Error Handling in Express & Mongoose — A Beginner’s Handbook

Welcome, aspiring backend developer! You're on an exciting journey to build powerful web applications with Node.js. As you write more code, you'll inevitably encounter one of the most crucial aspects of development: **errors**.

This handbook is your guide to understanding, anticipating, and gracefully managing errors in an Express and Mongoose application. We'll start with the basics and gradually build up to sophisticated, production-ready techniques. By the end, you'll not only fix bugs but also build more resilient and user-friendly APIs.

Table of Contents

Chapter 1: Introduction to Errors in Express

What are errors in backend development?

In the simplest terms, an error is an unexpected state in your application that prevents it from running normally. When a user requests data from your API, they expect a predictable response. An unhandled error breaks this contract, often resulting in a crashed server or a confusing, unhelpful response.

Common Causes of Errors

Errors can come from many places. Let's categorize the most common ones:

- **Syntax Errors:** Typos in your code, like a missing parenthesis or a misspelled keyword. The application usually won't even start.
  - *Example:* const x = 5; if (x == 5 { console.log("Missing bracket!"); }
- **Logical Errors:** The code runs without crashing, but it doesn't do what you intended. These are often the hardest to find.
  - *Example:* A function to calculate a shopping cart total accidentally subtracts an item's

price instead of adding it.
- **Runtime Errors (Exceptions):** Errors that occur while the application is running. This is our main focus.
    - **Validation Errors:** A user sends data in the wrong format (e.g., an email address without an "@" symbol, or a password that's too short).
    - **Database Errors:** The application can't connect to the database, a query fails, or you try to find a record that doesn't exist.
    - **Authorization Errors:** A user tries to access a resource they don't have permission to view.

How Express Handles Errors Internally

Express is built on a system of **middleware**. A middleware is just a function that has access to the request (req), response (res), and the next function in the application’s request-response cycle.

When a request comes in, it flows through a chain of middleware until a response is sent.

Request → Middleware 1 → Middleware 2 → Route Handler → Response

If an error occurs in any of these synchronous functions, Express will skip all remaining normal middleware and jump straight to the **error-handling middleware**.

Chapter 2: The Default Error Handler in Express

Express is smart enough to know that errors happen. It comes with a built-in, default error handler so that your application doesn't just crash silently.

How it Works

If you throw an error in your route handler and don't have any custom error-handling middleware defined, Express will catch it for you.

- It sets the status code to 500 (Internal Server Error).
- It sends a response back to the client. In a development environment, this response is an HTML page containing the error message and a full **stack trace**.

A stack trace is a report of the function calls that were active at the time of the error. It's incredibly useful for debugging but should **never** be shown to end-users in a production application because it can expose sensitive information about your server's structure.

Example: An Unhandled Error

Let's see it in action.

Code:

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

app.get("/broken", (req, res) => {
  // We are intentionally throwing a new error
  throw new Error("This page is broken!");
});

app.listen(8080, () => {
  console.log("Server listening on port 8080");
});
```

Behavior:

1. Run this code and navigate to http://localhost:8080/broken in your browser.
2. The server doesn't crash, but you won't get a nice JSON response.

Output (in browser):
You will see an ugly HTML page with the error message and a long stack trace.

```
<pre>
Error: This page is broken!
   at /home/user/project/app.js:5:9
   at Layer.handle [as handle_request] (.../node_modules/express/lib/router/layer.js:95:5)
   ... (many more lines)
</pre>
```

This is Express's default error handler at work. It's helpful for us developers, but not for our users. We need to take control.

Chapter 3: Custom Error-Handling Middleware

To provide consistent, user-friendly error responses (like a clean JSON object), we must create our own error handler.

An Express error-handling middleware is special. Unlike regular middleware which has 3 parameters (req, res, next), an error handler has **four**.

The 4-Parameter Middleware: (err, req, res, next)

This signature tells Express, "This isn't a regular middleware; it's a place to send all errors."

- err: The error object that was thrown or passed to next().
- req: The request object.
- res: The response object.

- next: A function to pass control to the next error-handling middleware (if any).

Important Rule: The error-handling middleware must be defined **last**, after all other app.use() and route calls. This is because it should only run after an error has occurred in one of the preceding routes.

Sample Custom Error Handler

Let's replace the default handler with our own.

Code:

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

app.get("/broken", (req, res) => {
  throw new Error("This page is broken!");
});

// Our Custom Error Handler
// This MUST be the last middleware
app.use((err, req, res, next) => {
  console.error(err.stack); // Log the error for debugging
  res.status(500).send("Oh no, something went wrong!");
});

app.listen(8080, () => {
  console.log("Server listening on port 8080");
});
```

