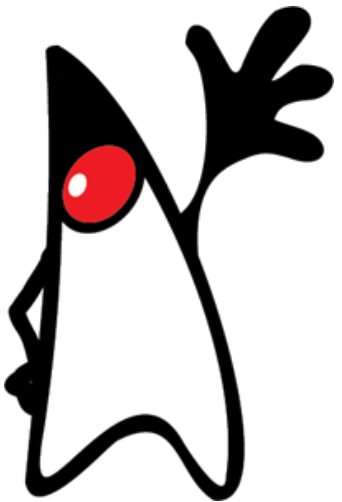




Exception handling



- **What is an exception?**
 - Categories of exceptions
- **Example project**
 - Catch wrong user input by catching the unchecked exception
 - Development of user-defined exception in REST service:
 - 1. Create Custom Error Response class
 - 2. Create Custom Exception class
 - 3. Update REST Service to throw Exception
 - 4. Add Exception Handler with @ExceptionHandler
 - Global Exception Handling with @ControllerAdvice

What is an exception?



- An **exception** is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
- An exception can occur for many different reasons. Following are some scenarios where an exception occurs:
 - a user has entered an invalid data.
 - a file that needs to be opened cannot be found.
 - a network connection has been lost in the middle of communications or the JVM has run out of memory
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
- Based on these, we have **2 categories of built-in Exceptions**:
 - Checked exceptions
 - Unchecked exceptions

Checked <-> Unchecked exceptions



- Checked Exceptions simply means that these exceptions require us to specify at compile time how the application will behave in case the exception occurs. We must anticipate in the code how to recover from the error e.g., with a try-catch-block
- The unchecked exceptions do not mandate compile time handling from us. Such exceptions occur at the time of execution. These are also called Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- Spring prefers unchecked exceptions because
 - “checked exceptions requires handling, result in cluttered code and unnecessary coupling”
 - “checked exceptions break the Open/Closed principle, since a change in the signature with a new throws declaration could have effects in many levels of our program calling the method”

Checked exceptions - example



- A **checked exception** is an exception that is checked by the compiler at compilation-time, these are also called compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions
 - For example, if you use FileReader class in your program to read data from a file:

```
import java.io.File;
import java.io.FileReader;

public class FileNotFound_Demo {

    public static void main(String args[]) {
        File file = new File("E://file.txt");
        FileReader fr = new FileReader(file);
    }
}
```

If the file specified in its constructor doesn't exist, then a FileNotFoundException occurs. The compiler prompts the programmer to handle the exception. An error is thrown at compile-time and you will not be able to run the program!

- How to handle a checked exception as a programmer?
 - Method 1: Declare the exception using throws keyword.
 - > public static void main(String args[]) throws FileNotFoundException{...}
 - Disadvantage: when the file is not found at runtime an ugly error (FileNotFoundException) will be thrown at the user
 - Method 2: Handle it using a try-catch block
 - ```
try{
 File file = new File("E://file.txt");
 FileReader fr = new FileReader(file);
}catch(FileNotFoundException fnfe){
 System.out.println("The specified file is not present at the given path");
}
```
    - Advantage: the user gets a meaningful error message

# Unchecked exceptions – example



- An **unchecked exception** is an exception that occurs at the time of execution. These are also called Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
  - For example, if you have declared an array of size 5 in your program and trying to call the 6th element of the array then an `ArrayIndexOutOfBoundsException` occurs.

```
public class Unchecked_Demo {

 public static void main(String args[]) {
 int num[] = {1, 2, 3, 4};
 System.out.println(num[5]);
 }
}
```

- The above program will compile, but when you run this program, you will get this error:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
 at Unchecked_Demo.main(Unchecked_Demo.java:5)
```

- `NullPointerException` is also an example of an unchecked exception
- We can handle unchecked exceptions with a try-catch block, just like checked exceptions but that is only a good idea when the exception happens due to wrong user input. Often unchecked exceptions are caused by programming errors. Then, of course, the idea is to fix the programming error and not to introduce a try-catch block.

# Unchecked Exception - example



demoREST / http://localhost:8080/api/products

GET http://localhost:8080/api/Products

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

| KEY | VALUE | DESCR  |
|-----|-------|--------|
| Key | Value | Descri |

Body Cookies Headers (8) Test Results

Status: 404 Not Found

Pretty Raw Preview Visualize JSON

