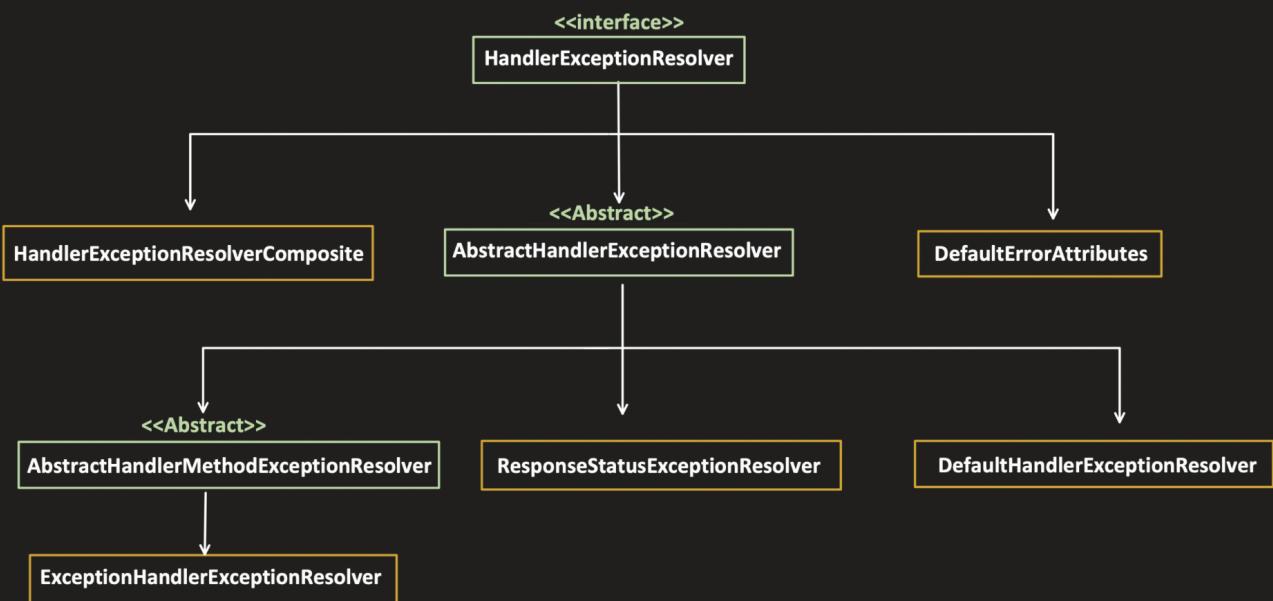


18. Spring boot Exception Handling

Classes involved in handling an Exception:

