Centralized Exception Handling & Logging

ADR Service Base Spring Boot Java Backend Project

📌 Overview

This document outlines the **centralized exception handling and logging mechanism** implemented in ADR Base Service Spring Boot-based application. It ensures that all runtime errors are handled uniformly, logged properly, and returned to the client in a consistent structure.

Exception Handling

📌 Goals

- Centralize exception handling using "GlobalExceptionHandler".
- Generate user-friendly and developer-informative error responses with correlation IDs.
- Support standard HTTP error codes (400, 404, 405, 412, 415, 500).
- Validation feedback using a structured "ValidationDetails" model.
- Avoid boilerplate try-catch in every controller.

📁 Project Structure

src > main > java > package > exception >

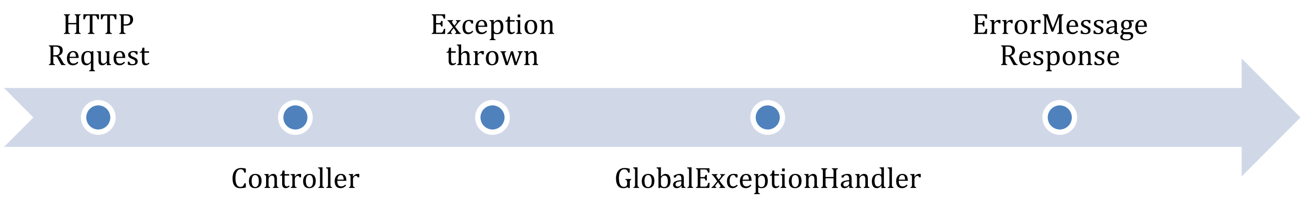
- errorUtil > ...
- ApiCustomException
- GlobalExceptionHandler

Component	Description
GlobalExceptionHandler	A @ControllerAdvice class that catches and processes exceptions globally.
ApiCustomException	Custom exception for application-specific errors.
ErrorMessage	POJO representing the error response.
ValidationDetails	Encapsulates field-specific error details.
ErrorConstants	Central store of error code keys.
ErrorHandlingUtil	Utility class to fetch messages and build validations.

📌 Exception Handling Scenarios

HTTP Status	Scenario	Description
500	Generic Exception	Unhandled exceptions return a 500 error.
405	Method Not Allowed	Triggered by invalid HTTP method.
415	Unsupported Media Type	Triggered when incorrect content type is used.
400	Invalid Input (Validation)	Triggers when request body fails validation.
404	Custom Not Found	Triggered when resource not found in DB.
412	Precondition Failed	Triggered by client version mismatch or duplicate.

📌 Exception Flow



📌 Error Response formats

```
{
   "requestId": "9d765ec5-bad3-4173-9c8d-570d80abd0e9",
   "errorCode": "EN004",
   "errorMessage": "Invalid request parameters provided",
   "errorSeverity": "error",
   "errorStatus": 400,
   "errorDetails": [
                     {
                        "code": "R001",
                        "data": {
                                   "field": "dateOfBirth",
                                   "format": "yyyy-MM-dd"
                                 },
                        "message": "Please provide valid value for 'dateOfBirth', in format 'yyyy-MM-dd'"
                     }
                   ]
 }
```

📌 Error Response Field

Response Field	Description
requestId	A random generated UUID for correlation
errorCode	Application-specific error code
errorMessage	User-readable error message
errorSeverity	Hardcoded as "ERROR"
errorStatus	Corresponding HTTP status code
errorDetails	Optional list of validation failure details

📌 Test Controller Use Case

- Generic 500 Error

```
@GetMapping("/generic")

public String throwGenericException () {

        throw new RuntimeException ("Generic internal error occurred.");

}
```

📌 Testing Each Scenario

URL	HTTP Method	Expected Status	Description
/api/test/generic	GET	500	Generic exception thrown.
/api/test/method-not-allowed	GET	405	Use GET instead of POST.
/api/test/media-type	POST	415	Send request with wrong media type.
/api/test/validation	POST	400	Send invalid input data.
/api/test/not-found/{id}	GET	404	ID not found.
/api/test/precondition-failed	GET	412	Missing/invalid X-Client-Version.
/api/test/duplicate-check	POST	412	Send duplicate email.
/api/test/success	GET	200	Happy path.

📌 Centralized handler Use Case

```
@ControllerAdvice

public class GlobalExceptionHandler {

        @ExceptionHandler(Exception.class)

        public ResponseEntity<ErrorMessage> handleAnyException (Exception ex) {

                return buildErrorResponse (HttpStatus.INTERNAL_SERVER_ERROR,
                "ERR_500", "Unexpected error", null);

}       ---> Other handlers...
```

📌 Benefits

- Single point of control for all exceptions.
- Improves maintainability and consistency.
- Easily extendable to add more business rules.
- Improves debuggability with detailed logs and request identifiers.
- Cleaner controller code.

