**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**