

# 정규 백엔드 스터디

6주차 – 유효성 검사, 예외처리, API 문서화

# 지난 주에는...

- 서비스, 컨트롤러 계층 구현
- Postman을 이용한 API 테스트

# 이번 주에는...

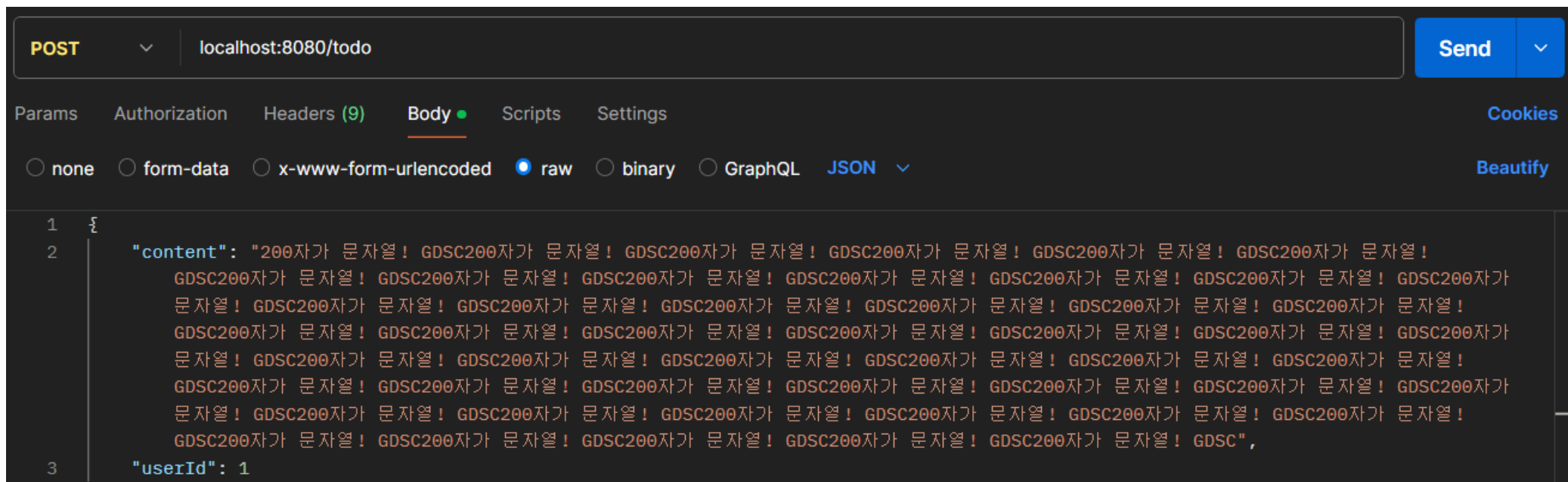
- 요청 데이터의 유효성 검증
- Global Exception Handler 로 예외 처리하기
- 예외 메시지 클래스 리팩토링
- API 문서화

# 유효성 검사

- 지금까지 만든 API 서버는 **의도한 요청**에 대해서만 처리한다.
- 하지만 실제로는 의도하지 않은 요청이 들어올 때도 많다.

# 유효성 검사

- 현재 테이블 제약에 명시된 content의 길이는 최대 100
- 만약 100보다 긴 문자열이 요청으로 들어온다면?



# 유효성 검사

- DB에 데이터를 넣을 때 에러가 발생한다.

```
org.h2.jdbc.JdbcSQLException: Value too long for column "TODO_CONTENT CHARACTER VARYING(200)":
```

- 결과적으로 길이가 100이 넘는 데이터는 저장되지 않으므로  
우리 서비스의 정책을 위반하지는 않는다.

# 유효성 검사 - 문제점

- 하지만 클라이언트는 무엇이 문제인지 알 수 없다.

```
{  
  "timestamp": "2024-11-12T11:38:40.776+00:00",  
  "status": 500,  
  "error": "Internal Server Error",  
  "path": "/todo"  
}
```

cf) 500에러는 문제의 원인이 클라이언트가 아닌 서버에 있다는 뜻을 나타냅니다. 클라이언트가 원인인 경우에는 4xx 상태코드를 명시적으로 응답해야 하므로, 지금 상황에서는 400과 같은 상태코드를 응답하는 것이 더 좋습니다.