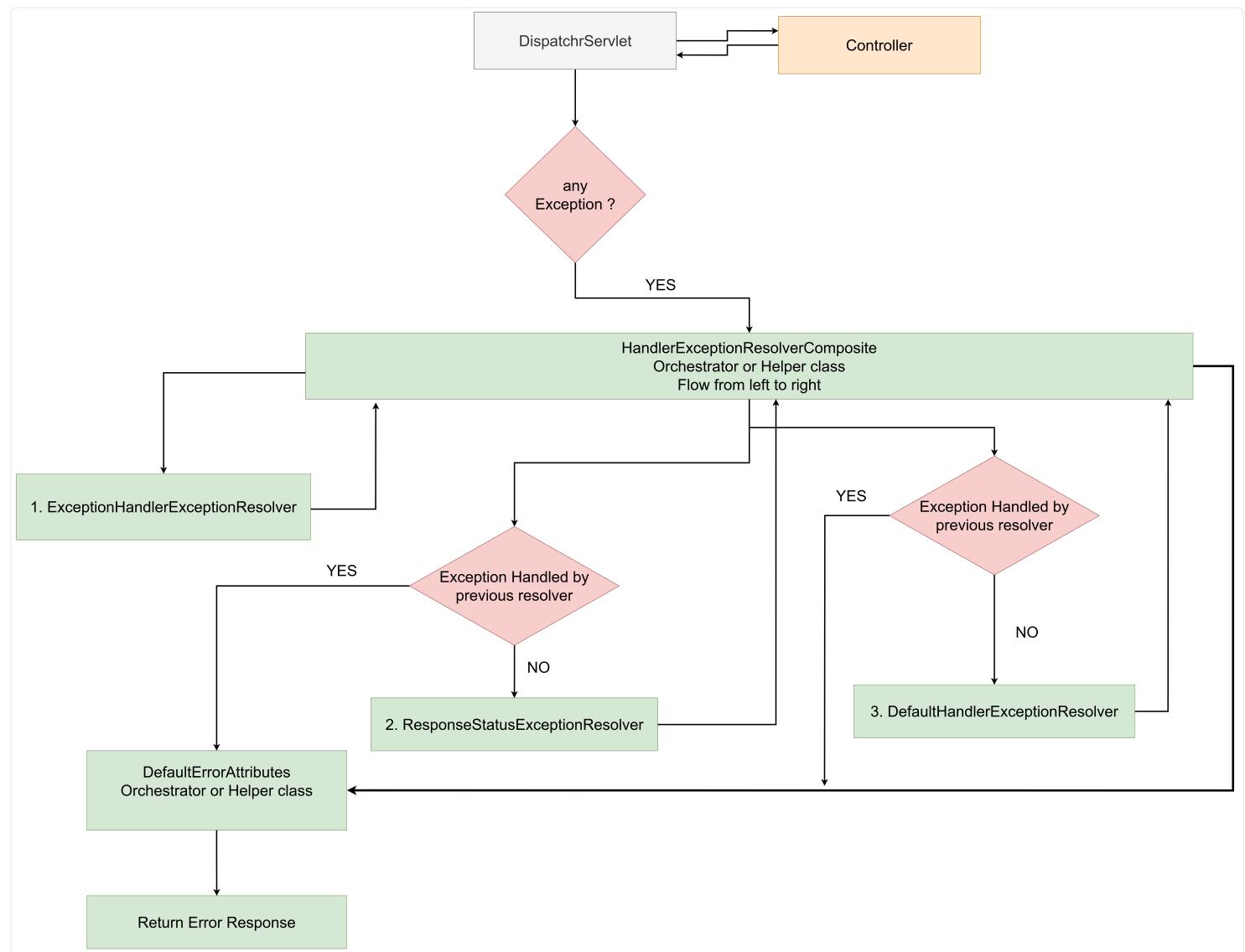


1. `ExceptionHandlerExceptionResolver`.
2. `ResponseStatusExceptionResolver`.
3. `DefaultHandlerExceptionResolver`.
4. `HandlerExceptionResolverComposite`.
5. `DefaultErrorAttributes`.

These are the important class we have to learn.

At the top we have HandlerExceptionResolver who expose methods like doResolve.

Let's understand the sequence when an exception occurs.



let's understand with example.

Let's see the flow with an example:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {
        throw new NullPointerException("throwing null pointer exception for testing");
    }
}
```

Output:

The screenshot shows a REST client interface with the following details:

- Method: GET
- URL: localhost:8080/api/get-user
- Params tab is selected.
- Query Params table:

Key	Value	Description
Expect	100	
Key	Value	Description
- Body tab is selected.
- JSON output:

```
1  {
2   "timestamp": "2024-10-22T16:36:34.796+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "path": "/api/get-user"
6 }
```
- Header status bar shows: 500 Internal Server Error

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public String getUser() {

        throw new CustomException(HttpStatus.BAD_REQUEST,
                "request is not correct, UserID is missing");
    }
}

```

```

public class CustomException extends RuntimeException{

    HttpStatus status;
    String message;

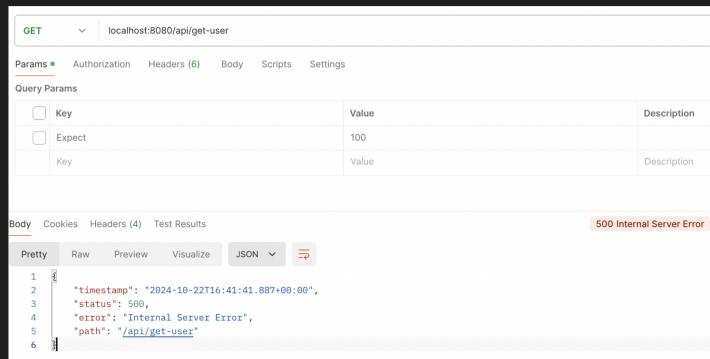
    CustomException(HttpStatus status, String message) {
        this.status = status;
        this.message = message;
    }

    public HttpStatus getStatus() {
        return status;
    }

    @Override
    public String getMessage() {
        return message;
    }
}

```

again output is same:



Why for both output is same ?

