Session 07

Response and exception handling

With Java 21 and Spring Boot 3.2

What’s left ?

Session #07	*03 June 2024*	Response and exception handling	
Session #08	*06 June 2024*	Beyond Frontend	
Session #09	*17 June 2024*	Final project with focus on Backend	
Session #10	*20 June 2024*	Final project with focus on Frontend	

TP

Deadline 14 June 2024

Final project

Deadline 3 July 2024
Presentation 4 July 2024

HTTP status code

HTTP defines these standard status codes that can be used to convey the results of a client's requests. They are divided into five categories

1xx	Informational
2xx	Success
3xx	Redirection
4xx	Client error
5xx	Server error

Common http status code

Request was successful

```
200 OK
```

Request was successful and a new resource was created

```
201 Created
```

Server couldn’t process the request to a client error

```
400 Bad request
```

Server encountered an unexpected exception

```
500 Internal server error
```

User don’t have the permissions to access the resource

```
403 Forbidden
```

Resource not found

```
404 Not Found
```

Server can not handle requests

```
503 Service unavailable
```

Example for OK (200) use case

```
@GetMapping  Elie Daher
@Operation(
        summary = "Get all categories",
        description = "Retrieve all categories or filter like name"
)
public ResponseEntity<List<Category>> getAll(@RequestParam(required = false) String name) {
    List<Category> categories = name == null || name.isBlank()
            ? categoryService.getAll()
            : categoryService.getAllLikeName(name);
    return ResponseEntity.ok(categories);
}
```

Code	Details
200	Response body


```
[
  {
    "id": "fe801430-9a66-4dd9-8c67-9b4793d09ec0",
    "name": "Adoption"
  },
  {
    "id": "e576cae7-a242-4b6f-b8b3-dc0548d00656",
    "name": "Children"
  },
  {
    "id": "5399f229-17aa-40d5-8239-730927f95464",
```

Example for Created (201) use case

```
@PostMapping  Elie Daher
@Operation(
        summary = "Create new category",
        description = "Create new category, only required field is the name of the category to create"
)
public ResponseEntity<Category> create(@RequestBody CategoryRequest categoryRequest) throws CategoryNameAlreadyExistsException {
    Category category = categoryService.create(categoryRequest.getName());
    return ResponseEntity
            .created(URI.create("v1/categories/" + category.getId()))
            .body(category);
}
```

201
Undocumented

Response body

```
{
  "id": "4a90980b-cc5b-46a9-ade6-c3b2cd52a861",
  "name": "My category"
}
```

Response headers

```
connection: keep-alive
content-type: application/json
date: Wed,24 Apr 2024 14:08:02 GMT
keep-alive: timeout=60
location: v1/categories/4a90980b-cc5b-46a9-ade6-c3b2cd52a861
transfer-encoding: chunked
```

Example for Not found (404) use case

```
@GetMapping("{id}")   Elie Daher *
@Operation(
        summary = "Get category by id",
        description = "Retrieve a category by id"
)
public ResponseEntity<Category> getById(@PathVariable UUID id) {
    Category category = categoryService.getById(id);
    if (category == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(category);
}
```

Code	Details

404
Undocumented
	Error: response status is 404
	Response headers
	`connection: keep-alive`
	`content-length: 0`
	`date: Wed,24 Apr 2024 14:24:12 GMT`
	`keep-alive: timeout=60`

Example for Not found (404) use case

We can enhance it, by returning an exception in the service layer instead of **`null`**

```
@Override  2 usages
public Category getById(UUID id) {
    return repository.findById(id)
            .orElse( other: null);
}
```

```
@Override  4 usages  Elie Daher
public Category getById(UUID id) throws CategoryNotFoundByIdException {
    return categoryRepository.findById(id)
            .orElseThrow(() -> new CategoryNotFoundByIdException(id));
}
```

Example for Not found (404) use case

And that way in the controller we will catch this exception and throw the adequate response

```
@GetMapping("{id}")  new *
@Operation(
        summary = "Get category by id",
        description = "Retrieve a category by id"
)
public ResponseEntity<Category> getById(@PathVariable UUID id) {
    try {
        Category category = categoryService.getById(id);
        return ResponseEntity.ok(category);
    } catch (CategoryNotFoundByIdException e) {
        return ResponseEntity.notFound().build();
    }
}
```

Example for Not found (404) use case

We can enhance by creating a handler for exception, by throwing the exception in the controller

```
@GetMapping("{id}")   Elie Daher
@Operation(
        summary = "Get category by id",
        description = "Retrieve a category by id"
)
public ResponseEntity<Category> getById(@PathVariable UUID id) throws CategoryNotFoundByIdException {
    Category category = categoryService.getById(id);
    return ResponseEntity.ok(category);
}
```

Example for Not found (404) use case

And we will have a global exception handler class, and every exception that is thrown at the controller layer, can be handled in this class.

