

WEB DEVELOPMENT COURSE

Express.js Intermediate/Advanced

8 Chapters

Philip Osztromok

Generated with Claude

EXPRESS.JS INTERMEDIATE/ADVANCED

Contents

Chapter 1 Authentication 2

Chapter 2 Database Integration 9

Chapter 3 File Uploads 17

Chapter 4 Security 23

Chapter 5 Testing Express APIs 30

Chapter 6 WebSockets 36

Chapter 7 API Docs 43

Chapter 8 Deployment 50

Authentication

> *Authentication is the gate between anonymous requests and your application's data. The central question is: how does the server know — on every request — who is making it? There are two fundamentally different answers: sessions (the server remembers you) and tokens (you carry proof of identity with you). JWT-based authentication is now the dominant approach for REST APIs. Getting it right means understanding what a JWT actually is, why the access/refresh split exists, and where each token should live on the client.*

Sessions vs Tokens

Session-Based (Stateful)

Server stores session state. Client holds only a random session ID in a cookie.

```
Client → POST /login → Server Server creates session, stores it (DB/Redis) Server → Set-Cookie: sessionId=abc123 Client sends cookie on every request Server looks up session in store on every request
```

Pro: Instant revocation (delete the session row).
Con: Needs a shared session store (Redis) for multiple server instances.

Token-Based (Stateless)

Server stores nothing. Client holds a signed token containing identity + expiry.

```
Client → POST /login → Server Server creates signed JWT, stores nothing Server → { accessToken, refreshToken } Client sends token: Authorization: Bearer <jwt> Server verifies signature — no DB lookup needed
```

Pro: Stateless — scales horizontally without a shared store.
Con: Can't revoke a token until it expires (need a denylist to force-logout).

What Is a JWT?

A JSON Web Token is three base64url-encoded JSON objects joined by dots: `header.payload.signature`. The header and payload are just base64 — anyone can decode them. The signature is a cryptographic MAC: it proves the token was issued by someone who knows the secret key, and that the header and payload have not been tampered with.

Header

Algorithm and token type.

`{ "alg": "HS256", "typ": "JWT" }`

Payload (Claims)

Identity + metadata. Readable by anyone.

`{ "sub": "42", "role": "admin", "iat": 1700000000, "exp": 1700000900 }`

Signature

Tamper-proof seal. Requires the secret key.

`HMAC-SHA256( base64(header) + "." + base64(payload), SECRET )`

```
// npm install jsonwebtoken
const jwt = require('jsonwebtoken');

// Sign a token — second arg is the secret, NEVER commit it
const token = jwt.sign(
    { sub: '42', role: 'admin' },   // payload (claims)
    process.env.JWT_SECRET,             // secret key
```

```
    { expiresIn: '15m', algorithm: 'HS256' }
);
// → "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI0MiIs..."

// Verify a token — throws if expired, tampered, or wrong secret
try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET, {
        algorithms: ['HS256'],   // always specify — prevents alg:none attack
    });
    console.log(decoded.sub);    // '42'
    console.log(decoded.role);   // 'admin'
    console.log(decoded.iat);    // issued-at timestamp (added automatically)
    console.log(decoded.exp);    // expiry timestamp (added automatically)
} catch (err) {
    // err.name === 'TokenExpiredError' | 'JsonWebTokenError' | 'NotBeforeError'
    console.error(err.name, err.message);
}
```

> **⚠ Signed ≠ Encrypted**
>
> A JWT payload is base64-encoded, not encrypted. Anyone who intercepts the token can read it by pasting it into jwt.io. Never put sensitive data in the payload — passwords, full credit card numbers, PII. The signature only proves authenticity (who signed it), not confidentiality (who can read it).

Access Tokens & Refresh Tokens

A single long-lived token creates a dilemma: if it's stolen, the attacker has it until it expires. If you make it expire in 5 minutes, users log out constantly. The access/refresh split resolves this.

Token	Lifetime	Stored where	Purpose
`Access token`	Short (5–15 min)	Memory (JS variable) or Authorization header	Authenticate API calls. Short lifetime limits damage if stolen.
`Refresh token`	Long (7–30 days)	httpOnly Secure cookie	Used *only* to get a new access token. Never sent to API routes.