5xx 상태 코드는 서버 환경에 문제가 발생했거나 (502 bad gateway 등) 개발자가 고려하지 않은 예외가 발생했다는 의미입니다. (심각한 상황)

# 유효성 검사 - 문제점

- 또한 데이터의 길이가 정책을 위반했다는 것은 문자열 길이 측정을 통해 스프링에서 사전에 확인할 수 있다.

하지만 지금은 데이터베이스까지 요청을 보내서 예외를 확인하므로 시간, 자원의 낭비가 발생한다.



# 유효성 검사

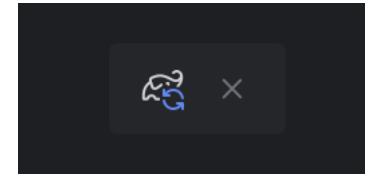
- 요청으로 들어오는 데이터가 **올바른 형식**인지 검사하는 것
- 스프링에서는 데이터를 받아들이는 DTO에서 유효성을 검사한다.  
(유효성 검사는 '형식' 만 검사한다. 존재하지 않는 멤버 아이디와 같은 경우는 유효성 검사로 체크할 수 없다.)

# 유효성 검사

- 유효성 검사를 도와줄 외부 의존성을 추가한다.

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

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.springframework.boot:spring-boot-starter-validation'  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.projectlombok:lombok'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
  
    // H2 DB  
    implementation 'com.h2database:h2'  
}
```



# 유효성 검사

- DTO 클래스에 제약 사항과 에러 메시지를 명시한다.

```
@Getter
public class TodoCreateRequest {

    @Length(max = 200, message = "할 일 내용은 200자를 넘을 수 없습니다.")
    private String content;

    @NotNull(message = "유저 ID는 필수입니다.")
    private Long userId;

}
```

```
@Min(20)
@Max(20)
@NotNull
@NotBlank
@email
@Size
```

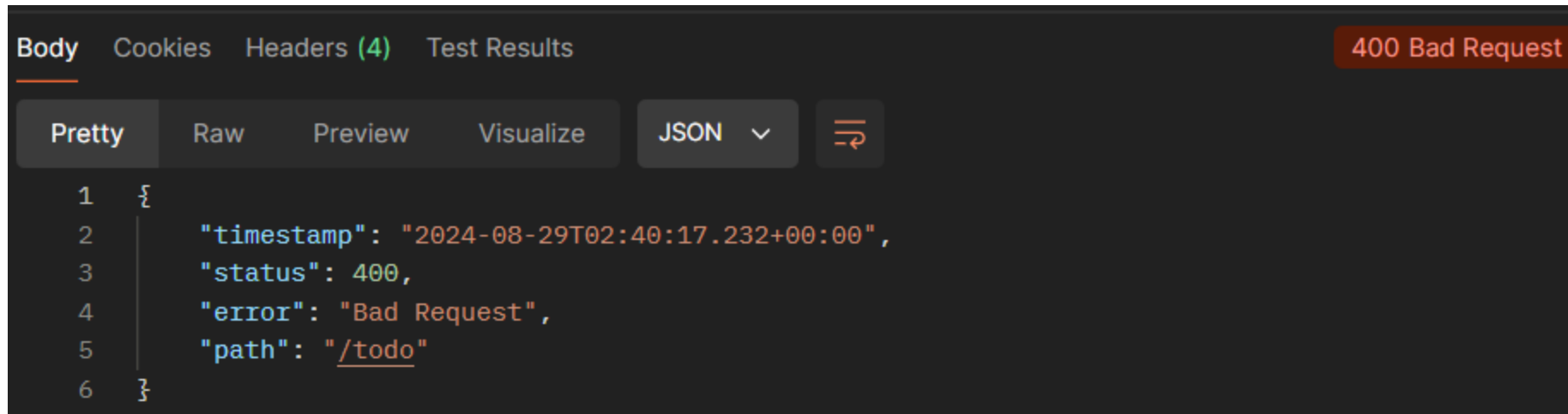
# 유효성 검사

- **@Valid** 를 사용하여 명시된 제약 조건에 맞는지 검사한다.