Why my return code **BAD_REQUEST** ie 400 and my error message is not shown in output.

- In both the example, we are not creating the **ResponseType** Object.
- If we need full control and don't want to rely on Exception Resolvers, then we have to create the **ResponseType** Object.

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        try {
            //your business logic and validations...
            throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
        } catch (CustomException e) {
            ErrorResponse errorResponse = new ErrorResponse(new Date(), e.getMessage(), e.getStatus().value());
            return new ResponseEntity<?>(errorResponse, e.getStatus());
        } catch (Exception e) {
            ErrorResponse errorResponse = new ErrorResponse(new Date(), e.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR.value());
            return new ResponseEntity<?>(errorResponse, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}

```

```

public class CustomException extends RuntimeException {

    HttpStatus status;
    String message;

    CustomException(HttpStatus status, String message) {
        this.status = status;
        this.message = message;
    }

    public HttpStatus getStatus() {
        return status;
    }

    @Override
    public String getMessage() {
        return message;
    }
}

public class ErrorResponse {

    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse(Date timestamp, String msg, int status) {
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }
}

```



If we want to handle the exception manually we need to return `ResponseEntity` object.
But if you don't want to handle it you new

So, When we don't return `ResponseEntity` like below:

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

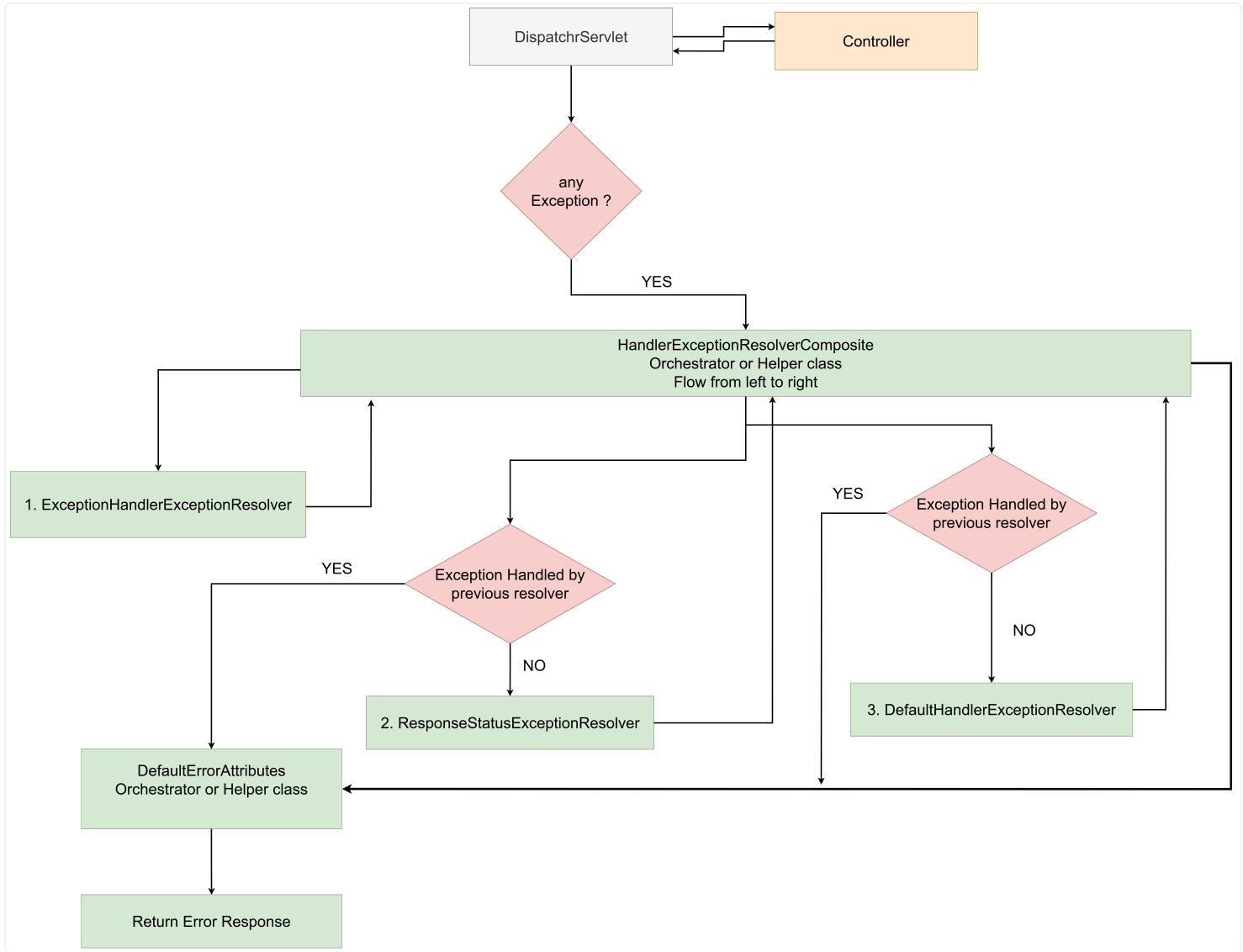
    @GetMapping(path = "/get-user")
    public String getUser() {

        throw new CustomException(HttpStatus.BAD_REQUEST,
                               "request is not correct, UserID is missing");
    }
}

