스프링api 시작하기

스프링 API 8장

1. HttpSatusCode
2. UserNotFound
3. ControllerAdvice
4. Exception 처리
5. 유효성검증

박명회

1.1 httpStatus Code

참조 https://developer.mozilla.org/ko/docs/Web/HTTP/Status

HTTP 상태 코드는 클라이언트와 서버 간의 통신에서 발생한 결과를 나타내는 숫자입니다. 이 코드는 서버가 클라이언트에게 해당 요청에 대한 결과를 전달하는 데 사용됩니다. 일반적으로 3자리 숫자로 이루어져 있습니다. 첫 번째 숫자는 상태의 일반적인 범주를 나타내고, 두 번째 두 자리 숫자는 상세한 상태를 나타냅니다. 다음은 주요 HTTP 상태 코드의 간략한 설명입니다:

1xx (Informational): 요청이 받았으며 처리 중입니다. 일반적으로 사용되지 않습니다.
2xx (Success): 요청이 성공적으로 처리되었습니다.
200 (OK): 요청이 성공했으며 데이터가 반환되었습니다.
201 (Created): 요청이 성공적으로 처리되었고 새로운 리소스가 생성되었습니다.
204 (No Content): 요청이 성공했지만 반환할 콘텐츠가 없습니다.
3xx (Redirection): 추가 조치가 필요합니다. 클라이언트는 추가 작업을 수행해야 합니다.
4xx (Client Error): 클라이언트의 요청이 잘못되었습니다.
400 (Bad Request): 서버가 요청을 이해하지 못했습니다.
401 (Unauthorized): 인증이 필요합니다.
404 (Not Found): 요청한 리소스를 찾을 수 없습니다.
403 (Forbidden): 요청이 서버에서 거부되었습니다.
5xx (Server Error): 서버에서 오류가 발생하여 요청을 처리할 수 없습니다.
500 (Internal Server Error): 서버에서 요청을 처리하는 중에 오류가 발생했습니다.

HTTP 상태 코드는 클라이언트에게 요청 상태에 대한 정보를 전달하여 적절한 조치를 취할 수 있도록 합니다. 예를 들어, 성공적인 요청에 대한 응답으로는 200 상태 코드를 받으며, 오류가 발생한 경우에는 해당 오류를 나타내는 상태 코드를 받습니다.

1.2 HttpStatus Code

```java
@RestController
@RequestMapping("/users")
public class UserController {

    private final UserService userService;

    @Autowired
    public UserController(UserService userService) {
        this.userService = userService;
    }

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        List<User> users = userService.getAllUsers();
        return new ResponseEntity<>(users, HttpStatus.OK);
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        User user = userService.getUserById(id);
        if (user != null) {
            return new ResponseEntity<>(user, HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    }
```

ResponseEntity<List<User>>
모든 사용자 목록을 반환하는 메서드에서는 ResponseEntity를 사용하여 목록과 함께 HTTP 상태 코드 OK(200)을 반환합니다.
ResponseEntity<User>
특정 ID의 사용자를 반환하는 메서드에서는 사용자를 찾았을 경우에는 사용자와 함께 OK(200)을 반환하고, 찾지 못한 경우에는 NOT_FOUND(404)를 반환합니다.
ResponseEntity<Void>
사용자를 추가하거나 삭제하는 메서드에서는 HttpStatus 코드에 따라서 HttpStatus.CREATED(201) 또는 HttpStatus.NO_CONTENT(204)를 반환합니다.

```java
    @PostMapping
    public ResponseEntity<Void> addUser(@RequestBody User user) {
        userService.addUser(user);
        return new ResponseEntity<>(HttpStatus.CREATED);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> removeUser(@PathVariable Long id) {
        userService.removeUser(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
```

1.3 UserNotFoundException

사용자를 찾을 수 없는 경우에 발생하는 예외인 UserNotFoundException을 만들어 보겠습니다.