```
@PostMapping
public ResponseEntity<Void> createTodo(@RequestBody @Valid TodoCreateRequest request) throws Exception {
    Long todoId = todoService.createTodo(request.getContent(), request.getUserId());
    return ResponseEntity.created(URI.create("/todo/" + todoId)).build();
}
```

# 유효성 검사 테스트

- 할 일 생성 API 호출 결과



The screenshot shows a REST client interface with tabs for Body, Cookies, Headers (4), and Test Results. The Body tab is selected, displaying a JSON response in 'Pretty' format. The response indicates a 400 Bad Request error for the path '/todo'. The JSON structure is as follows:

```
1 {  
2   "timestamp": "2024-08-29T02:40:17.232+00:00",  
3   "status": 400,  
4   "error": "Bad Request",  
5   "path": "/todo"  
6 }
```

```
default message [content],200]; default message [할 일 내용은 200자를 넘을 수 없습니다.]] ]
```

# 예외 처리

- 현재 서버에서 에러가 발생했을 때 response body 내용만으로는 에러가 발생한 원인을 알 수 없다.

```
{
  "timestamp": "2024-08-29T02:40:17.232+00:00",
  "status": 400,
  "error": "Bad Request",
  "path": "/todo"
}
```

# 예외 처리

- 에러가 발생했을 때, 원인을 알려주는 에러 메시지를 담도록 직접 응답 객체를 만들어서 전송하자.

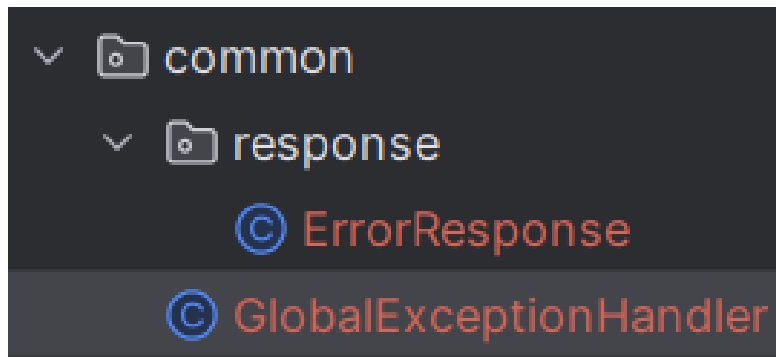
# Global Exception Handler

- 스프링은 예외 종류에 따라 응답할 response를 설정할 수 있는 **Global Exception Handler**를 제공한다.
- 이름 그대로, 스프링 어플리케이션 전역에서 발생하는 모든 에러에 대해 어떻게 처리할 지 결정한다.



# Global Exception Handler

- Global Exception Handler는 공통으로 사용하므로 **common** 패키지 밑에, **GlobalExceptionHandler** 를 생성한다.
- 에러 정보를 반환할 DTO도 response 패키지 밑에 생성한다.



# Global Exception Handler

- **ErrorResponse** 클래스는 아래와 같이 작성한다.
- Message에 더해 에러코드를 정의하여 함께 보낼 수 있다.

```
@Getter  
@AllArgsConstructor  
public class ErrorResponse {  
    private String message;  
}
```

# Global Exception Handler

- **GlobalExceptionHandler** 클래스는 다음과 같이 작성한다.

```
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {


    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleUnknownError(Exception ex) {
        ErrorResponse response = new ErrorResponse(ex.getMessage());
        return ResponseEntity.internalServerError().body(response);
    }
}
```

# Global Exception Handler

- **GlobalExceptionHandler** 클래스는 다음과 같이 작성한다.