In the case of Category not found by id it will return a 404 response

```
@ControllerAdvice   Elie Daher
public class GlobalDefaultExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalDefaultExceptionHandler.class);

    @ExceptionHandler({   Elie Daher
            CategoryNotFoundByIdException.class,
            PostNotFoundByIdException.class
    })
    public ResponseEntity<String> handleNotFoundException(Exception ex) {
        logger.warn("[NOT FOUND] {}", ex.getMessage());
        return ResponseEntity
                .status(404)
                .body(ex.getMessage());
    }
```

Make the changes to make sure all endpoints return a http status and all exceptions are handled

Sync with Github

add http status code and handle exceptions

Testing pyramid

End to End

Simulate real world functionality, slowest and expensive to run

Integration

Check software communicates, faster than E2E and slower than unit

Unit

Check small unit of code, cheapest to develop and fastest to run

Unit

Focuses on testing individual **units or components** of code in isolations

Mocks external dependencies to ensure isolation.

Verifies the correctness of small units of code, such as **functions** or **methods**, and ensures that they behave as expected.

Typically **fast** to execute since they don't involve external systems or interactions.

Integration

Tests interactions **between multiple units or components** to ensure they work together correctly.

Integration tests can be **slower and more complex than unit tests** because they require interaction with external sources.

Validates that different parts of the system **integrate seamlessly** and that **data flows correctly** between them.

End to End

Tests the **entire application** from start to finish, simulating **real user scenarios** and **interactions**.

Involves **real external systems** and dependencies, such as databases, APIs, and user interfaces.

Validates the **overall functionality and behavior of the system** as experienced by the end user.

Generally **slower** than unit and integration tests due to the comprehensive nature of testing the entire system.

Testing pyramid

End to End

Simulate real world functionality, slowest and expensive to run

Integration

Check software communicates, faster than E2E and slower than unit

Unit

Check small unit of code, cheapest to develop and fastest to run

Unit test in Spring Boot

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

With the dependency ***spring-boot-starter-test*** it will load the following libraries

- JUnit
- AssertJ
- Mockito

Best practices for writing unit tests

- **Define the purpose**: Before writing a unit test, clearly define its purpose.
- **Use a naming convention**: A good naming convention can help ensure that tests are clear, concise, and maintainable.
- **Keep tests small and focused**: Use the simplest possible input to verify the behavior being tested.
- **Use the Arrange, Act, and Assert pattern**: Break down tests into these three stages to identify what needs to be tested and ensure tests are comprehensive.
- **Avoid logic in tests**: Logic makes tests harder to read and maintain, and is more likely to have errors.
- **Create independent test cases**: Ensure that tests are independent of each other so developers can execute any test in any order.
- **Use mock objects**: Simulate dependencies, such as databases or web services, to isolate the code being tested

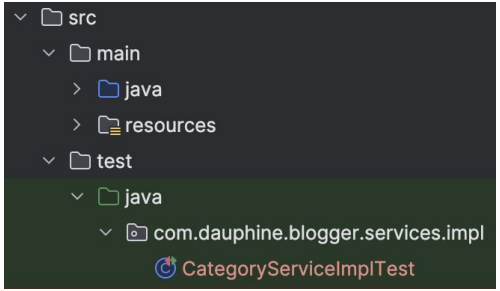
Use case : Get by id

```
@Override   4 usages   Elie Daher
public Category getById(UUID id) throws CategoryNotFoundByIdException {
    return categoryRepository.findById(id)
            .orElseThrow(() -> new CategoryNotFoundByIdException(id));
}
```

2 scenarios to test

- Should return category when id exists
- Should throws exception when id inexisting

Use case : Get by id



Let's create a new class under **`test/java`**
`CategoryServiceImplTest`
and each scenario will be a method
annotated with @Test

CategoryServiceImplTest.java

```java
package com.dauphine.blogger.services.impl;

import org.junit.jupiter.api.Test;

class CategoryServiceImplTest {  new *

    @Test  new *
    void shouldReturnCategoryWhenIdExists() {
        // TODO
    }

    @Test  new *
    void shouldThrowExceptionWhenIdDoesNotExist() {
        // TODO
    }

}
```

