**Handle API Errors by overwriting error class**

class ApiError extends Error {

    constructor(

        statusCode,errors = [],message = "Something went wrong",stack = ""

    ) {

        super(message);

        this.statusCode = statusCode;

        this.errors = errors;

        this.data = null;

        this.message = message;

        this.success = false;

        if (stack) {

            this.stack = stack;

        } else {Error.captureStackTrace(this, this.constructor);}}}

export { ApiError };

The line

Error.captureStackTrace(this, this.constructor);

is used to control the stack trace of the error for debugging purposes. Let’s break it down step-by-step:

**What is Error.captureStackTrace?**

Error.captureStackTrace is a Node.js-specific method that allows you to create a stack trace explicitly. It attaches a stack trace to an error object, which shows the sequence of function calls that led to the error.

**Syntax:**

Error.captureStackTrace(targetObject[, constructorOpt]);

* **targetObject**: The object that will receive the stack trace. In this case, this refers to the current ApiError instance.
* **constructorOpt** (optional): The constructor to exclude from the stack trace. This means the stack trace will omit the frame for the specified constructor.

**How It Works in the Code**

Error.captureStackTrace(this, this.constructor);

* **this**: Refers to the instance of ApiError being created. The stack trace will be attached to this object.
* **this.constructor**: Refers to the ApiError class itself. By passing this as the second argument, the ApiError constructor is omitted from the stack trace, making it cleaner and easier to debug.

**Why Use Error.captureStackTrace?**

1. **Cleaner Stack Traces**:
   * Normally, when you create an error, the constructor where the error was created would appear in the stack trace. By excluding the ApiError constructor, the stack trace starts at the point where the error was actually thrown, rather than where it was constructed.

Example without exclusion:

ApiError: Something went wrong

at new ApiError (/path/to/file.js:10:12)

at someFunction (/path/to/anotherFile.js:20:5)

...

Example with exclusion:

ApiError: Something went wrong

at someFunction (/path/to/anotherFile.js:20:5)

...

1. **Custom Errors**:
   * When creating custom error classes like ApiError, you might want the error to behave just like a native Error but with additional properties or custom behavior.
2. **Node.js Optimization**:
   * Error.captureStackTrace is optimized for performance and memory usage in Node.js. It avoids generating a stack trace for every error unless explicitly requested.

**Alternative Without Error.captureStackTrace**

If you didn’t use this line, the stack trace would still be captured automatically, but it might include unnecessary frames, such as the ApiError constructor itself.

**When Would stack Be Empty?**

In the code:

if (stack) {

this.stack = stack;

} else {

Error.captureStackTrace(this, this.constructor);

}

* If the stack parameter is provided when creating the ApiError, it uses the provided stack trace.
* If stack is not provided, it generates a fresh stack trace using Error.captureStackTrace.

This allows flexibility for scenarios where a custom stack trace needs to be passed or dynamically generated.

**The line**:

this.data = null;

is initializing a property called data on the ApiError instance and setting its initial value to null. Let’s break it down:

### ****Purpose of**** this.data

The data property is often used to store additional information or context about the error that doesn't fit into message, statusCode, or errors. While it is set to null by default in this class, the idea is that it can be assigned a meaningful value when needed, depending on the specific use case.

### ****Why Initialize**** data****?****

1. **Future Flexibility**:
   * By declaring this.data upfront, the class anticipates scenarios where extra context might need to be included in the error object.
   * This avoids errors where accessing data might fail if it hasn't been initialized yet.
2. **Consistent Error Structure**:
   * Ensuring that all errors have the same set of properties (e.g., statusCode, message, errors, data) simplifies debugging and error-handling logic.
   * For example, in an API response, the data field can consistently exist, even if its value is null.

### ****When and How to Use**** data****?****

The data field can be useful when additional details about the error are required. Examples include sending back debugging information, request metadata, or even partial results.

#### Example Use Case:

try {

    // Simulate an error with some additional context

    throw new ApiError(500, [], "Database connection failed");

} catch (error) {

    error.data = { host: "db-server", port: 5432 }; // Attach additional details

    console.error(error);

}

In the above example:

* The data property provides additional debugging information about the database connection that failed (host and port).

### ****Why Set It to**** null ****Initially?****

Setting it to null ensures that:

1. The property exists on all ApiError instances.
2. It has a defined (albeit empty) value if not explicitly set later.

Without initialization, accessing data on an error instance could result in undefined or errors in strict type-checking scenarios.

### ****Summary****

this.data = null; is a placeholder property for optional, additional error-related information. It provides a mechanism for extending error details without modifying other properties like message or errors.

**asyncHandler**

const asyncHandler = (requestHandler) => {

    return (req,res,next) => {

        Promise

        .resolve(requestHandler(req,res,next))

        .catch((error)=>next(error))

    }

}

export { asyncHandler }

This code defines a higher-order function called asyncHandler. It simplifies error handling in asynchronous request handlers, particularly in frameworks like **Express.js**. Let’s break it down step by step:

### ****Purpose of**** asyncHandler

When using asynchronous functions (e.g., async/await) in route handlers, unhandled errors inside the function won't automatically be passed to the error-handling middleware in Express. Instead, you must explicitly use try-catch blocks or forward errors to next().

asyncHandler eliminates the need for repetitive try-catch blocks by wrapping the asynchronous function and automatically catching errors.

### ****Code Walkthrough****

#### ****The Wrapper Function****

const asyncHandler = (requestHandler) => {

* **asyncHandler** is a higher-order function that takes a single argument, requestHandler.
* **requestHandler**: This is an asynchronous function (e.g., an Express route handler).

