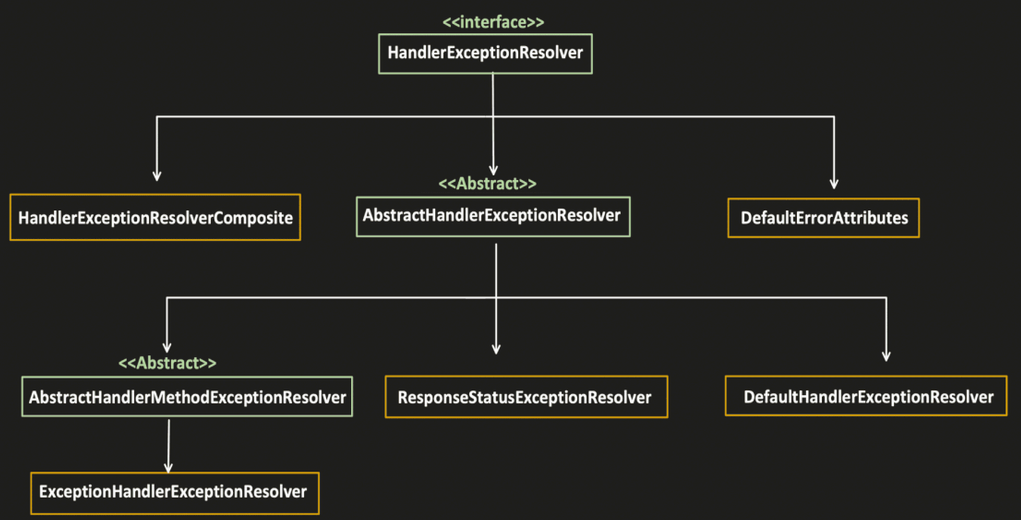
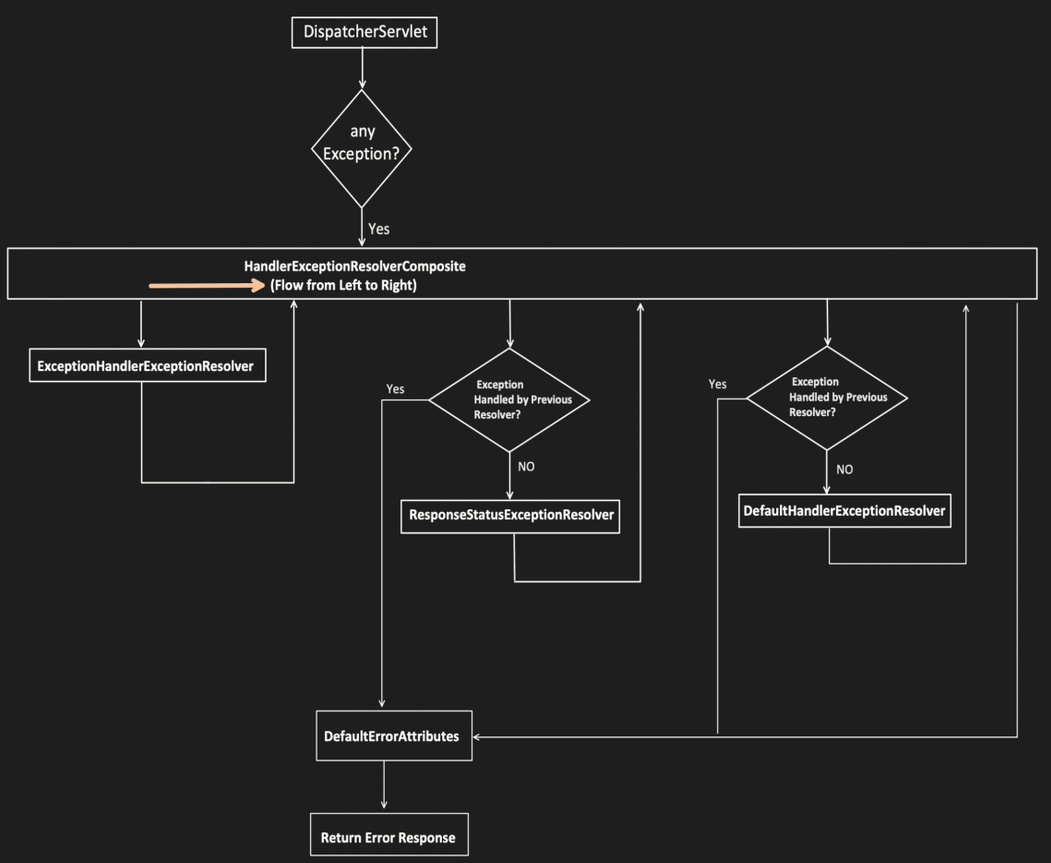
## Classes involved in handling an Exception:



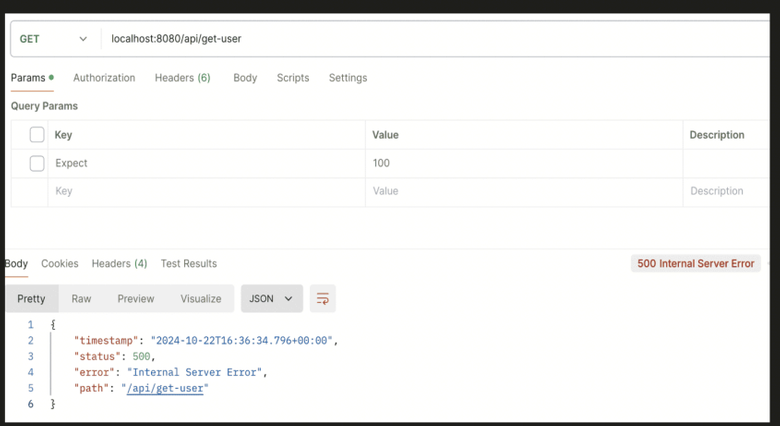

## Let's understand the sequence, when any exception occurs:



## Let's see the flow with an example:

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

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

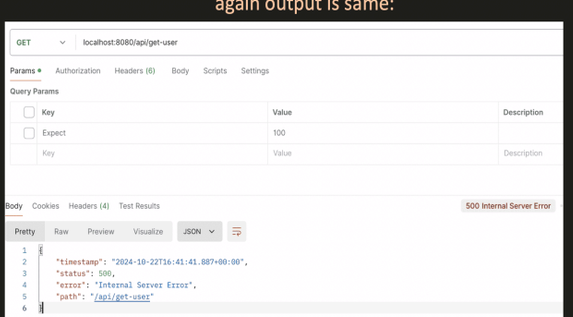
## Output:




```java
@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");
    }
}
```

```java
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" i.e. 400 and my error message  is not shown in output?

In both the example, we are not creating the **ResponseEntity** object, we are just returning the exception, some other class is creating the ResponseEntity Object.

If we need full control and don't want to rely on Exception Resolvers, then we have to create the ResponseEntity Object.

```java
@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);
        }
    }
}
```

```java
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;
    }
}
```

```java
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;
    }
}
```

Body Cookies Headers (4) Test Results — 400 Bad Request

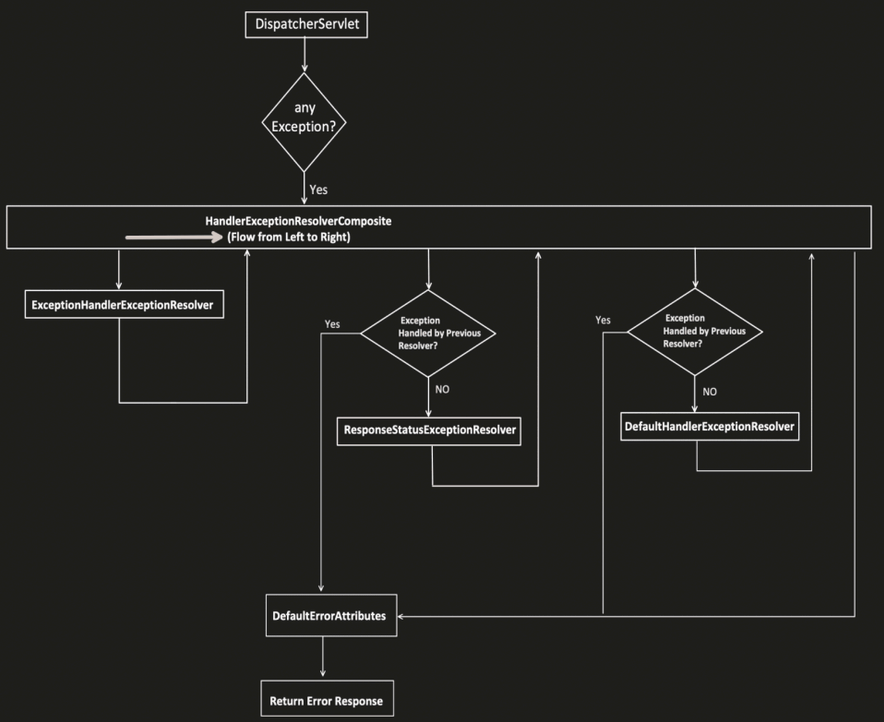
Pretty Raw Preview Visualize JSON

