What is Error Handling?

Error handling is the process of identifying and managing problems (called "errors" or "exceptions") in a backend application when something goes wrong.

It ensures that your backend:

- Doesn’t crash unexpectedly
- Sends a meaningful message to the user or frontend
- Logs the error for you to debug later

Why is Error Handling Important?

Reason Description

 Prevents Ensures server stays up even when errors crashes occur

 User-friendly Shows useful messages instead of vague errors

 Easier to debug Errors can be logged and traced

 Professional Maintains a clean, secure, and stable system

How to implement Error Handling:

Error handling is done using three main blocks:

- try: Contains code that might cause an error.
- catch: Catches the error if it occurs and allows you to handle it.
- finally: Runs code regardless of whether an error occurred, often used for clean-up tasks.

```
app.get('/api/event/:id', async (req, res) => {
  try {
    const event = await Event.findById(req.params.id);
    if (!event) {
      return res.status(404).json({ message: 'Event not found' });
    }
    res.status(200).json(event);
  } catch (err) {
    res.status(500).json({ message: 'Something went wrong' });
  }
});
```

This works, but writing try-catch in every route becomes messy, repetitive, and unscalable.

EXCEPTION WRAPPER (CENTRALIZED ERROR HANDLING)

An Exception Wrapper is a backend pattern where all application errors are handled in one place instead of writing try-catch everywhere. It uses:

- A custom error class
- An async handler utility A global error middleware

This pattern makes your backend:

- Cleaner
- Easier to debug
- More consistent How

to Implement:

A.Create Custom Error Class:

* CustomError is a class that extends JavaScript’s built-in Error.

* It adds a statusCode to show error types (like 404 or 500).

* You can set both an error message and a status code.
* This helps in handling errors better, especially in APIs.

* Makes it easier to manage and understand errors in the code.

```
// utils/customError.js
class CustomError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
  }
}

module.exports = CustomError;
```

```
// utils/asyncHandler.js
module.exports = fn => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next);
```

B. Create Async Handler Utility:

* AsyncHandler handles errors in async functions automatically.

* It wraps the async function and catches any errors.

* Caught errors are passed to the error-handling middleware with .catch (next).

Wraps async functions to auto-forward errors to middleware. C.

Global Error Middleware:

* `error Handler` catches and handles errors in the application.

* It sets the HTTP status code (default is `500` if not provided).
* Returns a JSON response with error details like message, status code, and timestamp.

* Helps manage and send consistent error responses to clients.

```
const errorHandler = (err, req, res, next) => {
  const statusCode = err.statusCode || 500;

  // Internal logging
  console.error(`[${new Date().toISOString()}]`, err.stack || err.message);

  // User response
  res.status(statusCode).json({
    success: false,
    message: statusCode === 500 ? 'Internal Server Error' : err.message,
    statusCode,
    timestamp: new Date().toISOString(),
  });
};

module.exports = errorHandler;
```

This sends all error responses in one clean format.

Usage in Routes:

```
router.get('/:id', asyncHandler(async (req, res, next) => {
  const event = await Event.findById(req.params.id);
  if (!event) {
    throw new CustomError("Event not found", 404);
  }
  res.status(200).json(event);
}));

module.exports = router;
```

Sample Error Response:

```
{
  "success": false,
  "message": "Event not found",
  "statusCode": 404,
  "timestamp": "2025-04-17T12:00:00Z"
}
```

Q. How should API errors be displayed to users (e.g., toast notifications)?

Soln:

Backend errors should not just sit in the console — they should be shown to the user in a friendly way without technical jargon.

Why?

- Users don't care about stack traces or error codes.
- They need to know: What went wrong? and What to do next?

