# express # guide

# Error Handling & Validation

**TCSS 460 — Client/Server Programming**

Your API will receive bad input. Clients will send missing fields, invalid IDs, malformed JSON, and requests for resources that do not exist. How your API responds to these situations is just as important as how it handles happy-path requests. This guide covers input validation, consistent error response formats, HTTP status codes for errors, and centralized error handling — the patterns that make your API trustworthy and debuggable.

---

## 1 Why Error Handling Matters

Three things will go wrong with every API you build:

1. **Bad input** — clients send missing fields, wrong types, or nonsense values
2. **Missing resources** — clients request things that do not exist
3. **Internal failures** — your code throws, a database query fails, an external service is down

If you do not handle these cases, two bad things happen:

- **Unhandled errors crash the server.** A single thrown exception in a route handler can take down your entire API, disconnecting every client.
- **Poor error responses waste hours.** If your API returns a raw `500` with no body, the client developer (who might be your teammate in the group project) has no idea what went wrong or how to fix their request.

> **Important**
>
> Good error responses are not a luxury — they are a requirement. Every error should return a JSON body with a clear message explaining what happened and what the client should do differently.

---

# 2 Express Error Handling Basics

## 2.1 Synchronous Errors

If a synchronous route handler throws an error, Express catches it automatically:

```
app.get('/fail', (request, response) => {
    throw new Error('Something broke'); // Express catches this
});
```

Express forwards the error to any registered error-handling middleware (covered in Section 8).

## 2.2 Asynchronous Errors in Express 5

Express 5 (which this course uses) also catches errors thrown inside `async` route handlers:

```
app.get('/data', async (request, response) => {
    const result = await someAsyncOperation(); // If this rejects, Express
catches it
    response.json(result);
});
```

This is a significant improvement over Express 4, where unhandled promise rejections could silently crash the process.

## 2.3 Explicit Try/Catch

When you want to send a specific error response instead of relying on a global handler, use try/catch. The lecture demo's proxy controller does this:

```
export const getWeather = async (request: Request, response: Response) => {
    const city = request.query.city || request.params.city;
    const apiKey = process.env.WEATHER_API_KEY;

    try {
        const fetchResponse = await fetch(
            `${BASE_URL}/weather?
q=${encodeURIComponent(String(city))}&appid=${apiKey}&units=metric`
        );
        const data = (await fetchResponse.json()) as Record<string, unknown>;

        if (!fetchResponse.ok) {
            response.status(fetchResponse.status).json({ error: data.message
|| 'Weather API error' });
```

```
            return;
        }

        response.json(data);
    } catch (_error) {
        response.status(502).json({ error: 'Failed to reach weather service'
});
    }
};
```

This handler does three things right:

1. **Checks `fetchResponse.ok`** — if the upstream API returns an error (like 404 for a bad city name), the handler forwards that status code
2. **Catches network failures** — if `fetch` itself throws (server unreachable, DNS failure), it returns `502 Bad Gateway`
3. **Never leaks internal details** — the client gets a clean message, not a stack trace

---

## 3 Consistent Error Response Format

Pick a format for error responses and stick with it. Every error from your API should look the same so clients can parse them predictably.

### 3.1 The Simple Format

The lecture demo uses a straightforward format:

```
{
    "error": "Missing required query parameter: q"
}
```

For validation errors with multiple issues:

```
{
    "error": "Validation failed",
    "details": [
        "name is required and must be a string",
        "email is required and must be a string"
    ]
}
```

### 3.2 Rules for Error Responses

1. **Always return JSON** — never return a raw status code with no body
2. **Always include a message** — tell the client what went wrong
3. **Use the same field names** — `error` for the message, `details` for specifics
4. **Don't leak internals** — no stack traces, no database error messages, no file paths

> ⚠ **Don't Leak Internal Details**
>
> This is a security issue, not just a style preference. A stack trace reveals your file paths, dependency versions, and database structure. An attacker can use this information. In production, log the full error on the server and send a clean message to the client.

```
// BAD — leaks internals
response.status(500).json({ error: err.stack });

// GOOD — clean message to client, full error in server logs
console.error(err);
response.status(500).json({ error: 'Internal server error' });
```

---

## 4 Input Validation

Never trust data from the client. Every value in `request.body`, `request.params`, and `request.query` could be missing, the wrong type, or malicious. Validation is the practice of checking input before using it.