```
{
    "timestamp": "2024-10-25T12:13:02.318+00:00",
    "status": 400,
    "message": "UserID is missing"
}
```

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

```java
@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.



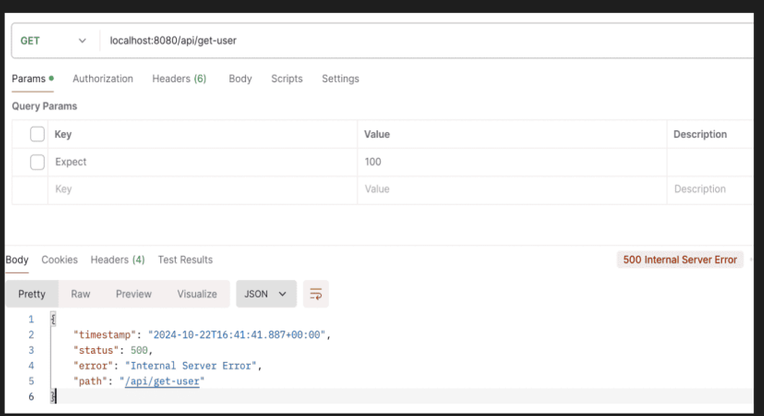

Each Resolver set the proper **status** and **message** in HTTP response for exceptions which they are taking care of.
and
NullPointerException and CustomException all the 3 Resolvers, do not understand, so Status and Message is not set.

So, when control reaches to "**DefaultErrorAttributes" class, it fills the values in HTTP Response with default values.**

# DefaultErrorAttributes

```java
@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;
}
```

```java
@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);
    }
}
```



So, now question is: what exception does "ExceptionHandlerExceptionResolver", "ResponseStatusExceptionResolver" and "DefaultHandlerExceptionResolver" handles?

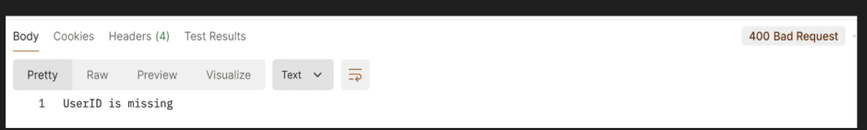
## 1. ExceptionHandlerExceptionResolver

Responsible for handling below annotations:

- @ExceptionHandler
- @ControllerAdvice

**Controller** level Exception handling:

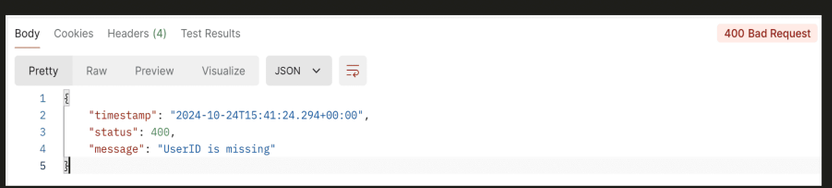
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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());
    }
}
```



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

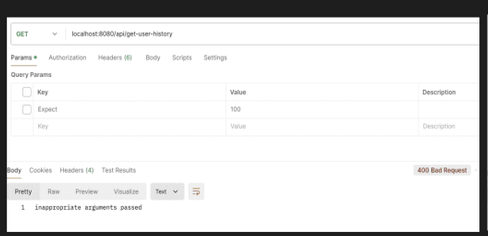
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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());
    }
}
```