```
1 {
2 "timestamp": "2022-03-16T15:26:13.456+00:00",
3 "status": 404,
4 "error": "Not Found",
5 "path": "/api/Products"
6 }
```

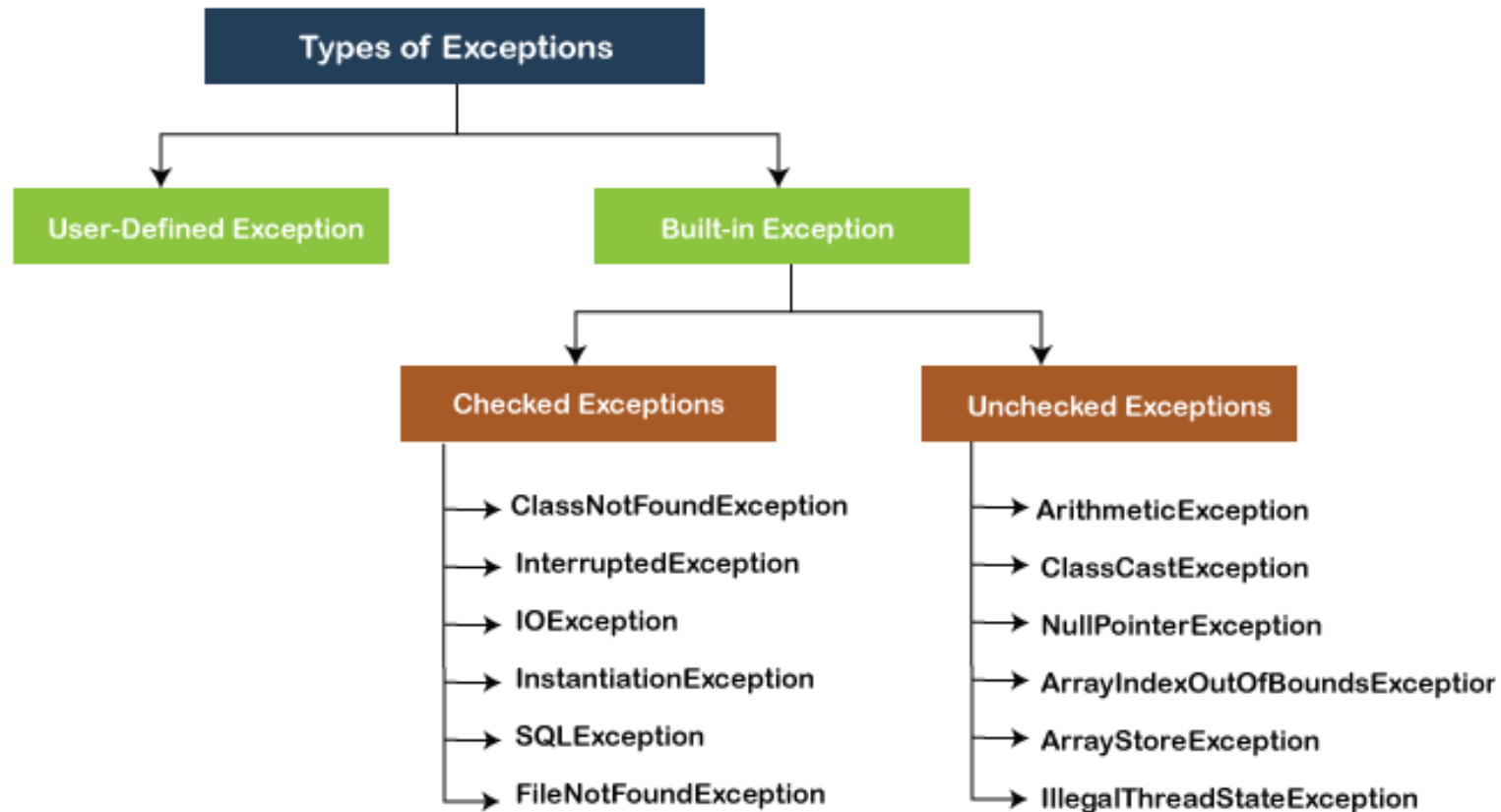
The path does not exist  
(capital letter is used...)  
=> this creates an unchecked  
exception => HTTP 404

The body of this exception

# Categories of exceptions



- in addition to built-in exceptions, we can also make our own User-Defined exceptions. That's what we are going to do in this lesson.





- What is an exception?
  - Categories of exceptions
- Example project
  - Catch wrong user input by catching the unchecked exception
  - Development of user-defined exception in REST service:
    - 1. Create Custom Error Response class
    - 2. Create Custom Exception class
    - 3. Update REST Service to throw Exception
    - 4. Add Exception Handler with @ExceptionHandler
  - Global Exception Handling with @ControllerAdvice

# Example project



- In this presentation, we will start from exampleProject-jpa from "Lesson 3 JPA, Keyword Queries & JPQL". We run the project and try to add a new bread with name "test" and price "test"

Search for a bread:

| Name   | Price | Options                                       |
|--------|-------|-----------------------------------------------|
| Bread9 | 16.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread8 | 17.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread7 | 18.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread6 | 19.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread5 | 20.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread4 | 21.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread3 | 22.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread2 | 23.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread1 | 24.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |
| Bread0 | 25.5  | <a href="#">Delete</a>   <a href="#">Edit</a> |

Totaal amount of breads: 10

[Add a new bread.](#)

**Add a new bread**

Name

Price

# Example project



- What happened?

In HTML:



A NumberFormatException is thrown at line 48 of our Breadcontroller!

