How Exception works in global handling?

→ Order of exception handling in SpringBoot.

① Controller → Service → Repo (your code)
If something goes wrong we throw an Error.

② Spring catches it

Springs dispatcherServlet catches exceptions.

③ global exception handler Handles it.

Your @RestControllerAdvice acts like a 'catch-all'

④ final JSON response is returned.

You decide -

- HTTP status code
- Error Message
- Error Structure.

→ To do all this we create GlobalExceptionHandler

All good API's return an error model like this JSON

```
{
  "timestamp": "2025-11-16T19:30:50",
  "status": 400,
  "error": "Validation Failed",
  "path": "/students",
  "details": {
    "name": "Name cannot be empty"
  }
}
```

So we create API Error.java

↓

Here we simply create

ApiError.java

```
@JsonPropertyOrder({"timestamp", "status", "error", "path", "details"}) 9 usages
public class ApiError {

    private LocalDateTime timestamp; 2 usages
    private int status; 2 usages
    private String error; 2 usages
    private String path; 2 usages
    private Map<String, String> details; 2 usages

    public ApiError(int status, String error, String path, Map<String, String> details) {
        this.timestamp = LocalDateTime.now();
        this.status = status;
        this.error = error;
```

Variables

Then create a constructor

Then getter methods.

```java
        this.path = path;
        this.details = details;
    }

    public LocalDateTime getTimestamp() {  no usages
        return timestamp;
    }

    public int getStatus() {  no usages
        return status;
    }

    public String getError() {  no usages
        return error;
    }

    public String getPath() {  no usages
        return path;
    }

    public Map<String, String> getDetails() {  no usages
        return details;
    }
```

Later in global exception handler we use this.

```java
GlobalExceptionalHandler.java
@RestControllerAdvice  no usages
public class GlobalExceptionalHandler {

    //    400 Errors
    @ExceptionHandler(MethodArgumentNotValidException.class)  no usages
    public ResponseEntity<ApiError> handleValidationErrors(
            MethodArgumentNotValidException ex, HttpServletRequest request
    ) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(
                FieldError err -> errors.put(err.getField(), err.getDefaultMessage()));

        ApiError apiError = new ApiError(
                HttpStatus.BAD_REQUEST.value(),
                error: "Validation Failed",
                request.getRequestURI(),
                errors
        );

        return ResponseEntity.badRequest().body(apiError);
    }
```

error, → error messages

① Method - 1 -

- handels Invalid arguments
- we use both of these to take in exception messages and status of Request.
- Then we map it to a Map
- Save everything to ApiErrors
- Response Entity return you with Api Errors found

Method - 2.

② StudentNotFound Same inputs

```java
    //   StudentNotFound - 404
    @ExceptionHandler(StudentNotFoundException.class)  no usages
    public ResponseEntity<ApiError> handleStudentNotFound(
            StudentNotFoundException ex,
            HttpServletRequest request
    ){
        ApiError apiError = new ApiError(
                HttpStatus.BAD_REQUEST.value(),
                ex.getMessage(),
                request.getRequestURI(),
                details: null
        );

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(apiError);
    }
```

New ApiError is created

Response body of ApiError with

Status code is returned

③ Method-3
Type Mismatch

- class needed to be called
- argument type.

```java
//    Invalid Type in Path
@ExceptionHandler(MethodArgumentTypeMismatchException.class)  no usages
public ResponseEntity<ApiError> handlesTypeMismatch(
        MethodArgumentTypeMismatchException ex,
        HttpServletRequest request
){
    ApiError apiError = new ApiError(
      HttpStatus.INTERNAL_SERVER_ERROR.value(),
      ex.getMessage(),
      request.getRequestURI(),
       details: null
    );

    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(apiError);
}
```

- New ApiError is created after calling ApiErrors constructor.
- Returns body of Status of Error.

CUSTOM EXCEPTION HANDLERS

↳ we normally handle errors via custom handlers such as Already exists Exception etc

① AlreadyExistsException

```java
package com.example.Day7.exception;

public class AlreadyExistsException extends RuntimeException{
    public AlreadyExistsException(String message) {  1 usage
        super(message);
    }
}
```

returns the String we type while throwing exceptions

errors on Runtim

like this →

```java
public StudentResponseDTO addStudent(StudentRequestDTO dto) {  1 usage  Pankaj-Singh-Rawat
    repo.findByEmail(dto.getEmail()).ifPresent( Student existing -> {
        throw new AlreadyExistsException("Student Already Exists: " + dto.getEmail());
    });

    Student student = mapToEntity(dto);
    Student saved = repo.save(student);
    return mapToResponseDTO(saved);
}
```

Similarly we have —

(2) BadRequestException

BadRequestException.java

```java
package com.example.Day7.exception;

public class BadRequestException extends RuntimeException{
    public BadRequestException (String message){  no usages
        super(message);
    }
}
```

(3) ResourceNotFoundException.

ResourceNotFoundException.java

```java
package com.example.Day7.exception;

public class ResourceNotFoundException extends RuntimeException{
    public ResourceNotFoundException(String message){  1 usage
        super(message);
    }
}
```

```java
    public StudentResponseDTO getStudentById(Long id){  1 usage  Pankaj-Singh-Rawat *
        Student student = repo.findById(id).orElseThrow(() ->
                new ResourceNotFoundException("Student Not Found with id: " + id));
        return mapToResponseDTO(student);
    }
```