Behavior:
Now, when you visit http://localhost:8080/broken, you won't see the HTML stack trace.
Output (in browser):
A plain text response:

```
Oh no, something went wrong!
```

This is much better! We've hidden the implementation details and provided a simpler message. But we can make it more structured and dynamic.

Chapter 4: Creating a Custom Error Class

Right now, our handler always sends a 500 status code. But what if the error is a 404 Not Found or a 400 Bad Request? Throwing a generic new Error() doesn't give us a way to attach a status code.

That's why we create a custom error class. It allows us to bundle a **message** and a **status code** together, making our error handler smarter.

Implementation: ExpressError.js

Let's create a new file, perhaps in a utils/ directory.

Code (utils/ExpressError.js):

```
class ExpressError extends Error {
  constructor(status, message) {
    super(); // Call the parent Error constructor
    this.status = status;
    this.message = message;
  }
}

module.exports = ExpressError;
```

This simple class inherits from JavaScript's built-in Error class and adds a status property.

Using ExpressError in Routes

Now we can require this class in our main file and use it. We'll also update our error handler to use the properties from our custom error.

Code (app.js):

```
const express = require("express");
const ExpressError = require("./utils/ExpressError.js"); // Import our custom error
const app = express();

// A route that simulates "Not Found"
app.get("/listings/:id", (req, res) => {
  const { id } = req.params;
  if (id !== "123") {
    throw new ExpressError(404, "Listing Not Found!");
  }
  res.send("Here is your listing...");
});

// A route to simulate a validation error
```

```
app.get("/admin", (req, res) => {
  throw new ExpressError(401, "You are not an Admin!");
});

// Updated Error Handler
app.use((err, req, res, next) => {
  // Set default values if status or message are not provided
  const { status = 500, message = "Something went wrong" } = err;
  res.status(status).json({
    error: {
      message: message,
      status: status,
    }
  });
});

app.listen(8080, () => {
  console.log("Server listening on port 8080");
});
```

Behavior:

- Visiting http://localhost:8080/listings/456 will throw our ExpressError.
- The error handler will catch it, extract the status (404) and message, and send a clean JSON response.

Output (for /listings/456):

```
{
  "error": {
    "message": "Listing Not Found!",
    "status": 404
  }
}
```

This is a professional, predictable API error response!

Chapter 5: Handling Asynchronous Errors

There's a major catch with Express's error handling: **it only automatically catches errors from synchronous code.**

What happens in an async function, like when we're interacting with a database?

The Problem: A Broken Async Route

Consider this route. It tries to find a user in a database (simulated with a delay).

```
app.get("/async-broken", async (req, res) => {
  // This will throw an error, but Express won't catch it!
  const user = await User.findById("invalid-id");
  res.send(user);
});
```

If User.findById() rejects its promise (which it will with an invalid ID), the error occurs in a different part of the event loop. Express's synchronous error handler misses it completely. The request will simply hang and eventually time out, leaving the client confused.

Solution 1: try...catch with next(err)

The traditional solution is to wrap our asynchronous code in a try...catch block. If an error occurs, we manually pass it to Express's error handler using next(err).

Code:

```
app.get("/async-fixed", async (req, res, next) => {
  try {
    const user = await User.findById("invalid-id");
    res.send(user);
  } catch (err) {
    // Pass the error to our error handler
    next(err);
  }
});
```

This works perfectly! But writing try...catch in every single async route is repetitive and clutters our code. We can do better.