```
45 @RequestMapping("/processadd")
46 public String processAdd(Model model, HttpServletRequest request) {
47 String breadName = request.getParameter("name");
48 Double breadPrice = Double.parseDouble(request.getParameter("price"));
49 Bread bread = new Bread();
50 bread.setName(breadName);
```

In IntelliJ console :

```
java.lang.NumberFormatException: For input string: "test" <2 internal calls>
 at java.base/java.lang.Double.parseDouble(Double.java:543) ~[na:na]
 at fact.it.voorbeeldprojectjpa.controller.BreadController.processAdd(BreadController.java:48) ~[classes/:na] <14 internal calls>
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:660) ~[tomcat-embed-core-9.0.31.jar:9.0.31] <1 internal call>
 at javax.servlet.http.HttpServlet.service(HttpServlet.java:741) ~[tomcat-embed-core-9.0.31.jar:9.0.31]
 at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:231) ~[tomcat-embed-core-9.0.31.jar:9.0.31]
 at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:166) ~[tomcat-embed-core-9.0.31.jar:9.0.31]
 at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:53) ~[tomcat-embed-websocket-9.0.31.jar:9.0.31]
```

# Example project: unchecked exception



- The NumberFormatException on the previous slide was caused by **incorrect user input** (a textual value was passed instead of a numerical value) and not a programming error.
- We want to give the user a nice error message that indicates exactly what is going on.
- To do that we use a try-catch block around the piece of code that generates the exception:

```
48 Double breadPrice = 0.0;
49 try {
50 breadPrice = Double.parseDouble(request.getParameter(s: "price"));
51 } catch (NumberFormatException exc) {
52 model.addAttribute(s: "message", o: "De broodprijs moet een numerieke waarde zijn");
53 return "error";
54 }
```

# Example project: try-catch



```
Double breadPrice = 0.0;
try {
 breadPrice = Double.parseDouble(request.getParameter("price"));
} catch (NumberFormatException exc) {
 model.addAttribute("message", "The price you entered was not a numeric value");
 return "error";
}
```

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
  - In this case we test if we can convert the price parameter to a double
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.
  - In this case: if we get an error converting the price to a double we want to return an error page with the message “De broodprijs moet een numerieke waarde zijn”

# Example project: error page



- To be able to show the error message to the user we make an error page: error.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
 <title>Errorpage</title>
</head>
<body>
 <h1>Something went wrong!</h1>
 <p th:text="${message}"></p>
 Home
</body>
</html>
```

# Test the example project



**Add a new bread**

Name

Price

## Something went wrong!

The price you entered was not a numeric value

[Home](#)

It works:

**Add a new bread**

Name

Price

Search for a bread:

Cheapest breads

Name	Price	Options
test	2.0	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread9	16.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread8	17.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread7	18.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread6	19.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread5	20.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread4	21.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread3	22.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread2	23.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread1	24.5	<a href="#">Delete</a>   <a href="#">Edit</a>
Bread0	25.5	<a href="#">Delete</a>   <a href="#">Edit</a>

Total amount of breads: 11

[Add a new bread.](#)

- **What is an exception?**
  - Categories of exceptions
- **Example project**
  - Catch wrong user input by catching the unchecked exception
  - Development of user-defined exception in REST service:
    - 1. Create Custom Error Response class
    - 2. Create Custom Exception class
    - 3. Update REST Service to throw Exception
    - 4. Add Exception Handler with @ExceptionHandler
  - Global Exception Handling with @ControllerAdvice



# Example project



- Now we will elaborate on the BreadRestController code.

