 영역이 늘어나고 예외 상황이 다양해지면서 사용하는 예외 타입도 많아지는데 대부분의 상황에서는 이미 적절한 상황에 사용할 수 있도록 제공하는 표준 예외(Standard Exception)를 사용하면 해결됨
    - 커스텀 예외를 만들어서 사용하면 네이밍에 개발자의 의도를 담을 수 있기 때문에 이름만으로도 어느 정도 예외 상황을 짐작할 수 있음
    - 표준 예외에서도 다양한 예외 상황을 처리할 수 있는 클래스를 제공하고 있지만 표준 예외에서 제공하는 클래스는 해당 예외 타입의 이름만으로 이해하기 어려운 경우가 있기 때문에 표준 예외를 사용할 때는 예외 메시지를 상세히 작성해야 하는 번거로움이 있음
    - 커스텀 예외를 사용하면 애플리케이션에서 발생하는 예외를 개발자가 직접 관리하기가 수월해지고 표준 예외를 상속받은 커스텀 예외들을 개발자가 직접 코드로 관리하기 때문에 책임 소재를 애플리케이션 내부로 가져올 수 있게 되고 이를 통해 동일한 예외 상황이 발생할 경우 한 곳에서 처리하며 특정 상황에 맞는 예외 코드를 적용할 수 있게 됨
    - 표준 예외를 사용하면 의도하지 않은 예외 상황도 정해진 예외 처리 코드에서 처리하기 때문에 어디에서 문제가 발생했는지 확인하기가 어렵지만 커스텀 예외로 관리하면 의도하지 않았던 부분에서 발생한 예외는 개발자가 관리하는 예외 처리 코드가 처리하지 않으므로 개발 과정에서 혼동할 여지가 줄어듬

Exception Handling

- Custom 예외
  - ✓ Custom 클래스
    - 커스텀 예외는 예외가 발생하는 상황에 해당하는 상위 예외 클래스를 상속받아서 생성
    - 커스텀 예외 는 상위 예외 클래스보다 좀 더 구체적인 이름을 사용

Exception Handling

- Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - Custom 예외 클래스를 위한 상수를 통합 관리하는 클래스 생성 – exception.Constants

```
public class Constants {
    public enum ExceptionClass {
        PRODUCT("Product");

        private String exceptionClass;

        ExceptionClass(String exceptionClass) {
            this.exceptionClass = exceptionClass;
        }

        public String getExceptionClass() {
            return exceptionClass;
        }

        @Override
        public String toString() {
            return getExceptionClass() + " Exception. ";
        }
    }
}
```

Exception Handling

- Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - Custom 예외 클래스 생성 – exception.CustomException

```
public class CustomException extends Exception{

    private static final long serialVersionUID = 43003333103792399 87L;

    private Constants.ExceptionClass exceptionClass;
    private HttpStatus httpStatus;

    public CustomException(Constants.ExceptionClass exceptionClass, HttpStatus
     httpStatus,
        String message) {
        super(exceptionClass.toString() + message);
        this.exceptionClass = exceptionClass;
        this.httpStatus = httpStatus;
    }
```

Exception Handling

- ❖ Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - ● Custom 예외 클래스 생성 – exception.CustomException

```
    public Constants.ExceptionClass getExceptionClass() {
        return exceptionClass;
    }

    public int getHttpStatusCode() {
        return httpStatus.value();
    }

    public String getHttpStatusType() {
        return httpStatus.getReasonPhrase();
    }

    public HttpStatus getHttpStatus() {
        return httpStatus;
    }

}
```

Exception Handling

- ❖ Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - ● CustomExceptionHandler 클래스에 메서드 추가

```
@ExceptionHandler(value = CustomException.class)
public ResponseEntity<Map<String, String>> handleException(CustomException e,
                                                           HttpServletRequest request) {
    HttpHeaders responseHeaders = new HttpHeaders();
    LOGGER.error("Advice 내 handleException 호출, {}, {}",
request.getRequestURI(),
            e.getMessage());
    Map<String, String> map = new HashMap<>();
    map.put("error type", e.getHttpStatusType()); map.put("code",
Integer.toString(e.getHttpStatusCode()));
    map.put("message", e.getMessage());
    return new ResponseEntity<>(map, responseHeaders, e.getHttpStatus());
}
```

Exception Handling

- ❖ Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - ● CustomController 클래스에 요청 처리 메서드 추가

```
@GetMapping("/custom")
public void getCustomException() throws CustomException {
        throw new CustomException(Constants.ExceptionClass.PRODUCT,
    HttpStatus.BAD_REQUEST, "getCustomException 메소드 호출");
}
```

Exception Handling

- Custom 예외
  - ✓ Custom 클래스를 이용한 예외 처리
    - 테스트