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class UserNotFoundException extends RuntimeException {
    public UserNotFoundException(String message) {
        super(message);
    }
}
```

애노테이션은 컨트롤러 메서드에서 발생하는 예외에 대한 특정 HTTP 상태 코드를 지정할 수 있는 스프링 프레임워크의 애노테이션입니다.
속성에 원하는 HTTP 상태 코드를 지정하면 됩니다.
를 사용하여 UserNotFoundException을 NOT_FOUND(404) 상태 코드로 매핑하는 예제입니다.

위의 코드는 UserNotFoundException이라는 예외 클래스를 정의합니다. 이 클래스는 RuntimeException을 확장하므로 예외가 발생할 때 반드시 처리할 필요는 없습니다.
UserNotFoundException은 사용자를 찾을 수 없을 때 throw되는 예외입니다.
이제 사용자를 찾을 수 없는 경우에 이 예외를 throw하여 클라이언트에게 해당 상황을 알릴 수 있습니다. 이것은 주로 컨트롤러나 서비스 레이어에서 사용됩니다. 사용자를 찾을 수 없는 경우에는 예외를 생성하고 throw하면 됩니다.

1.3 ExceptionResponse

사용자를 찾을 수 없는 경우에 발생하는 예외인 UserNotFoundException을 만들어 보겠습니다.

```java
import java.time.LocalDate;

public class ExceptionResponse {
    private LocalDate timestamp;
    private String message;
    private String details;

    // 생성자, getter, setter 생략

    public ExceptionResponse() {
        this.timestamp = LocalDate.now();
    }
}
```

애노테이션은 컨트롤러 메서드에서 발생하는 예외에 대한 특정 HTTP 상태 코드를 지정할 수 있는 스프링 프레임워크의 애노테이션입니다.
속성에 원하는 HTTP 상태 코드를 지정하면 됩니다.
를 사용하여 UserNotFoundException을 NOT_FOUND(404) 상태 코드로 매핑하는 예제입니다.

1.4 ControllerAdvice UerNotFoundException처리

```java
//@RestController
@ControllerAdvice
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler(UserNotFoundException.class)
    public final ResponseEntity<Object> handleUserNotFoundException(Exception ex, WebRequest request) {
        ExceptionResponse exceptionResponse =
                new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));

        return new ResponseEntity(exceptionResponse, HttpStatus.NOT_FOUND);
    }
```

1.4 ControllerAdvice

위의 예제에서 CustomResponseEntityExceptionHandler 클래스는 ResponseEntityExceptionHandler를 확장하고, @ControllerAdvice 애노테이션이 적용되어 전역적으로 예외를 처리합니다. @ExceptionHandler 애노테이션을 사용하여 특정 예외를 처리하고, 해당 예외가 발생할 경우 적절한 HTTP 상태 코드와 메시지를 반환합니다.

이제 UserNotFoundException이 발생하면 해당 예외 핸들러가 실행되어 클라이언트에게 404 상태 코드와 "User not found" 메시지를 반환합니다.

1.5 모든예외처리

```
@ExceptionHandler(Exception.class)
public final ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request) {
    ExceptionResponse exceptionResponse =
            new ExceptionResponse(new Date(), ex.getMessage(), request.getDescription(false));