```

then in Exception scenario, exception passes through each Resolver like "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" in sequence.

then this flow comes into picture



These exception are predefined exception which handled exception like `mediatype`, `pathvariablenotfound`,

these resolver support Spring Boot to handle predefined exception.

- Each resolver set the proper status and message in HTTP response for exceptions which they are taking care of .
- **NullpointerException** and **CustomException** all the 3 **Resolver**, do not understand, so **status** and **message** is not set
- So, when control reaches to **DefaultErrorAttributes** class, it fills the values in **HTTP Response** with default values.

Exception	HTTP Status Code
HttpRequestMethodNotSupportedException	405 (SC_METHOD_NOT_ALLOWED)
HttpMediaTypeNotSupportedException	415 (SC_UNSUPPORTED_MEDIA_TYPE)
HttpMediaTypeNotAcceptableException	406 (SC_NOT_ACCEPTABLE)
MissingPathVariableException	500 (SC_INTERNAL_SERVER_ERROR)
MissingServletRequestParameterException	400 (SC_BAD_REQUEST)
MissingServletRequestPartException	400 (SC_BAD_REQUEST)
ServletRequestBindingException	400 (SC_BAD_REQUEST)
ConversionNotSupportedException	500 (SC_INTERNAL_SERVER_ERROR)
TypeMismatchException	400 (SC_BAD_REQUEST)
HttpMessageNotReadableException	400 (SC_BAD_REQUEST)
HttpMessageNotWritableException	500 (SC_INTERNAL_SERVER_ERROR)
MethodArgumentNotValidException	400 (SC_BAD_REQUEST)
<u>MethodValidationException</u>	500 (SC_INTERNAL_SERVER_ERROR)
<u>HandlerMethodValidationException</u>	400 (SC_BAD_REQUEST)
NoHandlerFoundException	404 (SC_NOT_FOUND)
NoResourceNotFoundException	404 (SC_NOT_FOUND)
AsyncRequestTimeoutException	503 (SC_SERVICE_UNAVAILABLE)
AsyncRequestNotUsableException	Not applicable

In Below image what's happen, let's understand.

- After throw of exception first control goes to **ExceptionHandlerExceptionResolver**, not handled, secondly it's goes to **ResponseStatusExceptionResolver**, not handled,
- third it's goes to **DefaultExceptionHandlerResolver**.

In whole process not a single resolver able to resolve the exception.

- So, Finally Default Attributes comes into the picture.

- Status→500

- error→"Internal Server Error"

DefaultErrorAttributes

```
@Override
public Map<String, Object> getErrorAttributes(WebRequest webRequest, ErrorAttributeOptions options) {
    Map<String, Object> errorAttributes = getErrorAttributes(webRequest, options.isIncluded(Include.STACK_TRACE));
    options.retainIncluded(errorAttributes);
    return errorAttributes;
}

private Map<String, Object> getErrorAttributes(WebRequest webRequest, boolean includeStackTrace) {
    Map<String, Object> errorAttributes = new LinkedHashMap<>();
    errorAttributes.put("timestamp", new Date());
    addStatus(errorAttributes, webRequest);
    addErrorDetails(errorAttributes, webRequest, includeStackTrace);
    addPath(errorAttributes, webRequest);
    return errorAttributes;
}
```

```
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    HttpStatus status = this.getStatus(request);
    if (status == HttpStatus.NO_CONTENT) {
        return new ResponseEntity(status);
    } else {
        Map<String, Object> body = this.getErrorAttributes(request, this.getErrorAttributeOptions(request, MediaType.ALL));
        return new ResponseEntity(body, status);
    }
}
```