```
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleUnknownError(Exception ex) {
        ErrorResponse response = new ErrorResponse(ex.getMessage());
        return ResponseEntity.internalServerError().body(response);
    }
}
```



Exception 타입의 에러가 발생하면  
handleUnknownError 메서드가 대신 response를 만들어서 응답한다.

# Global Exception Handler

- 스프링 어플리케이션에서 에러가 발생하면, 해당 에러 타입에 대한 핸들러가 기존 컨트롤러 대신 response body를 생성해 응답한다.
- 에러 클래스를 매칭할 때는, 상속관계를 따라 올라가며 매칭된다.
- Exception 클래스는 모든 에러 클래스의 공통 부모  
→ Exception 클래스에 대한 핸들러를 작성하면 특정 핸들러로 처리하지 못한 에러는 이 핸들러가 처리해준다.

# Global Exception Handler

- **@ControllerAdvice**는 무슨 어노테이션일까?

```
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleUnknownError(Exception ex) {
        ErrorResponse response = new ErrorResponse(ex.getMessage());
        return ResponseEntity.internalServerError().body(response);
    }
}
```

# AOP

- Aspect-Oriented Programming (관점 지향 프로그래밍)
- 객체지향 프로그래밍을 보완하는 개념

# AOP

- Aspect : 여러 클래스에서 공통적으로 갖는 관심사





# AOP

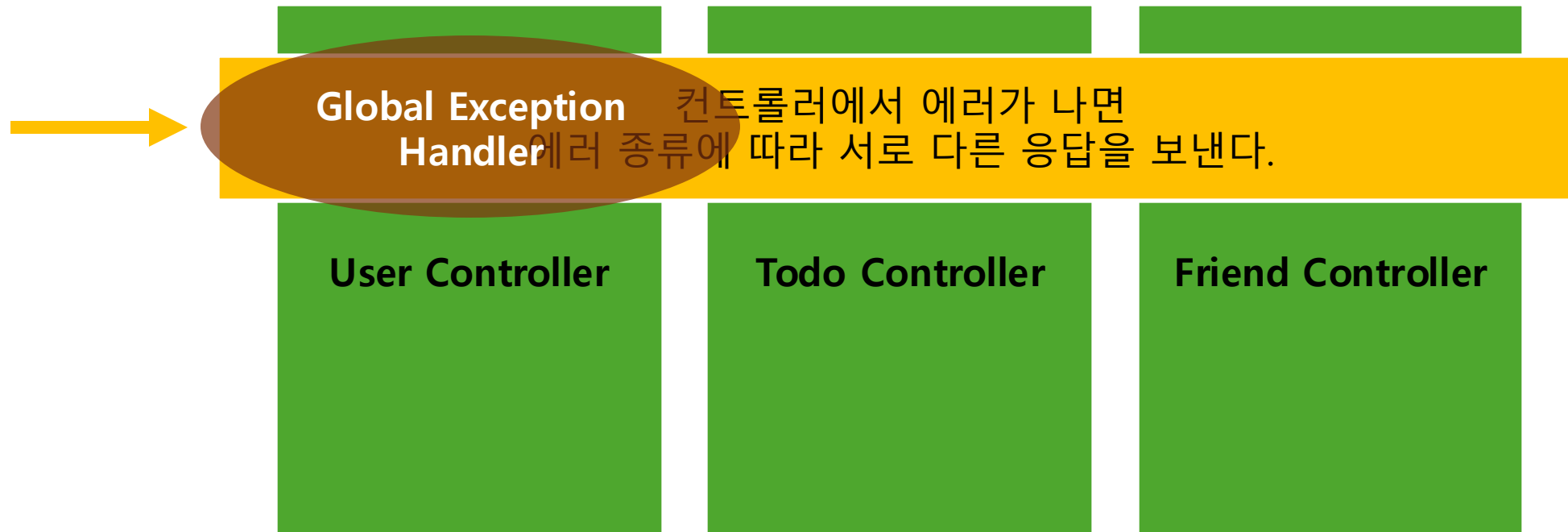
- Joint Point : 실제 프로그램의 실행 중 관심사가 발생하는 곳



User Controller 실행 중 에러가 발생하면  
에러에 맞는 응답을 보낸다.

# AOP

- Advice : 특정 joint point 에서 실행하는 action



User Controller 실행 중 에러가 발생하여  
Global Exception Handler로 대신 응답을 생성한다.

# AOP

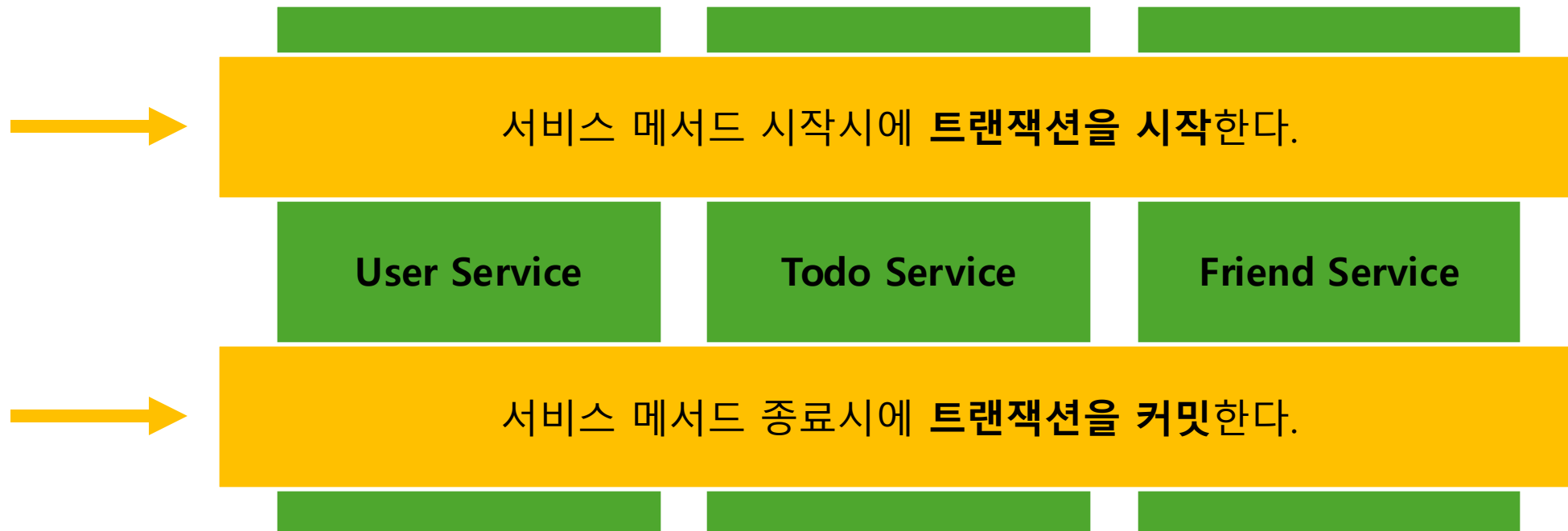
- **@ControllerAdvice**는 모든 컨트롤러의 공통 관심사(에러처리)를 별도의 클래스로 분리하여 구현한 것