So instead of displaying raw error messages, we catch API errors and display them using UI-friendly components like:

- Toast notifications (popup alerts)
- Inline messages near forms

Example:

1. Assume your backend sends this JSON on error:

```
{
  "success": false,
  "message": "Invalid credentials",
  "statusCode": 401
}
```

2. We will use toast notification to show this on frontend:

```jsx
import axios from 'axios';
import { toast } from 'react-toastify';
import 'react-toastify/dist/ReactToastify.css';

function LoginForm() {
  const handleLogin = async () => {
    try {
      await axios.post('/api/login', { username: 'admin', password: '1234' });
    } catch (error) {
      // Show toast with backend error message
      toast.error(error.response.data.message || 'Something went wrong');
    }
  };

  return <button onClick={handleLogin}>Login</button>;
}
```

If login fails due to invalid credentials, the user sees:

🔔 "Invalid credentials"

Q. How do you log errors centrally while avoiding sensitive data leaks?

Soln:

Logging errors centrally means saving all error details in one place (like logs or monitoring tools) so developers can track and fix issues easily.

But here's the catch:

We must never leak sensitive information (like passwords, tokens, stack traces) in error logs that might be seen by users or exposed online.

```
const errorHandler = (err, req, res, next) => {
  const statusCode = err.statusCode || 500;

  // Log everything (internally)
  console.error(`[${new Date().toISOString()}]`, err.stack || err.message);

  // Send safe error response to the client
  res.status(statusCode).json({
    success: false,
    message:
      statusCode === 500 ? 'Internal Server Error' : err.message,
    statusCode,
    timestamp: new Date().toISOString(),
```

For developer: The error will be like

```
[2025-05-25T12:24:00.456Z] Error: Database connection failed at DB.js:14
```

For the user:

```
{
  "success": false,
  "message": "Internal Server Error",
  "statusCode": 500,
  "timestamp": "2025-05-25T12:24:00.456Z"
}
```

Q. What are common pitfalls in error handling to avoid?

Soln:

1.Using Generic or Wrong HTTP Status Codes

Pitfall: Returning 200 OK even when there's an error, or always using 500 for all failures.

Fix: Use specific status codes:

- 400 for bad input
- 401 for unauthorized
- 404 for not found

Exposing Sensitive Error Details:

Pitfall: Sending stack traces or database error messages to the client.

Fix: Hide technical details in production and log them internally instead.

```
res.status(500).json({ message: "Something went wrong!" });
```

Handling Errors Too Late (Missing Next in Middleware):

Pitfall: Errors never reach the global error handler if you forget to call next(error) Fix: Always call next.

Inconsistent Error Structure :

Pitfall: Different routes return errors in different formats.

Fix: Use a **standard JSON format** via middleware:

Best Practices:

- Use `asyncHandler` to simplify routes
- Structure all errors using `CustomError` • Never leak stack traces to the client.

HTTP STATUS CODES:

HTTP status codes are three-digit numbers returned by the server to indicate the result of an API request. They help the client know:

- Was the request successful?
- Was the input wrong?
- Did the server crash?

Why Are They Important?

- Help frontend apps understand API response
- Let mobile/web apps show proper messages to users
- Follow RESTful conventions
- Improve debugging and error tracking

Where Are They Used?

In every backend API, for every request:

- Login (401 if invalid credentials)
- Data fetch (404 if not found)
- Form submit (400 if input is missing)
- Server crash (500)

Common HTTP Status Codes Table

Code	Meaning	Use Case
200	OK	Successful GET or PUT request
201	Created	Resource successfully created (POST)
204	No Content	Request succeeded, no body returned
400	Bad Request	Invalid input from client
401	Unauthorized	Login required / token missing
403	Forbidden	Logged in, but access denied
404	Not Found	Resource does not exist
500	Internal Server Error	Backend crashed or bug occurred

How to Use in Code:

```
if (!req.body.email) {
  throw new CustomError("Email is required", 400);
}

const user = await User.findById(id);
if (!user) {
  throw new CustomError("User not found", 404);
}

res.status(201).json({ message: "User created" });
```

Best Practices:

- Always send the correct code (don't use 200 for all)
- Use 500 only for unknown/unexpected server errors
- Avoid sending sensitive error info with 500s
- Combine with structured JSON error messages

What is Winston?

Winston is a powerful logging library for Node.js that lets you:

Save logs to console, file, or even remote services

Set different log levels (info, error, warn, debug, etc.)

Create timestamped logs

Format logs cleanly and even as JSON