The screenshot shows a POSTMAN API client interface. The top bar indicates a 'GET' method and the URL 'localhost:8080/api/get-user'. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Scripts', and 'Settings'. Under 'Params', there is a table for 'Query Params' with three entries: 'Key' (checkbox), 'Expect' (checkbox), and 'Key' (checkbox). The 'Pretty' tab is selected in the bottom navigation bar, and the JSON response is displayed:

```
1 "timestamp": "2024-10-22T16:41:41.887+00:00",
2 "status": 500,
3 "error": "Internal Server Error",
4 "path": "/api/get-user"
```

A red box highlights the '500 Internal Server Error' status code in the top right corner of the response area.

So, now question is: what exception does

"ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" handles?

1. ExceptionHandlerExceptionResolver

Responsible for handling below annotations:

- @ExceptionHandler
- @ControllerAdvice

Controller level Exception handling:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
    }
}
```

The screenshot shows a REST API testing interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (4)', and 'Test Results'. On the right, it says '400 Bad Request'. Below these are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'Text' (with a dropdown arrow). Under the 'Text' button is a red 'Copy' icon. The main area displays the response body: '1 UserID is missing'.

Use-case just to show returning Error Response object instead of just message:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<Object> handleCustomException(CustomException ex) {
        ErrorResponse errorResponse = new ErrorResponse(new Date(), ex.getMessage(), ex.getStatus().value());
        return new ResponseEntity<?>(errorResponse, ex.getStatus());
    }
}
```

```
public class ErrorResponse {

    private Date timestamp;
    private String msg;
    private int status;

    public ErrorResponse(Date timestamp, String msg, int status) {
        this.msg = msg;
        this.status = status;
        this.timestamp = timestamp;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public String getMessage() {
        return msg;
    }

    public int getStatus() {
        return status;
    }
}
```

The screenshot shows a REST API testing interface. At the top, there are tabs for Body, Cookies, Headers (4), and Test Results. The Body tab is selected and displays a JSON response. The response is shown in Pretty format, with line numbers 1 through 5. The JSON content is as follows:

```
1  {
2      "timestamp": "2024-10-24T15:41:24.294+00:00",
3      "status": 400,
4      "message": "UserID is missing"
5  }
```

In the top right corner of the interface, there is a red box containing the text "400 Bad Request".

Use-case just to show multiple @ExceptionHandler in single Controller class:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<?> getUserHistory() {
        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
    }

    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleCustomException(IllegalArgumentException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

The image displays two side-by-side screenshots of a REST client interface, likely Postman, used to test a Spring REST API.

Left Screenshot (localhost:8080/api/get-user-history):

- Method: GET
- URL: localhost:8080/api/get-user-history
- Headers: Authorization, Headers (6), Body, Scripts, Settings
- Query Params:
 - Key: 100
 - Value: 100
- Body: Cookies, Headers (4), Test Results
 - Pretty
 - Raw
 - Preview
 - Visualize
 - Text (selected)
 - Copy
- Test Results: 400 Bad Request
 - 1 inappropriate arguments passed

Right Screenshot (localhost:8080/api/get-user):

- Method: GET
- URL: localhost:8080/api/get-user
- Headers: Authorization, Headers (6), Body, Scripts, Settings
- Query Params:
 - Key: Value
 - Value: 100
- Body: Cookies, Headers (4), Test Results
 - Pretty
 - Raw
 - Preview
 - Visualize
 - Text (selected)
 - Copy
- Test Results: 400 Bad Request
 - 1 UserID is missing

Use-case just to show 1 @ExceptionHandler handling multiple exceptions:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<?> getUserHistory() {

        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler({CustomException.class, IllegalArgumentException.class})
    public ResponseEntity<String> handleCustomException(Exception ex) {
        return new ResponseEntity<String>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

Postman screenshot showing a successful GET request to `localhost:8080/api/get-user`. The response status is `200 OK` and the body contains the JSON object `{ "id": 1, "name": "John Doe", "age": 30 }`.