```
@ControllerAdvice
public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleUnknownError(Exception ex) {
        ErrorResponse response = new ErrorResponse(ex.getMessage());
        return ResponseEntity.internalServerError().body(response);
    }
}
```

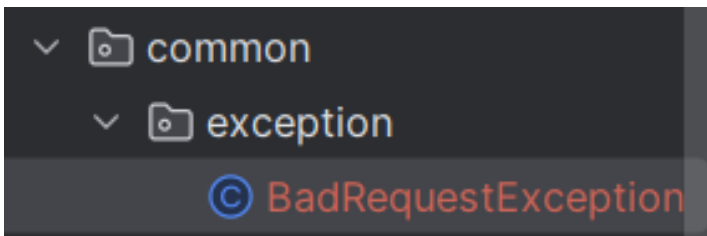
# AOP

- **@Transactional**도 모든 서비스 메서드의 공통 관심사(트랜잭션)를 미리 구현된 별도의 클래스가 대신 처리하도록 구현한 것



# 커스텀 예외 처리

- 커스텀 예외를 만들어보자.
- 어플리케이션이 공통으로 사용할 예외이므로 common 패키지 밑에 exception 패키지를 만든다.



# 커스텀 예외 처리

- 커스텀 예외 클래스를 구현한다.
- 실행 중 발생하는 예외이므로 RuntimeException 을 구현한다.