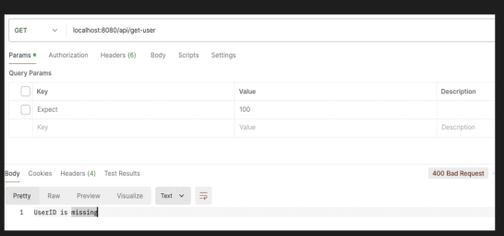
```java
public class ErrorResponse {

    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;
    }
}
```




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

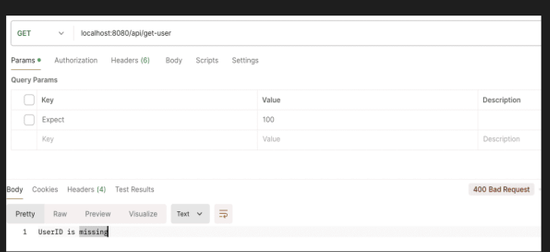
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

    @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);
    }
}
```

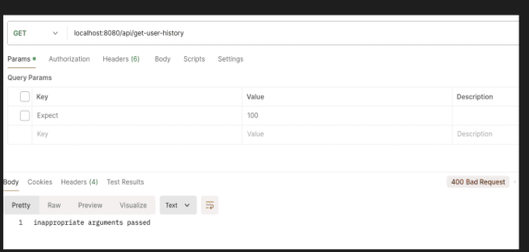


Use-case just to show 1 @ExceptionHandler handling multiple exceptions:

```java
@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<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```



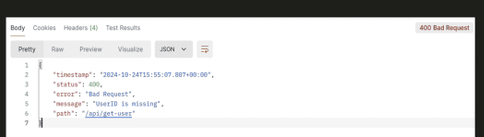

Use-case just to show @ExceptionHandler not returning ResponseEntity and let "**DefaultErrorAttributes**" to create the ResponseEntity.

```java
@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 void handleCustomException(HttpServletResponse 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

```
server.error.include-message=always
```

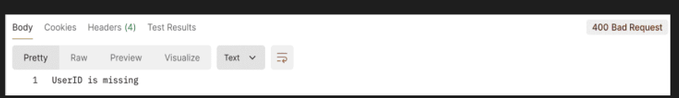


## **Global** Exception handling:

Problem with Controller level @ExceptionHandler is:

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

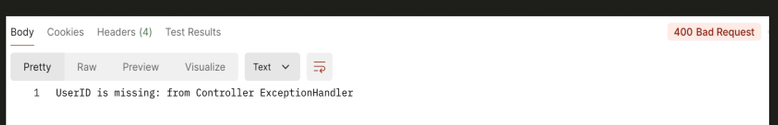
```java
@ControllerAdvice
public class GlobalExceptionHandling {

    @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?

```java
@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());
    }
}
```

```java
@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 hierarch, from bottom to up (first look for exact match if not, check for its parent and so on...)

```java
@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");
    }
}
```

```java
@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);
    }
}
```

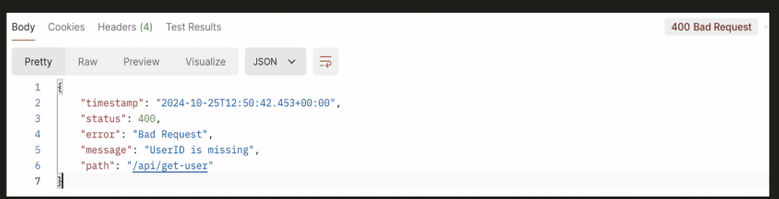
## 2. ResponseStatusExceptionResolver

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

Use-case1: Used above an Exception class

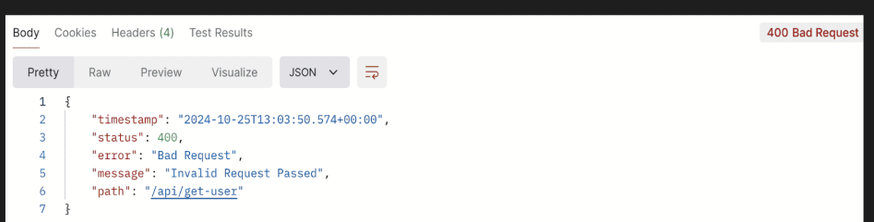
```java
@RestController
@RequestMapping(value = "/api/")
public class UserController {