Postman screenshot showing a failed GET request to `localhost:8080/api/get-user-history`. The response status is `400 Bad Request` and the body contains the JSON object `{ "error": "inappropriate arguments passed" }`.

Use-case just to show `@ExceptionHandler` not returning `ResponseEntity` and let "**DefaultErrorAttributes**" to create the `ResponseEntity`.

```
@RestController
@RequestMapping(value = "/api/*")
public class UserController {
    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @GetMapping(path = "/get-user-history")
    public ResponseEntity<UserHistory> getUserHistory() {
        //your business logic and validations...
        throw new IllegalArgumentException("inappropriate arguments passed");
    }

    @ExceptionHandler(CustomException.class)
    public void handleCustomException(HttpServletRequest response, CustomException ex) throws IOException {
        response.sendError(HttpStatus.BAD_REQUEST.value(), ex.message);
    }
}
```

Without this `DefaultErrorAttributes`,
filter out the message field in response

application.properties

1 | server.error.include-message=always

```
Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize JSON ↻
1 400 Bad Request
2
3
4
5
6
7
```

```
1
2   "timestamp": "2024-10-24T15:55:07.007+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "UserID is missing",
6   "path": "/api/get-user"
```

Global Exception handling:

Problem with Controller level @ExceptionHandler is:

- if multiple controller has the same type of Exceptions then same handling we might do in multiple controller
- which is nothing but a code duplication.

```
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.message, ex.getStatus());
    }
}
```



What if, I provide both Controller level and Global level @ExceptionHandler, which one has more priority?

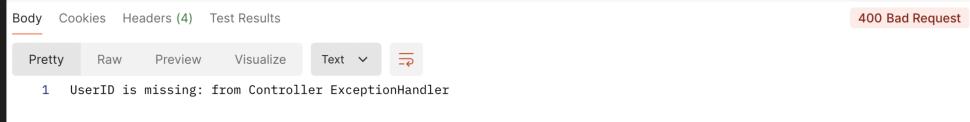
```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(body: ex.message + ": from Controller ExceptionHandler", ex.getStatus());
    }
}
```

```
@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(body: ex.message + ": from Global ExceptionHandler", ex.getStatus());
    }
}
```



What if there are 2 handlers which can handle an exception, which one will be given priority:

It always follow an hierarchy, from bottom to up (first look for exact match if not, check for its parent and so on...)

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.BAD_REQUEST, "UserID is missing");
    }
}

@ControllerAdvice
public class GlobalExceptionHandling {

    @ExceptionHandler(CustomException.class)
    public ResponseEntity<String> handleCustomException(CustomException ex) {
        return new ResponseEntity<>(ex.message , ex.getStatus());
    }

    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

2. ResponseStatusExceptionResolver

Handles Uncaught exception annotated with **@ResponseStatus** annotation.

Use-case1: Used above an Exception class

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException("UserID is missing");
    }
}
```

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
public class CustomException extends RuntimeException {

    CustomException(String message) {
        super(message);
    }
}
```



```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {
        //your business logic and validations...
        throw new CustomException("UserID is missing");
    }
}
```

```
@ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Passed")
public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}
```

Body Cookies Headers (4) Test Results 400 Bad Request

Pretty Raw Preview Visualize JSON

```
1  {
2      "timestamp": "2024-10-25T13:03:50.574+00:00",
3      "status": 400,
4      "error": "Bad Request",
5      "message": "Invalid Request Passed",
6      "path": "/api/get-user"
7 }
```

Use-case2: Used above an @ExceptionHandler method

Again **ResponseStatusExceptionResolver** handles Uncaught exception annotated with **@ResponseStatus** annotation but if used with **@ExceptionHandler** then it will not be handled by "**ResponseStatusExceptionResolver**", it will be handled by Spring request handling mechanism itself.

```

@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<User> getUser() {
        //your business logic and validations...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public ResponseEntity<Object> handleCustomException(CustomException e) {
        return new ResponseEntity<Object>((body: "you are not authorized", HttpStatus.FORBIDDEN);
    }
}

```

```

public class CustomException extends RuntimeException {

    HttpStatus status;

    CustomException(HttpStatus status, String message) {
        super(message);
        this.status = status;
    }
}

```

The screenshot shows a browser's developer tools Network tab. A request to '/api/get-user' resulted in a 400 Bad Request response. The JSON body of the response is:

```

1  {
2     "timestamp": "2024-10-25T14:05:53.089+00:00",
3     "status": 400,
4     "error": "Bad Request",
5     "message": "Invalid Request Sent",
6     "path": "/api/get-user"
7   }