```
public class BadRequestException extends RuntimeException {  
  
    2 usages  
    public BadRequestException(String message) {  
        super(message);  
    }  
}
```

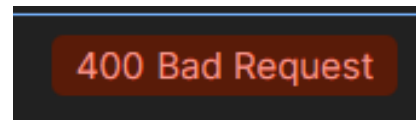
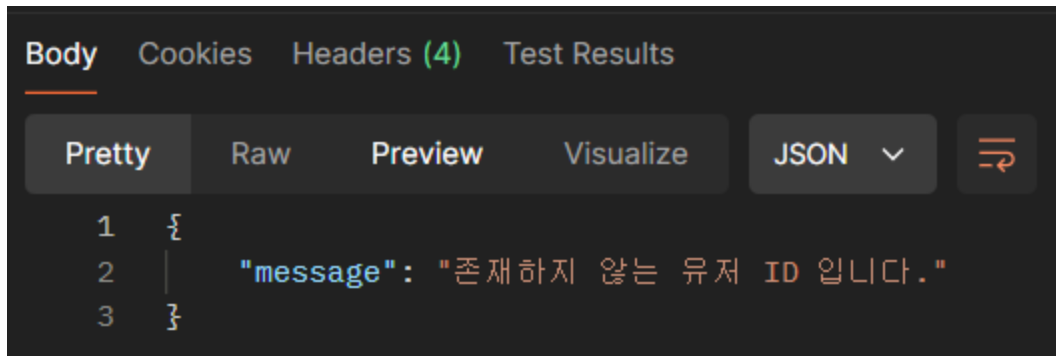
# 커스텀 예외 처리

- 커스텀 예외를 만들어보자.
- 어플리케이션이 공통으로 사용할 예외이므로 common 패키지 밑에 exception 패키지를 만든다.

```
@ExceptionHandler(BadRequestException.class)
public ResponseEntity<ErrorResponse> handleBadRequestException(Exception ex) {
    ErrorResponse response = new ErrorResponse(ex.getMessage());
    return ResponseEntity.badRequest().body(response);
}
```

# 커스텀 예외 처리

- Postman으로 존재하지 않는 user id 를 요청 보내면 400 에러가 발생하고, 에러 원인을 명확하게 응답한다.



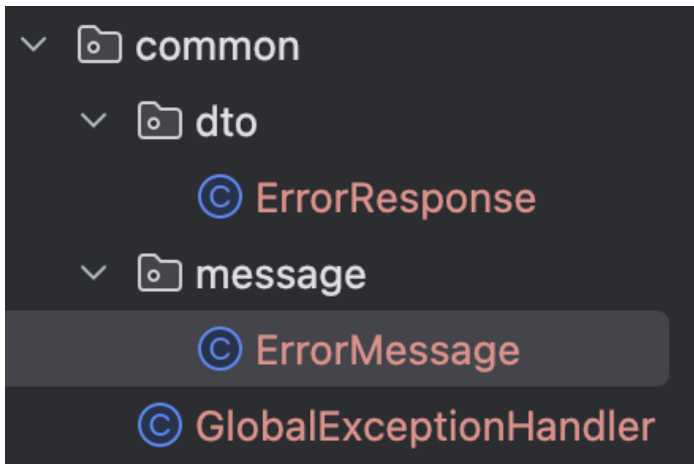


# 여러 메세지 클래스

- 예외 메세지 문자열이 여러 곳에 중복적으로 사용되는 상황
- 기존 예외 메시지를 추가하거나, 수정하기가 불편하다.

# 에러 메시지 클래스

- 예외 메시지를 상수로 정의한 클래스를 만들자.



# 에러 메시지 클래스

- 예외 메시지를 상수로 정의한 클래스를 만들자.

```
public class ErrorMessage { 0개의 사용위치  
  
    public static final String MEMBER_NOT_EXISTS = "존재하지 않는 멤버입니다."; 0개의 사용위치  
}
```

# 에러 메시지 클래스

- 서비스에 작성한 에러 메시지를 상수로 변경한다.