The screenshot shows an IDE with a project named 'voorbeeldproject-jpa'. The project structure on the left includes a 'src/main/java' directory with a package 'fact.it.voorbeeldprojectjpa'. Inside this package, there is a 'controller' directory containing 'BreadController' and 'BreadRestController'. The 'BreadRestController' file is selected and its code is displayed in the editor. The code defines a REST controller for bread records, using Spring annotations like @RestController, @RequestMapping, and @Autowired.

```
1 package fact.it.voorbeeldprojectjpa.controller;
2
3 import ...
4
5
6
7
8
9
10
11
12 @RestController
13 @RequestMapping("/api")
14 public class BreadRestController {
15 @Autowired
16 private BreadRepository breadRepository;
17
18
19 // een service om alle bread-records uit de database op te halen
```

# Exception handling



- We will create a new API endpoint to retrieve a single bread by id. For that we add the following code to BreadRestController:

```
//een service om een brood op te vragen op basis van zijn id
@GetMapping("/breads/{id}")
public Bread getBread(@PathVariable long id){
 return breadRepository.findBreadById(id);
}
```

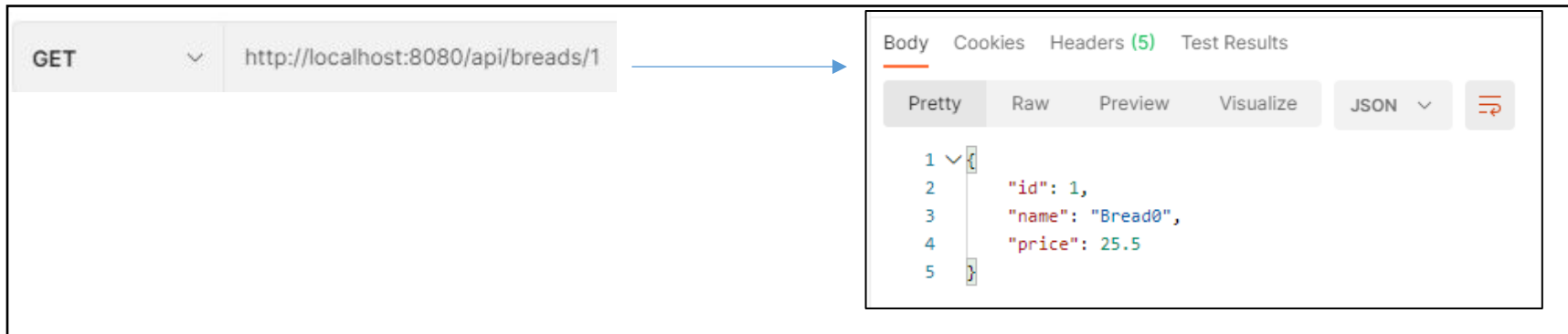
- In breadRepository we add this method:

```
Bread findBreadById(long id);
```

