# Error Handling in Express

## 1. Problem Statement

Web applications inevitably encounter errors—bad user input, failed database calls, or bugs in code. If not handled properly, these errors can crash your server or expose sensitive stack traces to users.

**The challenge:**
How do you design an Express.js application that gracefully catches, logs, and responds to errors—whether they are synchronous or asynchronous—without leaking sensitive information or breaking the user experience?

## 2. Learning Objectives

By the end of this lesson, you will be able to:

- Explain the importance of error handling in Express.js.
- Distinguish between synchronous and asynchronous errors in Express.
- Use Express's default error handler and implement custom error-handling middleware.
- Pass errors using next(err) and handle them centrally.
- Structure error responses for both debugging and user-friendliness.
- Apply best practices for robust, maintainable error handling.

## 3. Concept Introduction with Analogy

### Analogy: Safety Nets in a Circus

Imagine a trapeze artist performing high above the ground. If something goes wrong, a safety net catches them, preventing injury. In Express.js, error handling middleware acts as a safety net for your application—catching errors before they can crash your server or reach your users in an uncontrolled way.

- **No safety net:** The performer (your app) falls and gets hurt (crashes or leaks info).
- **With a safety net:** Even if mistakes happen, the show goes on safely.

## 4. Technical Deep Dive

### What is Error Handling in Express?

Error handling is the process of detecting, catching, and managing errors that occur during the request-response cycle. Express provides both a built-in error handler and the ability to define custom error-handling middleware.

### How Error Handling Works

- **Synchronous Errors:**
  Thrown in route handlers or middleware, these are automatically caught by Express and passed to the error handler.

- **Asynchronous Errors:**
  Errors in async code (promises, async/await) are not caught by default in Express 4. You must pass them to the error handler using next(err). Express 5 will catch async errors automatically.

- **Error-Handling Middleware:**
  Special middleware with four arguments: (err, req, res, next). It must be registered after all other routes and middleware.

#### Default Error Handler

If you don’t define your own error handler, Express will send a generic 500 Internal Server Error, showing the stack trace in development but hiding it in production.

#### Custom Error-Handling Middleware

You can create middleware to log errors, customize responses, and hide sensitive details:

```
app.use((err, req, res, next) => {
  console.error(err.stack); // Log for debugging
  res.status(500).json({ message: "Oops! Something went wrong." });
});
```

#### Synchronous Error Example

```
app.get("/", (req, res) => {
  throw new Error("Something went wrong!");
});
```

Express catches this and forwards it to the error handler.

#### Asynchronous Error Example

```
app.get("/async-error", async (req, res, next) => {
  try {
    await Promise.reject(new Error("Async error occurred!"));
  } catch (err) {
    next(err); // Pass to error handler
  }
});
```

#### Handling Errors with next(err)

Any time you call next(err), Express skips remaining handlers and jumps to the error middleware.

#### Best Practices

- Always register error-handling middleware last.
- Use next(err) for both sync and async errors.
- Hide stack traces in production.
- Structure error responses for clarity.
- Optionally, use packages like express-async-handler for cleaner async routes.

---

## 5. Step-by-Step Data Modeling & Code Walkthrough

#### 1. Default Error Handler

```
const express = require("express");
const app = express();

app.get("/", (req, res) => {
  throw new Error("Something went wrong!");
});

app.listen(3000, () => {
  console.log("Server running on port 3000");
});
```

- Express sends a 500 error response, stack trace visible in development.

### 2. Custom Error Middleware

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: "Oops! Something went wrong." });
});
```

- Place after all routes to catch errors and send a user-friendly response.

### 3. Synchronous Error Handling

```
app.get("/sync-error", (req, res) => {
  throw new Error("Synchronous error occurred!");
});
```

- Thrown errors are caught and sent to the error middleware.

### 4. Asynchronous Error Handling

```
app.get("/async-error", async (req, res, next) => {
  try {
    await Promise.reject(new Error("Async error occurred!"));
  } catch (err) {
    next(err);
  }
});
```

- Use try/catch and next(err) for async code in Express 4.

### 5. Manual Error Forwarding

```
app.get("/manual-error", (req, res, next) => {
  const err = new Error("Manually triggered error!");
  next(err);
});
```

- Directly pass errors to the error handler.

### 6. Centralized Error Handler Example

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);
  res.status(err.status || 500).json({
    message: err.message || "Internal Server Error",
    stack: process.env.NODE_ENV === "production" ? "💥" : err.stack,
  });
};
app.use(errorHandler);
```

- Centralizes error handling, hides stack in production.

---

## 6. Common Pitfalls & Best Practices

- **Don’t forget to handle async errors with next(err) in Express 4.**
- **Always register error middleware after all routes.**
- **Never expose stack traces or sensitive data in production.**
- **Centralize error handling for cleaner, more maintainable code.**

- **Use third-party packages like express-async-handler for DRY async error handling.**
- **Catch all unhandled routes and errors for a robust API.**

---

## 7. Quick Recap & Key Takeaways

- Express catches synchronous errors automatically, but async errors require next(err) (in Express 4).
- Custom error-handling middleware gives you control over logging and user responses.
- Always place error middleware last.
- Centralized error handling improves maintainability and reliability.
- Hide sensitive details from users, especially in production.

---