```
@Transactional 3개 사용 위치 ㄹ kckc0608 *  
public Long createTodo(String content, Long memberId) throws Exception {  
    Member member = memberRepository.findById(memberId);  
  
    if (member == null) {  
        throw new Exception(ErrorMessage.MEMBER_NOT_EXISTS);  
    }  
  
    Todo todo = new Todo(content, member);  
    todoRepository.save(todo);  
    return todo.getId();  
}
```

# 에러 메시지 클래스

- DTO에 작성한 에러 메시지도 상수로 적용할 수 있다.

```
@Getter 2개 사용 위치 kckc0608 *  
public class TodoCreateRequest {  
    @Length(max = 200, message = "content 길이는 200를 넘을 수 없다.")  
    private String content;  
  
    @NotNull(message = ErrorMessage.MEMBER_ID_NOT_NULL)  
    private Long memberId;  
}
```

```
public class ErrorMessage { 6개 사용 위치  
  
    public static final String MEMBER_NOT_EXISTS = "존재하지 않는 멤버입니다."; 3개 사용 위치  
    public static final String MEMBER_ID_NOT_NULL = "멤버 아이디는 필수입니다."; 1개 사용 위치  
}
```

# API 문서화

- 백엔드가 만든 API에 대한 사용법을 문서로 공유하는 것
- 프론트엔드와 협업할 때 API 문서를 공유한다.

# API 문서화 과정

1. **spring doc**을 이용해서 OpenAPI 규격으로 API 문서 생성  
(OpenAPI : 표준 API 문서 규격)
2. **Swagger-ui**를 사용하여 spring doc이 생성한 API 문서에 swagger 디자인 적용  
(swagger 이외에도 다양한 API 문서화 도구가 있습니다.)

# API 문서화

- Spring doc 의존성을 추가한다.

implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.6.0'

```
dependencies {  
    implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.6.0'  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    implementation 'org.springframework.boot:spring-boot-starter-validation'  
    compileOnly 'org.projectlombok:lombok'  
    annotationProcessor 'org.projectlombok:lombok'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    testRuntimeOnly 'org.junit.platform:junit-platform-launcher'  
}
```



# API 문서화

- spring doc은 swagger-ui를 적용한 문서를 만들어준다.  
<http://localhost:8080/swagger-ui/index.html> 접속

todo-controller	
POST	/todo
DELETE	/todo/{todoId}
PATCH	/todo/{todoId}
GET	/todo/list



# API 문서화

- API 호출 시 필요한 정보를 알 수 있고, 요청을 보내볼 수도 있다.

The screenshot shows a web interface for an API endpoint. At the top, there's a header bar with a green tab labeled 'POST' and the path '/todo'. To the right of the path are icons for a clipboard and an expand/collapse arrow. Below the header, there's a section titled 'Parameters' with a 'Try it out' button. The parameters section is currently empty, displaying 'No parameters'. Below that, there's a 'Request body' section with a red 'required' label and a dropdown menu set to 'application/json'. At the bottom, there's a tabbed interface with 'Example Value' selected, showing a JSON object: 

```
{  "content": "string",  "userId": 0}
```

 The 'Schema' tab is also visible but not selected.

**POST** /todo  

**Parameters** Try it out

No parameters

**Request body** required application/json ▼


**Example Value** | Schema

```
{  "content": "string",  "userId": 0}
```

# API 문서화

- API 호출 시 필요한 정보를 알 수 있고, 요청을 보내볼 수도 있다.

Server response

Code	Details
400 <i>Undocumented</i>	<p>Error: response status is 400</p> <p>Response body</p> <pre>{   "message": "존재하지 않는 user id 입니다." }</pre> <p> <a href="#">Download</a></p> <p>Response headers</p> <pre>connection: close content-type: application/json date: Mon, 02 Sep 2024 01:09:38 GMT transfer-encoding: chunked</pre>

# API 문서화

- @ApiResponse를 사용하여 status code마다 설명을 적을 수 있다.