```

ExceptionHandlerExceptionResolver.java

```

protected ModelAndView doResolveHandlerMethodException(HttpServletRequest request,
                                                      HttpServletResponse response, @Nullable HandlerMethod handlerMethod, Exception exception) {
    ServletInvocableHandlerMethod exceptionHandlerMethod = getExceptionHandlerMethod(handlerMethod, exception);
    if (exceptionHandlerMethod == null) {
        return null;
    }

    if (this.argumentResolvers != null) {
        exceptionHandlerMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
    }

    if (this.returnValueHandlers != null) {
        exceptionHandlerMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
    }

    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    ModelAndView mavContainer = new ModelAndViewContainer();

    ArrayList<Throwable> exceptions = new ArrayList<>();
    try {
        if (logger.isDebugEnabled()) {
            logger.debug("Using " + exceptionHandlerMethod);
        }
        // Expose cause as provided arguments as well
        Throwable exToExpose = exception;
        while (exToExpose != null) {
            exceptions.add(exToExpose);
            Throwable cause = exToExpose.getCause();
            exToExpose = (cause != exToExpose ? cause : null);
        }
        Object[] arguments = new Object[exceptions.size() + 1];
        exceptions.toArray(arguments); // efficient arraycopy call in ArrayList
        arguments[arguments.length - 1] = handlerMethod;
        exceptionHandlerMethod.invokeAndHandle(webRequest, mavContainer, arguments);
    }
}

```

ServletInvocableHandlerMethod.java

```

public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
                           Object... providedArgs) throws Exception {
    Object returnValue = invokeForWebRequest(webRequest, mavContainer, providedArgs);
    setResponseStatus(webRequest);

    if (returnValue == null) {
        if (!isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {
            disableContentCachingIfNecessary(webRequest);
            mavContainer.setRequestHandled(true);
            return;
        }
    } else if (StringUtils.hasText(getResponseStatusReason())) {
        mavContainer.setRequestHandled(true);
        return;
    }

    mavContainer.setRequestHandled(false);
    Assert.state(expression: this.returnValueHandlers != null, message: "No return value handlers");
    try {
        this.returnValueHandlers.handleReturnValue(
            returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
    } catch (Exception ex) {
        if (logger.isTraceEnabled()) {
            logger.trace(formatErrorForReturnValue(returnValue), ex);
        }
        throw ex;
    }
}

```

What if @ExceptionHandler method, set Response status and message itself instead of returning the response entity:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

        //your business logic and validations...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e, HttpServletResponse response) throws IOException{
        response.sendError(HttpStatus.FORBIDDEN.value(), s: "you are not authorized");
    }
}
```

Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize JSON ↻

500 Internal Server Error

```
1 {  
2   "timestamp": "2024-10-25T14:08:14.396+00:00",  
3   "status": 500,  
4   "error": "Internal Server Error",  
5   "message": "YOUR ARE NOT AUTHORIZED",  
6   "path": "/api/get-user"  
7 }
```

Its because, Response.sendError first set the status and message in response and do COMMIT.

2nd ResponseStatus method will try to do the same thing, and Exception will occur in ExceptionHandlerResolver class as we try to reset already committed status field.

So its advisable not to use together @ExceptionHandler and @ResponseStatus together to avoid confusion.

But if you have to, use like below:

```
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @GetMapping(path = "/get-user")
    public ResponseEntity<?> getUser() {

        //your business logic and validations...
        throw new CustomException(HttpStatus.INTERNAL_SERVER_ERROR, "UserID is missing");
    }

    @ExceptionHandler(CustomException.class)
    @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason = "Invalid Request Sent")
    public void handleCustomException(CustomException e) {
        //do nothing here
    }
}
```

Body Cookies Headers (4) Test Results
Pretty Raw Preview Visualize JSON ↻

400 Bad Request

```
1 {  
2   "timestamp": "2024-10-25T14:14:52.389+00:00",  
3   "status": 400,  
4   "error": "Bad Request",  
5   "message": "Invalid Request Sent",  
6   "path": "/api/get-user"  
7 }
```

3. DefaultHandlerExceptionResolver

Handles Spring framework related exceptions only like MethodNotFound, NoResourceFound etc..