#### ****Returned Middleware****

return (req,res,next) => {

* It returns a new middleware function that takes the standard Express arguments: req (request), res (response), and next (next middleware).

#### ****Promise Resolution and Error Catching****

Promise

        .resolve(requestHandler(req,res,next))

        .catch((error)=>next(error))

* **Promise.resolve**:
  + Ensures that requestHandler(req, res, next) is treated as a Promise, whether it’s an async function or a regular function.
  + If requestHandler returns a value or resolves successfully, it’s handled normally.
* **.catch**:
  + If the requestHandler throws an error or the Promise it returns rejects, the error is caught here.
  + The caught error is passed to next(error), which triggers the Express error-handling middleware.

### ****How It Works in Express.js****

Here's how you would use asyncHandler in an Express app:

#### Without asyncHandler

app.get('/example', async (req, res, next) => {

    try {

        const data = await someAsyncOperation();

        res.json({ success: true, data });

    } catch (error) {

        next(error); // Manually forwarding the error

    }

});

#### With asyncHandler

import { asyncHandler } from './asyncHandler';

app.get('/example', asyncHandler(async (req, res, next) => {

    const data = await someAsyncOperation();

    res.json({ success: true, data });

}));

* **Benefits**:
  + No need for repetitive try-catch blocks.
  + Errors are automatically forwarded to next().

### ****Example with Full Workflow****

#### Error-Handling Middleware

Define an error-handling middleware in Express:

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).json({ success: false, message: err.message });

});

#### Asynchronous Route

Using asyncHandler in a route:

import { asyncHandler } from './asyncHandler';

app.get(

    '/user/:id',

    asyncHandler(async (req, res) => {

        const user = await getUserById(req.params.id); // Asynchronous operation

        if (!user) {

            throw new Error('User not found'); // Error gets forwarded automatically

        }

        res.json({ success: true, user });

    })

);

#### Explanation:

1. If getUserById resolves successfully, the route sends a successful response with the user data.
2. If getUserById rejects (e.g., the user doesn't exist), the error is caught by .catch and passed to next().
3. The Express error-handling middleware sends the appropriate error response.

### ****Summary****

The asyncHandler function:

* Wraps asynchronous route handlers.
* Automatically catches and forwards errors to the next middleware.
* Eliminates the need for repetitive try-catch blocks, making the code cleaner and more maintainable.

**Standardized Api Response**

class ApiResponse{

    constructor(statusCode ,message='Success',data ){

        this.statusCode = statusCode

        this.message = message

        this.data = data

        this.success = statusCode < 400

    }

}

export {ApiResponse}

This code defines a class called ApiResponse that standardizes the structure of responses sent back to the client in an application, typically an API. Here's a detailed explanation of the code:

### ****Purpose of**** ApiResponse

The ApiResponse class is designed to encapsulate all the necessary information for an HTTP response in a consistent format. This approach ensures that all responses from the server have a predictable structure, making it easier for clients to handle and interpret them.

### ****Code Walkthrough****

#### Class Definition

class ApiResponse {

* Defines a class named ApiResponse.
* This class is used to create instances representing API responses.

#### Constructor

constructor(

    statusCode,

    message = 'Success',

    data

) {

* The constructor is called when a new ApiResponse instance is created.
* It takes three parameters:
  1. **statusCode**: (Required) The HTTP status code for the response (e.g., 200, 404, 500).
  2. **message**: (Optional, defaults to 'Success') A human-readable message describing the outcome of the request.
  3. **data**: (Optional) Any data payload to be sent with the response (e.g., user details, a list of items, etc.).

#### Property Initialization

this.statusCode = statusCode;

this.message = message;

this.data = data;

this.success = statusCode < 400;

* **this.statusCode**: Stores the provided HTTP status code.
* **this.message**: Stores the provided message or the default message 'Success'.
* **this.data**: Stores the data payload, which can be anything (e.g., null, an object, or an array).
* **this.success**: A boolean flag automatically set to true if the status code is less than 400 (indicating a successful response) and false otherwise (for client or server errors).

### ****How It Works****

#### Example Usage in an API Controller:

Here's how you might use the ApiResponse class in an Express.js route handler:

import { ApiResponse } from './ApiResponse';

// Success Example

app.get('/api/data', (req, res) => {

    const data = { id: 1, name: 'John Doe' };

    const response = new ApiResponse(200, 'Data fetched successfully', data);

    res.status(response.statusCode).json(response);

});

// Error Example

app.get('/api/error', (req, res) => {

    const response = new ApiResponse(404, 'Resource not found', null);

    res.status(response.statusCode).json(response);

});

#### Outputs:

1. **Success Response:**

{

    "statusCode": 200,

    "message": "Data fetched successfully",

    "data": { "id": 1, "name": "John Doe" },

    "success": true

}

1. **Error Response:**

{

    "statusCode": 404,

    "message": "Resource not found",

    "data": null,

    "success": false

}

### ****Advantages****

1. **Consistency**:
   * All responses follow a uniform structure, making it easier for clients to parse and handle them.
2. **Automatic Success Calculation**:
   * The success property is automatically derived based on the statusCode, reducing the chance of inconsistencies.
3. **Customizable Messages**:
   * Developers can provide custom messages for different scenarios.
4. **Encapsulation**:
   * Encapsulates all response-related logic in a single class, keeping controllers and routes cleaner.

### ****Summary****

The ApiResponse class:

* Standardizes the structure of API responses with statusCode, message, data, and success properties.
* Ensures consistency and reduces boilerplate code in API response handling.
* Automatically determines success based on the status code.

It’s a clean and efficient way to handle responses in an API-driven application.