Logs Mechanism

📌 Goals

- Consistent and human-readable logs with color-coded formatting
- Unique traceability with using traceId
- Dynamic project name injection into log pattern
- Structured in such way that includes all important details yet easy to grasp quickly

📌 Logs Setup

- SLF4J + Logback Configuration

- **SLF4J** (Simple Logging Facade for Java) provides a generic interface for logging.
- **Logback** is the chosen implementation backend. Logback is configured programmatically to include enhanced formatting and traceability.

📁 Project Structure

src > main > java > package > logging >

- Logged
- LoggedAspect
- LoggedConfig
- TraceIdFilter

📌 Log Aspect Use Case

- LoggedAspect is the main key class behind central logging mechanism which handles all the aspects of managing and displaying logs across the application using @Aspect annotation

```
@Aspect

public class LoggedAspect {

...

}
```

📌 Custom Logs Pattern

- **LoggedConfig** class plays a vital role to setup custom patterns and required configurations using @Configuration

```
encoder. setPattern (
 "%cyan([%d{yyyy-MM-dd HH:mm:ss.SSS}]) " +        - Timestamp in cyan (sky blue)
    "%highlight(%-5level) " +                     - Level in green/yellow/red/etc.
    "[traceId = %X{traceId}] " +                  - Trace ID (default color)
    "%blue([" + projectName + "]) " +             - Project name in blue
    "%cyan(%class -> %M) " +                      - Class with package → Method in cyan
    ":%line " +                                   - Line number (default color)
    "%yellow(error=%X{errorCode}) " +             - error in yellow
    "%msg%n" );                                   - Message (default terminal color)
```

📌 Trace ID Generation

A unique traceId is generated for each request and added to **MDC (Mapped Diagnostic Context)** using a Filter.

```
@Component
public class TraceIdFilter implements Filter {
  public void doFilter (ServletRequest req, ServletResponse res, FilterChain chain) {
    try {
      MDC.put ("traceId", UUID.randomUUID(). toString ());
      chain. doFilter (req, res);
    } finally {
      MDC.clear();
    }
  }
}
```

📌 Sample Log

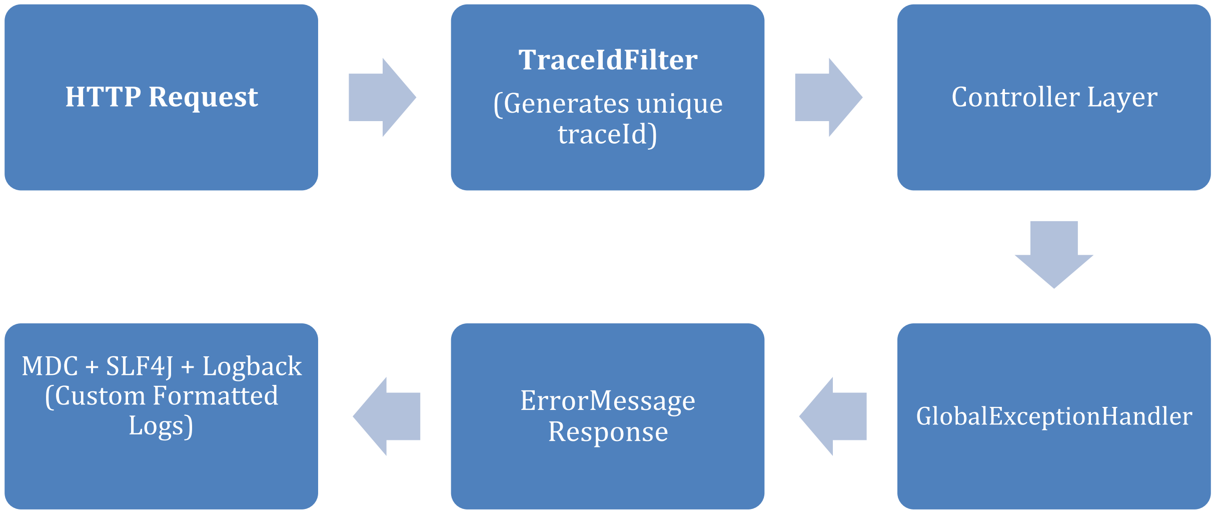
```
[2025-07-27 21:10:03.456] ERROR [traceId = 4078402a-cf89-4033-a9ec-4256ae2267fd] [Project Name] Class name with package -> Method name : 17 error = [404 NOT_FOUND] - Entity Not Found | Employee not found with ID: 4
```

📌 Dynamic Project Name

Project name is dynamically derived from the **current working directory name**, avoiding hardcoding.

String projectName = new File (System.getProperty("user.dir")). getName ();

📌 Logging Flow



📌 Benefits

- Unified logging across modules
- Easy debugging with traceId
- Highly structured and user-friendly error responses
- Extensible for microservices or cloud-native apps
- Environment-agnostic setup