### 4.1 What to Validate

| Source | What to Check | Example |
|---|---|---|
| `request.params` | Correct format (numeric ID, valid slug) | `id` is a positive integer |
| `request.query` | Required params present, valid values | `q` is not empty |
| `request.body` | Required fields exist, correct types, valid ranges | `name` is a string, `email` is a string |

### 4.2 Everything is a String (Until You Convert It)

Route parameters and query string values are **always strings**. Even if the URL contains `/users/42`, `request.params.id` is the string `"42"`, not the number `42`. Request body values have their JSON types (string, number, boolean), but you should still verify them because the client controls what JSON they send.

---

## 5 Validation as Middleware

The lecture demo puts validation logic in middleware functions. This keeps the route handlers clean — they only deal with the happy path.

### 5.1 Validating Query Parameters

The `validateSearchQuery` middleware checks that the `q` query parameter exists:

```
export const validateSearchQuery = (request: Request, response: Response,
next: NextFunction) => {
    if (!request.query.q) {
        response.status(400).json({ error: 'Missing required query parameter:
q' });
        return;
    }
    next();
};
```

Used on the route:

```
inputRouter.get('/search', validateSearchQuery, searchByQuery);
```

If `q` is missing, the middleware sends `400` and the handler never runs. If `q` is present, `next()` passes control to `searchByQuery`.

### 5.2 Validating Route Parameters

The `validateNumericId` middleware checks that `:id` is a positive integer:

```
export const validateNumericId = (request: Request, response: Response, next:
NextFunction) => {
    const id = Number(request.params.id);
    if (!Number.isInteger(id) || id <= 0) {
        response.status(400).json({ error: 'Parameter "id" must be a positive
```

```
integer' });
        return;
    }
    next();
};
```

This catches several bad inputs:

| Input | `Number(input)` | Valid? |
|---|---|---|
| `"42"` | `42` | Yes |
| `"0"` | `0` | No (not positive) |
| `"-5"` | `-5` | No (not positive) |
| `"abc"` | `NaN` | No (not an integer) |
| `"3.14"` | `3.14` | No (not an integer) |

## 5.3 Validating Request Bodies

The `validateUserBody` middleware checks for required fields and accumulates all errors before responding:

```
export const validateUserBody = (request: Request, response: Response, next:
NextFunction) => {
    const { name, email } = request.body;
    const errors: string[] = [];

    if (!name || typeof name !== 'string') {
        errors.push('name is required and must be a string');
    }
    if (!email || typeof email !== 'string') {
        errors.push('email is required and must be a string');
    }

    if (errors.length > 0) {
        response.status(400).json({ error: 'Validation failed', details:
errors });
        return;
    }
    next();
};
```

### Accumulate Errors

Don't return on the first error. Collect all validation failures and return them together. This lets the client fix everything in one shot instead of playing whack-a-mole — fix one error, submit, find the next error, fix, submit, repeat.

### Try It Yourself

1. Start the lecture demo: `npm run dev`
2. Send `POST /v2/input/users` with an empty JSON body `{}`
3. You should get a `400` response with **both** errors: name and email are missing
4. Now send `POST /v2/input/users` with `{"name": "Alice"}` (missing email)
5. You should get a `400` with only the email error
6. Finally, send a valid body `{"name": "Alice", "email": "alice@example.com"}` — you should get `201`
7. Try `GET /v2/input/users/abc` — the numeric ID validation returns `400`
8. Try `GET /v2/input/users/42` — valid ID, `200` response