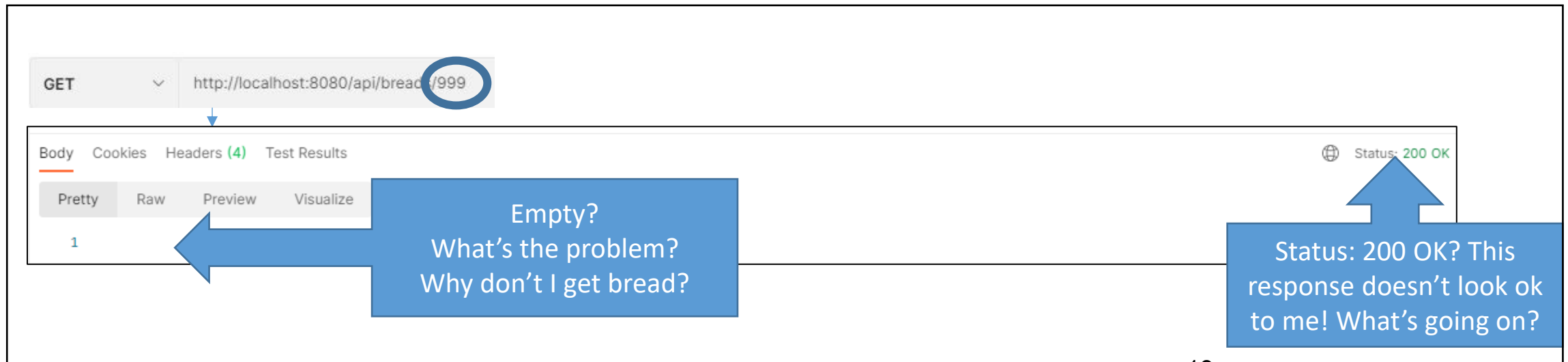
# Exception handling



- We test our new API endpoint with Postman



- What happens if the user inputs a non-existent id?



# Exception handling



- What happens if we shut down the MySQL database server?

The screenshot shows a web browser with the address bar set to `http://localhost:8080/api/breads/1`. The response is displayed in the 'Body' tab, showing a JSON error message. A blue arrow points from the text 'Error message formatted as JSON' to the JSON object. Another blue arrow points from the text 'Status: 500 Something went seriously wrong...' to the 'Status: 500 Int' indicator in the top right corner.

```
{
 "timestamp": "2021-05-04T12:08:04.777+0000",
 "status": 500,
 "error": "Internal Server Error",
 "message": "Unable to acquire JDBC Connection; nested exception is org.hibernate.exception.JDBCConnectionException: Unable to acquire JDBC Connection",
 "path": "/api/breads/1"
}
```

Status: 500 Int

Status: 500  
Something went seriously wrong...

Error message formatted as JSON

- When a built-in unchecked exception occurs, we get a JSON formatted error message + the status code is changed.

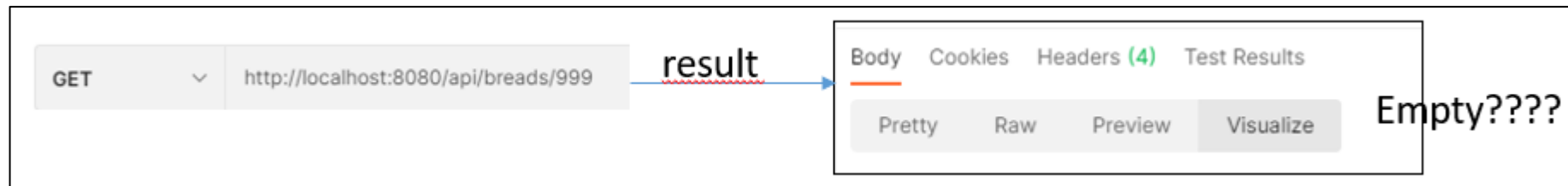
# Exception handling



- Suppose we want to have a service to retrieve a bread with a certain id:

```
//a service to retrieve a bread with a certain id
@GetMapping("/bread/{id}")
public ResponseEntity<Bread> getBread(@PathVariable("id") long id) {
 Optional<Bread> bread = breadRepository.findBreadById(id);
 if (bread.isPresent()){
 return new ResponseEntity<>(bread.get(), HttpStatus.OK);
 }
 return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
```


- When no bread with the given id is found, we get an empty return:






# Exception handling



- When no bread with the given id is found, we want to throw our own exception instead of the null return
- Desired output: error formatted as JSON, just like the connection error but with a customised message:

```
Pretty Raw Preview Visualize JSON v 
```

```
1 
2 "timestamp": "2022-04-21T09:49:25.351+0000",
3 "status": 500,
4 "error": "Internal Server Error",
5 "message": "No bread found with id = 999",
6 "path": "/api/breads/999"
7 
```



# Exception handling



- Instead of returning an Optional the service has to return a Bread-object:

```
@GetMapping("/{id}")
public Bread getBread(@PathVariable("id") long id) {
 return breadRepository.findBreadById(id)
 .orElseThrow(() -> new RuntimeException("No bread found with id = " + id));
}
```