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

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

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

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




```java
@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");
    }
}
```

```java
@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;
    }
}
```

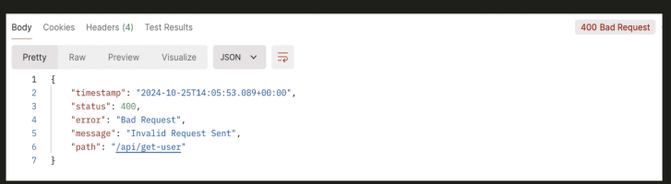


## Use-case2: Used above an @ExceptionHandler method

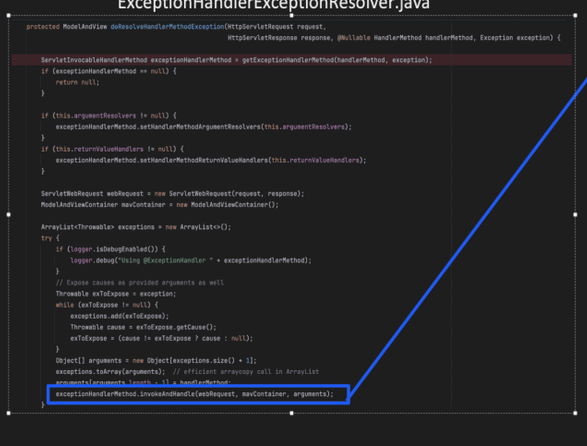
Again ***ResponseStatusExceptionResolver*** handles Uncaught exception annotated with **@ResponseStatus** annotation but if used with ***@ExceptionHandler*** then it will not be handled by "***ResponseStatusExceptionResolver ",*** it will be handled by Spring request handling mechanism itself.

```java
@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 ResponseEntity<Object> handleCustomException(CustomException e) {
        return new ResponseEntity<>( body: "you are not authorized", HttpStatus.FORBIDDEN);
    }
}
```

```java
public class CustomException extends RuntimeException {

    HttpStatus status;

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

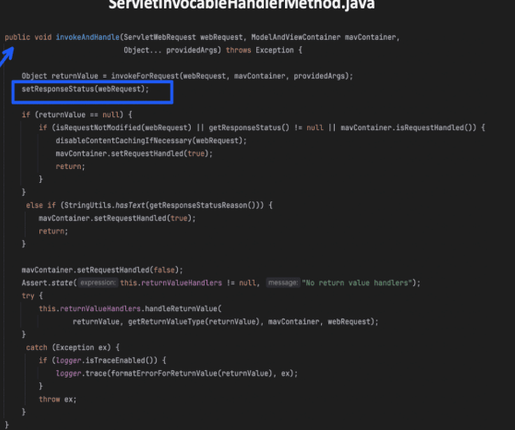



### ExceptionHandlerExceptionResolver.java



```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);
    ModelAndViewContainer mavContainer = new ModelAndViewContainer();

    ArrayList<Throwable> exceptions = new ArrayList<>();
    try {
        if (logger.isDebugEnabled()) {
            logger.debug("Using @ExceptionHandler " + exceptionHandlerMethod);
        }
        // Expose causes 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



```java
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
                            Object... providedArgs) throws Exception {

    Object returnValue = invokeForRequest(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:

```java
@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");
    }
}
```

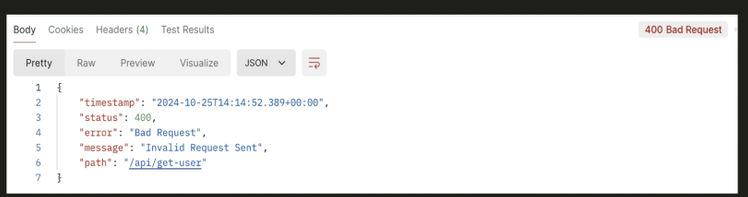

- Its because, Response.sendError first set the status and message in response and do COMMIT.

  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:

```java
@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
    }
}
```




## 3. DefaultHandlerExceptionResolver

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