Scenario #1 : Should return category when id exists

```java
@Test  new *
void shouldReturnCategoryWhenIdExists() throws CategoryNotFoundByIdException {
    // Arrange
    CategoryRepository categoryRepository = mock(CategoryRepository.class);
    CategoryServiceImpl categoryService = new CategoryServiceImpl(categoryRepository);
    UUID id = UUID.randomUUID();
    Category expected = new Category( name: "Category");
    when(categoryRepository.findById(id)).thenReturn(Optional.of(expected));

    // Act
    Category actual = categoryService.getById(id);

    // Assert
    assertEquals(expected, actual);
}
```

Scenario #2 : Should throws exception when id inexisting

```java
@Test
void shouldThrowExceptionWhenNotFoundById() {
    // Arrange
    CategoryRepository categoryRepository = mock(CategoryRepository.class);
    CategoryServiceImpl categoryService = new CategoryServiceImpl(categoryRepository);
    UUID id = UUID.randomUUID();
    when(categoryRepository.findById(id)).thenReturn( t: Optional.empty());

    // Act
    CategoryNotFoundByIdException exception = assertThrows(
            CategoryNotFoundByIdException.class,
            () -> categoryService.getById(id)
    );

    // Assert
    assertEquals( expected: "Category with id " + id + " not found", exception.getMessage());
}
```

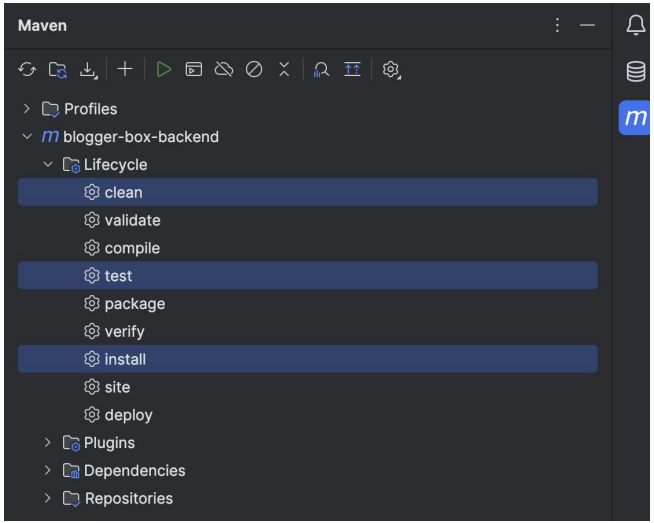
Declare service & repository at class level

Initialize the service implementation with the mocked repository every time a test is ran by annotating the **`setUp`** method with **`@BeforeEach`**

```
class CategoryServiceImplTest {  new *

    private CategoryRepository categoryRepository;  4 usages
    private CategoryServiceImpl categoryService;  3 usages

    @BeforeEach  new *
    void setUp() {
        categoryRepository = mock(CategoryRepository.class);
        categoryService = new CategoryServiceImpl(categoryRepository);
    }
```

```java
class CategoryServiceImplTest {  new *

    private CategoryRepository categoryRepository;  4 usages
    private CategoryServiceImpl categoryService;  3 usages

    @BeforeEach  new *
    void setUp() {
        categoryRepository = mock(CategoryRepository.class);
        categoryService = new CategoryServiceImpl(categoryRepository);
    }

    @Test  new *
    void shouldReturnCategoryWhenGetById() throws CategoryNotFoundByIdException {
        // Arrange
        UUID id = UUID.randomUUID();
        Category expected = new Category( name: "Category");
        when(categoryRepository.findById(id)).thenReturn(Optional.of(expected));

        // Act
        Category actual = categoryService.getById(id);

        // Assert
        assertEquals(expected, actual);
    }

    @Test  new *
    void shouldThrowExceptionWhenNotFoundById() {
        // Arrange
        UUID id = UUID.randomUUID();
        when(categoryRepository.findById(id)).thenReturn( t: Optional.empty());

        // Act
        CategoryNotFoundByIdException exception = assertThrows(
                CategoryNotFoundByIdException.class,
                () -> categoryService.getById(id)
        );

        // Assert
        assertEquals( expected: "Category with id " + id + " not found", exception.getMessage());
    }

}
```

Run test cases



Run test cases

```
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JunitPlatformProvider
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.dauphine.blogger.services.impl.CategoryServiceImplTest
WARNING: A Java agent has been loaded dynamically (/Users/elie/.m2/repository/net/bytebuddy/byte-buddy-agent/1.14.13/byte-buddy-agent-1.14.13.jar)
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDynamicAgentLoading to hide this warning
WARNING: If a serviceability tool is not in use, please run with -Djdk.instrument.traceUsage for more information
WARNING: Dynamic loading of agents will be disallowed by default in a future release
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.524 s -- in com.dauphine.blogger.services.impl.CategoryServiceImplTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Sync with Github

add unit test cases at service layer