Solution 2: The asyncWrap Utility Function

We can create a simple utility function that wraps our async route handlers. This function will attach a .catch() block to the promise returned by our async function, automatically forwarding any errors to next.

This is a higher-order function—a function that returns another function.

Code (utils/asyncWrap.js):

```
function asyncWrap(fn) {
  return function (req, res, next) {
    fn(req, res, next).catch((err) => next(err));
  };
}

module.exports = asyncWrap;
```

How to use it:
Now, we can import asyncWrap and simply wrap our route handler functions with it. No more try...catch blocks!

```
const asyncWrap = require("./utils/asyncWrap.js");

app.get(
  "/listings",
  asyncWrap(async (req, res) => {
    // If Listing.find() fails, the error is automatically caught
    // and passed to our error-handling middleware.
    const listings = await Listing.find({});
    res.json(listings);
  })
);
```

This is clean, reusable, and the modern standard for handling async errors in Express.

Chapter 6: Integrating Error Handling in an Express + Mongoose App

Let's build a small, but complete, CRUD application for "Listings" to see all these concepts work together.

Code Structure:

```
/
|-- app.js
|-- models/
|   |-- listing.js
|-- utils/
|   |-- ExpressError.js
|   |-- asyncWrap.js
```

Code (app.js):

```
const express = require("express");
const mongoose = require("mongoose");
const Listing = require("./models/listing.js");
const ExpressError = require("./utils/ExpressError.js");
const asyncWrap = require("./utils/asyncWrap.js");

const app = express();
app.use(express.json()); // for parsing application/json

async function main() {
  await mongoose.connect("mongodb://127.0.0.1:27017/wanderlust");
}
main().then(() => console.log("Connected to DB")).catch(console.log);

// GET all listings
app.get("/api/listings", asyncWrap(async (req, res) => {
  const listings = await Listing.find({});
  res.json(listings);
}));

// GET a single listing by ID
app.get("/api/listings/:id", asyncWrap(async (req, res) => {
  const { id } = req.params;
  const listing = await Listing.findById(id);
  if (!listing) {
    throw new ExpressError(404, "Listing not found");
  }
  res.json(listing);
}));

// CREATE a new listing
app.post("/api/listings", asyncWrap(async (req, res) => {
  // This route is vulnerable to validation errors
  const newListing = new Listing(req.body);
  await newListing.save();
  res.status(201).json(newListing);
}));
```

```
// Final error handler
app.use((err, req, res, next) => {
  const { status = 500, message = "Something went wrong" } = err;
  res.status(status).json({ success: false, status, message });
});

app.listen(8080, () => {
  console.log("Listening on port 8080");
});
```

This structure is robust. Every async database operation is safely wrapped, and any "Not Found" cases throw a specific ExpressError.

Chapter 7: Handling Mongoose-Specific Errors

Mongoose has its own error types that can occur during database operations. Our error handler can be made even smarter by recognizing these specific errors and formatting them nicely.

Common Mongoose Errors

1. **ValidationError:** Occurs when you try to save a document that violates the schema's rules (e.g., a required field is missing, or a field has the wrong type).
2. **CastError:** Occurs when Mongoose cannot convert a value to the type defined in the schema. The most common case is providing an invalid string as a MongoDB ObjectId.
3. **Duplicate Key Error (Code 11000):** Occurs when you try to save a document with a value that violates a unique index in your schema.

A Smarter Error-Handling Middleware

We can update our final middleware to inspect the err object. We'll check its name property to identify the type of Mongoose error and create a custom response for each.