```
@PostMapping  
@ApiResponse(responseCode = "404", description = "요청에 들어온 user id 가 존재하지 않음")  
public ResponseEntity<Void> createTodo(@RequestBody @Valid TodoCreateRequest request) throws Exception {  
    Long todoId = todoService.createTodo(request.getContent(), request.getUserId());  
    return ResponseEntity.created(URI.create("/todo/" + todoId)).build();  
}
```

Code	Description
404	요청에 들어온 user id 가 존재하지 않음

# 총 정리

- 1주차 : http와 백엔드의 역할
- 2주차 : 스프링 빈과 스프링 컨테이너
- 3주차 : JPA, 영속성 컨텍스트, 엔티티
- 4, 5주차 : 레포지토리 / 서비스 / 컨트롤러 레이어 구현
- 6주차 : 유효성 검사 / 예외 처리 / API 문서화

# 총 정리

## Http와 백엔드의 역할

- 웹에서 데이터를 주고받을 때는 HTTP를 따름
- 웹 서비스는 UI와 콘텐츠를 분리해서 받아옴
- 백은 프론트가 요청하는 콘텐츠의 CRUD 요청을 처리
- 요청을 처리하는 수단으로 API 제공

# 총 정리

스프링 컨테이너, 스프링 빈

- 스프링 빈 : 어플리케이션 전체에서 공유하는 하나의 객체
- 스프링 컨테이너 : 스프링 빈을 저장하는 공용 공간
  
- 의존성 : 빈의 기능을 수행할 때 특정 빈에 의존하는 것
- 의존성 주입 : 의존하는 빈을 컨테이너로부터 주입받는 것

(주요 빈 생성 & 주입 방법 : 컴포넌트 스캔, 생성자 주입, 필드 주입)

# 총 정리

- JPA

- 자바 어플리케이션이 DB와 데이터를 주고받는 표준 기술
- 영속성 컨텍스트 : 데이터 변경 사항을 보관하는 임시 공간
- 엔티티 : DB와 주고 받는 데이터 단위
- 엔티티 매니저 : DB와 직접 엔티티를 주고받는 객체



# 총 정리

- 스프링 어플리케이션 구조

- 컨트롤러 : 요청 수신, 요청 유효성 검사, 응답 전송
- 서비스 : 비즈니스 로직 처리, 응답 데이터 생성
- 레포지토리 : DB와 소통하며 CRUD 수행

각 기능을 수행하는 객체는 하나만 있으면 되기 때문에  
셋 모두 스프링 빈으로 등록하고 의존성 주입

# 총 정리

- 유효성 검사, 예외 처리
  - 유효성 검사 : 요청 데이터의 형식 (포맷, 길이 등) 검사  
DTO 클래스에서 validation 관련 어노테이션 활용
  - 예외 처리 : Global Exception Handler 클래스 활용  
AOP를 적용한 Controller Advice 어노테이션 활용

# 다음으로

- 프로젝트 시작해보기
- 스터디에서 다룬 개념을 더 깊게 파보기 (인프런 강의 추천)
- 다양한 구현 방법을 찾아보고, 이유를 고민하고 적용해보기
  - 서비스 계층이 항상 필요할까?
  - 예외 메시지 -> 문자열 상수 vs Enum
  - 다른 패키지 구조로 구성할 수 있을까?

# 다음으로

- AWS, Docker를 공부하고 직접 서버를 설정해서 배포해보기
- Java가 어떻게 빌드되고 실행되는지, **gradle, jvm** 공부해보기
- Spring Security를 공부하고 로그인 구현해보기

(개발자 유미 유튜브 강의 영상 추천 : <https://www.youtube.com/@xxxjjhhh> )

# 프로젝트 – 과제

- 과제 명세를 참고하여, 유효성 검증 추가하기
- **Global Exception Handler**를 만들고, postman으로 테스트하기
- 스웨거로 API 명세 문서화하기

# 프로젝트 명세

Todo mate API 서버 클론 코딩

## <주요 기능>

- 유저 회원가입, 로그인
- 로그인한 유저의 할 일 생성 / 조회 / 수정 / 삭제
- 로그인한 유저의 할 일 체크 / 체크 해제
- 다른 유저에게 친구 요청 / 요청 수락 / 친구 조회 / 친구 삭제
- 친구 유저의 할 일 조회

수고하셨습니다.

기말고사도 화이팅하세요~☺