    return new ResponseEntity(exceptionResponse, HttpStatus.INTERNAL_SERVER_ERROR);
}
```

위의 예제에서 handleAllExceptions 메서드는 Exception 클래스를 처리합니다. 즉, 애플리케이션에서 발생하는 모든 예외를 처리합니다. 이 핸들러는 500 상태 코드와 "An internal server error occurred" 메시지를 반환합니다.

이제 사용자 정의 예외 및 애플리케이션에서 발생하는 다른 예외 모두에 대해 일관된 방식으로 처리할 수 있습니다.

1.4 유효성 검사

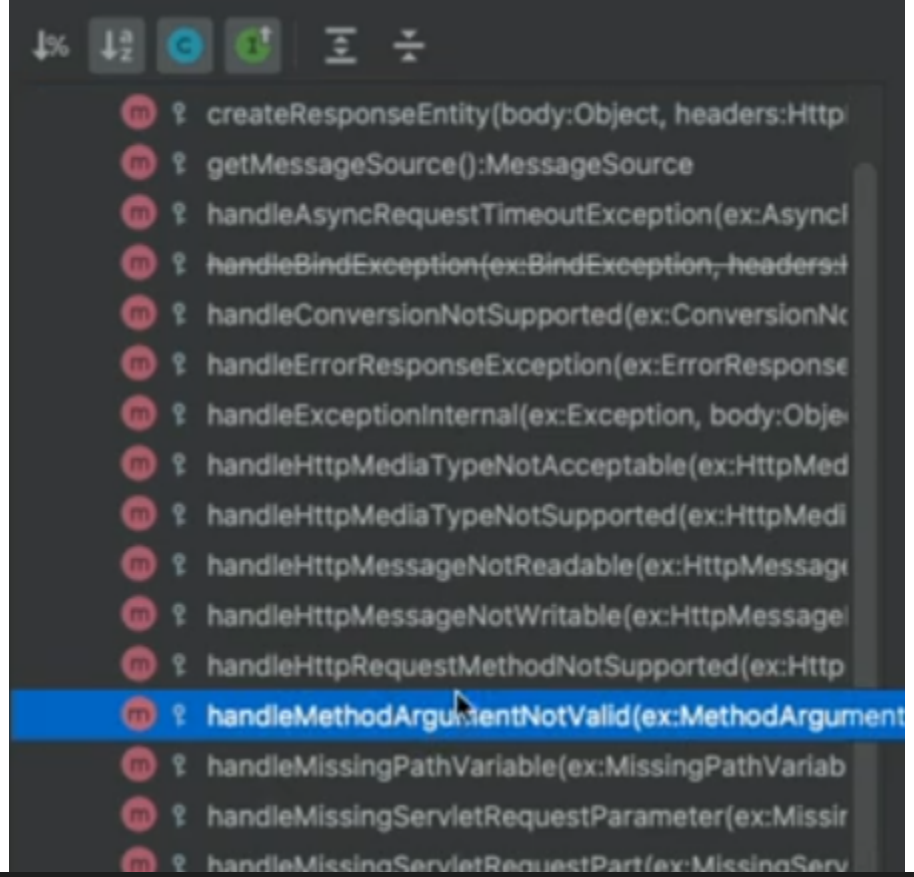
```java
import javax.validation.constraints.NotBlank;

public class User {
    private Long id;

    @NotBlank(message = "사용자 이름은 필수입니다")
    private String username;

    @NotBlank(message = "이메일은 필수입니다")
    private String email;

    // 생성자, getter, setter 생략
}
```




```java
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                                              HttpHeaders headers,
                                                              HttpStatusCode status,
                                                              WebRequest request) {
    ExceptionResponse exceptionResponse = new ExceptionResponse(new Date(),
            message: "Validation Failed", ex.getBindingResult().toString());

    return new ResponseEntity(exceptionResponse, HttpStatus.BAD_REQUEST);
}
```

1.4 유효성 검사

1.4 유효성 검사

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@Valid @RequestBody User user) {
    User savedUser = service.save(user);
```

유효성 검사에 실패한 경우에 대한 예외 처리를 @ControllerAdvice와 MethodArgumentNotValidException을 사용하여 처리할 수 있습니다. 아래의 코드는 이러한 상황을 처리하는 예제입니다.

위의 코드에서 handleValidationExceptions 메서드는 MethodArgumentNotValidException을 처리합니다. 이 예외는 유효성 검사에 실패한 경우에 발생합니다. 해당 메서드는 HTTP 상태 코드 400과 "Validation error" 메시지를 클라이언트에게 반환합니다.

이제 유효성 검사에 실패할 때 이러한 예외 핸들러가 실행되어 클라이언트에게 적절한 응답을 제공합니

고맙습니다.