## 5.4 Validating Environment Configuration

Not all validation is about client input. The lecture demo also validates that required server configuration is present:

```
export const requireEnvVar = (key: string) => {
    return (_request: Request, response: Response, next: NextFunction) => {
        if (!process.env[key]) {
            response.status(500).json({ error: `${key} is not configured` });
            return;
        }
        next();
    };
};
```

This is a **middleware factory** — a function that returns a middleware function. It takes the environment variable name as an argument, so you can reuse it:

```
proxyRouter.use(requireEnvVar('WEATHER_API_KEY'));
```

If `WEATHER_API_KEY` is not set in the `.env` file, every route on the proxy router returns `500` with a clear message. This catches configuration mistakes immediately instead of letting them cause cryptic errors deep in a controller.

## 6 Validation Approaches

The lecture demo uses **manual validation** — explicit `if` checks in middleware functions. This is the approach you should learn first.

### 6.1 Manual Validation (What This Course Uses)

Manual checks are explicit, easy to read, and have zero dependencies:

```
if (!request.body.name || typeof request.body.name !== 'string') {
    errors.push('name is required and must be a string');
}
```

You can see exactly what is being checked and what error message the client receives. There is no magic.

### 6.2 Validation Libraries (For Awareness)

In production codebases, teams often use validation libraries like **Zod** or **Joi** to define schemas declaratively:

```
// Example with Zod (not required for this course)
import { z } from 'zod';

const userSchema = z.object({
    name: z.string().min(1),
    email: z.string().email(),
});
```

These libraries are powerful but add a dependency and a learning curve. For this course, manual validation is sufficient and teaches you what these libraries do under the hood.

**Gen AI & Learning: Validation and AI-Generated Code**

AI coding assistants are very good at generating validation code. If you describe your data shape and constraints, an agent can produce Zod schemas or manual checks quickly. But you need to understand what "validate that `id` is a positive integer" means before you can review what the agent generates. Learn manual validation first — it makes you a better judge of generated code.

# 7 HTTP Status Codes for Errors

HTTP status codes communicate what went wrong at the protocol level. Clients (including browsers, Postman, and front-end apps) use these codes to decide how to handle the response.

## 7.1 Client Errors (4xx)

These mean **the client made a mistake** — the request was wrong in some way:

| Code | Name | When to Use | Example |
|---|---|---|---|
| `400` | Bad Request | Malformed input, missing required fields, invalid types | `validateUserBody` returns this |
| `401` | Unauthorized | No credentials provided, or credentials are invalid | Missing or expired JWT token |
| `403` | Forbidden | Valid credentials, but insufficient permissions | Regular user accessing admin route |
| `404` | Not Found | The requested resource does not exist | `GET /users/99999` when user 99999 doesn't exist |
| `409` | Conflict | Action conflicts with current state | Creating a user with a duplicate email |

## 7.2 Server Errors (5xx)

These mean **the server failed** — the request may have been valid, but something went wrong on your end:

| Code | Name | When to Use | Example |
|---|---|---|---|
| `500` | Internal Server Error | Unexpected failure, unhandled exception | Missing env var, uncaught bug |

| Code | Name | When to Use | Example |
|---|---|---|---|
| `502` | Bad Gateway | Upstream service returned an invalid response | `getWeather` when OpenWeatherMap is down |

## 7.3 Choosing the Right Code

A common mistake is returning `500` for everything. Use this decision tree:

```
Is the request itself wrong?
├── Yes → 4xx (client error)
│   ├── Missing/invalid input? → 400
│   ├── No authentication? → 401
│   ├── Wrong permissions? → 403
│   ├── Resource doesn't exist? → 404
│   └── Duplicate/conflict? → 409
└── No → 5xx (server error)
    ├── Your code broke? → 500
    └── External service failed? → 502
```

**Important**

The distinction between `401` and `403` confuses many developers. **401** means "I don't know who you are" (missing or invalid credentials). **403** means "I know who you are, but you're not allowed to do this" (valid credentials, insufficient permissions).