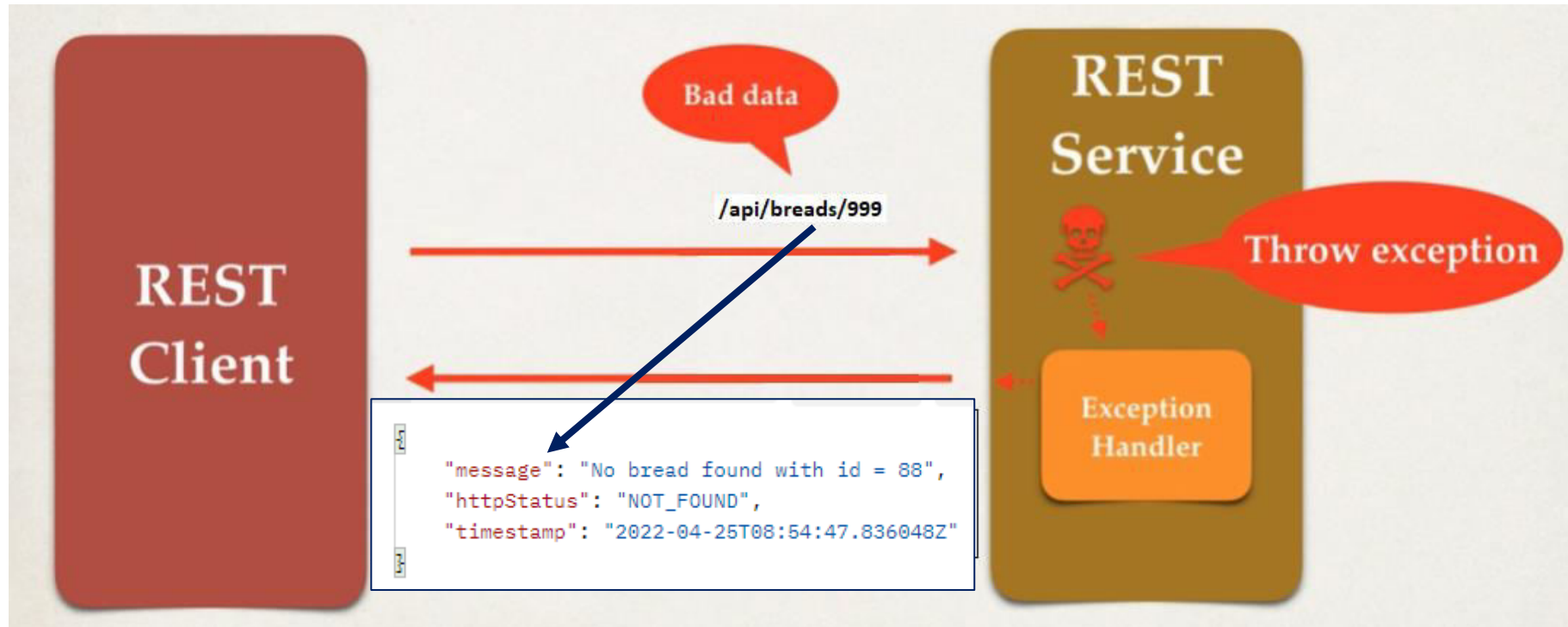
- When no bread with the given id is found, we now get this output:

```
1 {
2 "timestamp": "2022-04-21T09:49:25.351+0000",
3 "status": 500,
4 "error": "Internal Server Error",
5 "message": "No bread found with id = 999",
6 "path": "/api/breads/999"
7 }
```

# Exception handling



- It is possible to further customize the exception thrown and avoid the standard runtime exception.





- To fully customize the standard exception, several steps are required:
  1. Create a Custom Exception class
    - inherits from the RuntimeException-class (*ApiRequestException-class*)
  2. Create a Custom ExceptionHandler class (*ApiExceptionHandler-class*)
    - contains a method that returns a ResponseEntity-object that will return a customized message which is an object of the Custom Response class (see step 3) when an error has occurred.
  3. Create a Custom Response class (*ApiExceptionHandlerErrorResponse-class*)
    - contains all details of the customized error you want to see in the output-window if an error occurs
  4. Change the controller in order to throw the customized exception

# Exception handling



- In the [video](#) below you see in detail how to make your own Exception Handler in Java Spring Boot:



- For your information: for the exams, we will not ask you to implement your own Exception Handler in practice. But you need to know this concept in theory.