SECTION 8 — Centralized Error Handling

Designing Structured and Scalable Error Management

1. Why Error Handling Is Important

Every backend application will encounter errors.

Examples:

- Invalid login credentials
- Missing required fields
- Database connection failure
- Unauthorized access
- Server crashes

If errors are not handled properly:

- The server may crash.
- The frontend may receive inconsistent responses.
- Debugging becomes difficult.
- Security risks increase.

Enterprise systems require:

- Consistent error responses
- Proper HTTP status codes
- Structured error format
- Centralized error control

2. The Problem With Traditional Error Handling

In beginner-level applications, developers often write:

```
try {
  // logic
} catch (error) {
  res.status(500).json({ message: "Something went wrong" });
}
```

Problems with this approach:

- Repeated try/catch blocks everywhere
- Inconsistent error messages
- No structured error codes
- Difficult maintenance
- Poor scalability

This approach does not scale in large applications.

3. What Is Centralized Error Handling?

Centralized error handling means:

All errors are handled in one dedicated middleware.

Instead of handling errors inside every controller, we:

1. Throw errors.
2. Let middleware catch them.
3. Send structured response from one place.

This ensures consistency.

4. Components of Centralized Error Handling

In Monsta, centralized error handling includes:

- AppError utility
- asyncHandler utility
- error.middleware.js

These work together.

5. Step 1 — Throw Structured Errors

Instead of:

throw new Error("Invalid credentials");

We use:

throw new AppError("Invalid credentials", 401, "INVALID_LOGIN");

This allows:

- Custom message
- HTTP status code
- Optional error code

This makes errors predictable.

6. Step 2 — Async Handler For Catching Errors

Async functions normally require try/catch.

Instead, we wrap controllers using:

```
asyncHandler(async (req, res) => {
  ...
});
```

If an error occurs, asyncHandler automatically forwards it to:

next(error)

No need for manual try/catch in every controller.

7. Step 3 — Global Error Middleware

This is the final error handler.

It must be placed last in server.js:

app.use(errorMiddleware);

Example: error.middleware.js

```
const errorMiddleware = (err, req, res, next) => {
  console.error("Error:", err);

  const statusCode = err.statusCode || 500;

  res.status(statusCode).json({
    success: false,
    message: err.message || "Internal Server Error",
    errorCode: err.errorCode || "SERVER_ERROR"
  });
};

export default errorMiddleware;
```

8. Error Handling Flow

When an error occurs:

```
Controller
   ↓
Throws AppError
   ↓
asyncHandler catches it
   ↓
Forwards to next(error)
   ↓
errorMiddleware handles it
   ↓
Structured response sent to client
```

This flow ensures:

- Clean controllers
- Standardized error format
- Single error control point

9. Handling Mongoose Errors

Database errors should also be handled properly.

Example cases:

- ValidationError
- Duplicate key error
- Invalid ObjectId

Inside error middleware, we can detect:

```
if (err.name === "ValidationError") { ... }
if (err.code === 11000) { ... }
if (err.name === "CastError") { ... }
```

This ensures database errors are also standardized.

10. Example Error Response Format

Example response:

```
{
  "success": false,
  "message": "Invalid credentials",
  "errorCode": "INVALID_LOGIN"
}
```

This is predictable and frontend-friendly.

Enterprise systems require consistent response contracts.

11. Why Centralized Error Handling Is Enterprise-Level

It provides:

- Consistency
- Clean architecture
- Separation of concerns
- Better logging
- Easy debugging
- Better frontend integration
- Future extensibility

For example:

- Adding logging service later
- Sending errors to monitoring tools
- Categorizing error severity

All possible from one place.

12. Common Mistakes Students Should Avoid

1. Returning raw error objects.
2. Sending stack traces to frontend.
3. Writing try/catch in every controller.
4. Not setting proper HTTP status codes.
5. Placing error middleware before routes.

13. Key Architectural Principle

Controllers should:

- Focus on request and response.
- Not manage error formatting.

Services should:

- Throw errors when needed.

Error middleware should:

- Decide how error is presented to client.

This separation keeps system clean.

14. Summary

By the end of this section, students should understand:

- Why centralized error handling is important
- How AppError works
- How asyncHandler removes repetitive try/catch
- How error middleware standardizes responses
- How database errors are managed
- Why this approach is scalable

Centralized error handling is a foundational concept in enterprise backend systems.