**Gen AI & Learning: AI and Status Codes**

AI coding agents frequently choose the wrong status code. A common pattern: the agent returns `500` for everything that goes wrong, even when the client sent bad input (which should be `400`). Another: using `401` when it should be `403`, or `200` for a resource creation that should return `201`. When reviewing AI-generated route handlers, check the status codes carefully — the logic may be correct while the HTTP semantics are wrong. If your API documentation specifies the expected status codes (see the OpenAPI Documentation guide), you can catch these mismatches quickly.

---

# 8 Centralized Error Handler

Instead of handling every possible error in every route handler, you can create a single error-handling middleware that catches everything.

## 8.1 The Pattern

```
import { Request, Response, NextFunction } from 'express';

const errorHandler = (err: Error, _request: Request, response: Response,
_next: NextFunction) => {
    // Log the full error for debugging (server-side only)
    console.error(err);

    // Send a clean response to the client
    response.status(500).json({ error: 'Internal server error' });
};
```

Register it **after** all routes:

```
app.use(routes);
app.use(errorHandler);   // Must be last
```

## 8.2 Development vs. Production

In development, you want as much detail as possible. In production, you want to hide internals:

```
const errorHandler = (err: Error, _request: Request, response: Response,
_next: NextFunction) => {
    console.error(err);

    if (process.env.NODE_ENV === 'development') {
        response.status(500).json({
            error: err.message,
            stack: err.stack,
        });
    } else {
        response.status(500).json({ error: 'Internal server error' });
    }
};
```

**Try It Yourself**

Set `NODE_ENV=development` in your `.env` file, start the server, and intentionally throw an error in a route handler (`throw new Error('test')`). You should see the stack trace in the response. Now change to `NODE_ENV=production`, restart, and trigger the same error — you should get a clean "Internal server error" message with no stack trace.

### 8.3 When to Use Each Approach

| Approach | When to Use |
| --- | --- |
| **Validation middleware** | Checking input before the handler runs (Sections 5-6) |
| **Try/catch in handlers** | Handling specific failures with specific responses (Section 2.3) |
| **Centralized error handler** | Catching anything that falls through — the safety net |

These are not mutually exclusive. A well-built API uses all three:

1. Validation middleware catches bad input early (returns `400`)
2. Try/catch in handlers deals with expected failures (returns specific codes)
3. The centralized handler catches everything else (returns `500`)

---

## 9 Summary

| Concept | Key Point |
| --- | --- |
| Never trust input | Validate `request.body`, `request.params`, and `request.query` before using them |
| Validation middleware | Check input before the handler runs; return `400` if invalid |
| Accumulate errors | Collect all validation failures and return them together |
| Consistent format | Every error response should be JSON with an `error` field |
| Don't leak internals | Log full errors server-side, send clean messages to clients |
| 4xx vs. 5xx | 4xx = client's fault, 5xx = server's fault |
| `401` vs. `403` | 401 = "who are you?", 403 = "you can't do that" |

| Concept | Key Point |
| --- | --- |
| Try/catch | Use for expected async failures with specific error responses |
| Centralized handler | `(err, request, response, next)` — the safety net for unhandled errors |
| Express 5 | Async errors are caught automatically — no wrapper needed |

## 10 References

**Official Documentation:**

- Express.js — Error Handling — Error-handling middleware and async patterns
- Express.js — API Reference: `express.json()` — Body parsing and error behavior
- MDN — HTTP Status Codes — Complete reference for all status codes
- MDN — 400 Bad Request — When to use 400
- MDN — 401 vs 403 — The distinction between unauthorized and forbidden

**TCSS 460 Lecture Demo**

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS-460-Backend-1 — Validation middleware in `src/middleware/validation.ts`

## 11 Further Reading

**External Resources**

- Zod Documentation — TypeScript-first schema validation library
- OWASP Input Validation Cheat Sheet — Security-focused validation guidance
- RFC 9110 — HTTP Semantics: Status Codes — The definitive reference for HTTP status codes

*This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.*