Code (Updated Error Handler in app.js):

```
app.use((err, req, res, next) => {
  console.error("--- ERROR ---");
  console.error(err);

  // Mongoose Validation Error
  if (err.name === "ValidationError") {
    const errors = Object.values(err.errors).map(el => el.message);
    const message = `Invalid input data. ${errors.join('. ')}`;
    err = new ExpressError(400, message);
  }
  // Mongoose CastError (e.g., invalid ObjectId)
  if (err.name === "CastError") {
    const message = `Invalid ${err.path}: ${err.value}.`;
    err = new ExpressError(400, message);
  }

  // Mongoose Duplicate Key Error
  if (err.code === 11000) {
    const key = Object.keys(err.keyValue)[0];
    const value = err.keyValue[key];
    const message = `Duplicate field value: "${value}" for field "${key}". Please use another
value.`;
    err = new ExpressError(400, message);
  }

  const { status = 500, message = "Something went wrong" } = err;
  res.status(status).json({ success: false, status, message });
});
```

How it works:

- If a ValidationError occurs, we extract all the specific validation failure messages from Mongoose and combine them into a single, helpful message with a 400 Bad Request status.
- If a CastError occurs (e.g., a user provides .../listings/123 instead of a valid ID), we create a clear error telling them which field was invalid.
- This creates a fantastic developer and user experience, as API errors are now specific and actionable.

Chapter 8: Best Practices & Tips

1. **Standardize JSON Error Responses:** Always send errors in the same format. A good structure includes a success flag, status code, and message.
   ```
   {
     "success": false,
     "status": 404,
     "message": "Resource not found"
   }
   ```

2. **Log Errors for Debugging:** Always log the full error object or stack trace on the server (console.error(err) or use a dedicated logging library like Winston or Pino). This is your primary tool for diagnosing problems in production.

3. **Don’t Expose Stack Traces in Production:** Stack traces are a security risk. Use environment variables to conditionally hide them.
```
// In your error handler
const response = {
  success: false,
  status: err.status,
  message: err.message,
};

if (process.env.NODE_ENV === 'development') {
  response.stack = err.stack;
}

res.status(err.status).json(response);
```
4. **Use Environment Variables (NODE_ENV):** Set NODE_ENV to "production" on your live server and "development" on your local machine. This allows your code to behave differently in each environment, like showing more detailed error logs during development.

Chapter 9: Advanced Topics (Optional)

- **Graceful Shutdown:** For critical database connection errors on startup, it's often best to shut down the application process gracefully (process.exit(1)). This prevents the app from running in a broken state and allows process managers (like PM2) or container orchestrators (like Docker) to restart it automatically.
- **Third-Party Libraries:** The asyncWrap utility is great, but for even less boilerplate, you can use the express-async-errors package. Simply require('express-async-errors') at the top of your app.js file, and it will automatically patch Express to handle promise rejections in async routes. No wrappers or try...catch blocks needed!
- **Unit Tests for Error Handling:** Using a testing framework like Jest and Supertest, you can write tests to ensure your error handling works as expected.
  - **Test Case Example:** "It should return a 404 error if the listing ID does not exist."
  - **Implementation:** Your test would make an API request to a non-existent ID and then assert that the HTTP response has a status code of 404 and that the JSON body matches your standard error format.

Appendix: Summary Cheat Sheet

Error Type / Concept	**What It Is**	**Common Fix / Implementation**
Default Handler	Express's built-in handler. Sends HTML with a stack	Replace it with a custom error-handling middleware.

	trace.	
Custom Error Middleware	A special middleware with 4 parameters: (err, req, res, next).	Place it at the very end of your app.js. Use it to send standardized JSON error responses.
ExpressError Class	A custom class extending Error to include a status code.	throw new ExpressError(404, "Not Found"). Allows you to control the HTTP status from your routes.
Async Errors	Errors in async functions that Express doesn't catch by default.	Wrap your async route handlers in a utility function (asyncWrap) that adds a .catch(next).
Mongoose ValidationError	Schema validation fails (e.g., required field missing).	In your error handler, check for err.name === 'ValidationError'. Format the messages into a user-friendly 400 response.
Mongoose CastError	Invalid data type for a schema path (e.g., bad ObjectId).	Check for err.name === 'CastError' and return a 400 response explaining which field was invalid.
Production Errors	Hiding sensitive details from users.	Use process.env.NODE_ENV to check if you are in production. If so, send a generic message and never send the stack trace.