The Token Flow

```
── Login ──────────────────────────────────────────────────────── Client ──POST /auth/login {
email, password }──────────────► Server verify credentials, generate access + refresh tokens Client ◄──{
accessToken } Set-Cookie: refreshToken=... (httpOnly)── Server ── Authenticated request
──────────────────────────────────────────── Client ──Authorization: Bearer
<accessToken>──────────────────► Server verifies signature in memory — no DB lookup Client ◄──200 { data
}──────────────────────────────────── Server ── Access token expires → silent refresh
──────────────────────────────── Client ──POST /auth/refresh (cookie sent automatically)──────► Server
verify refresh token, issue NEW access token + NEW refresh token invalidate old refresh token (rotation)
Client ◄──{ accessToken } Set-Cookie: refreshToken=... (new)── Server ── Logout
──────────────────────────────────────────────────────────── Client ──POST /auth/logout (cookie sent
```

```
automatically)──────► Server invalidate refresh token in DB, clear cookie Client ◄──204 No Content Set-
Cookie: refreshToken=; Max-Age=0─ Server
```

Implementation

Token Helper

```
// src/utils/tokens.js
const jwt    = require('jsonwebtoken');
const crypto = require('node:crypto');

const ACCESS_SECRET  = process.env.JWT_SECRET;
const REFRESH_SECRET = process.env.JWT_REFRESH_SECRET;
const ALG            = ['HS256'];

function signAccess(payload) {
    return jwt.sign(payload, ACCESS_SECRET, { expiresIn: '15m', algorithm: 'HS256' });
}

function signRefresh(payload) {
    return jwt.sign(payload, REFRESH_SECRET, { expiresIn: '7d', algorithm: 'HS256' });
}

function verifyAccess(token)  {
    return jwt.verify(token, ACCESS_SECRET,  { algorithms: ALG });
}

function verifyRefresh(token) {
    return jwt.verify(token, REFRESH_SECRET, { algorithms: ALG });
}

// Opaque token for refresh — random bytes, stored hashed in DB
function generateRefreshId() {
    return crypto.randomBytes(40).toString('hex');
}

module.exports = { signAccess, signRefresh, verifyAccess, verifyRefresh, generateRefreshId };
```

Auth Routes

```
// src/routes/v1/auth.js
require('express-async-errors');
const router      = require('express').Router();
const bcrypt      = require('bcrypt');
const { signAccess, signRefresh, verifyRefresh } = require('../../utils/tokens');
const createError = require('../../utils/createError');

// Refresh token denylist — in production: a DB table or Redis set
const revokedTokens = new Set();
```

```
const COOKIE_OPTS = {
    httpOnly: true,
    secure:   process.env.NODE_ENV === 'production',
    sameSite: 'strict',
    maxAge:   7 * 24 * 60 * 60 * 1000,  // 7 days in ms
    path:     '/auth',   // cookie only sent to /auth/* routes
};

// POST /auth/login
router.post('/login', async (req, res) => {
    const { email, password } = req.body ?? {};
    if (!email || !password) throw createError(400, 'email and password required');

    const user = await db.findByEmail(email);
    if (!user) throw createError(401, 'Invalid credentials');

    const ok = await bcrypt.compare(password, user.passwordHash);
    if (!ok) throw createError(401, 'Invalid credentials');

    const accessToken  = signAccess({ sub: user.id, role: user.role });
    const refreshToken = signRefresh({ sub: user.id });

    res.cookie('refreshToken', refreshToken, COOKIE_OPTS)
       .json({ accessToken });
});

// POST /auth/refresh — silent token rotation
router.post('/refresh', async (req, res) => {
    const token = req.cookies.refreshToken;
    if (!token) throw createError(401, 'No refresh token');
    if (revokedTokens.has(token)) throw createError(401, 'Refresh token revoked');

    const payload = verifyRefresh(token);  // throws if expired/tampered

    // Rotation: revoke old token, issue a new pair
    revokedTokens.add(token);
    const newAccess  = signAccess({ sub: payload.sub, role: payload.role });
    const newRefresh = signRefresh({ sub: payload.sub });

    res.cookie('refreshToken', newRefresh, COOKIE_OPTS)
       .json({ accessToken: newAccess });
});

// POST /auth/logout
router.post('/logout', (req, res) => {
    const token = req.cookies.refreshToken;
    if (token) revokedTokens.add(token);
    res.clearCookie('refreshToken', { path: '/auth' })
       .status(204).end();
});

module.exports = router;
```

Authentication Middleware

```
// src/middleware/authenticate.js
const { verifyAccess } = require('../utils/tokens');
const createError      = require('../utils/createError');

function authenticate(req, res, next) {
    const header = req.get('Authorization');
    if (!header?.startsWith('Bearer ')) {
        return next(createError(401, 'Bearer token required'));
    }
    const token = header.slice(7);
    try {
        req.user = verifyAccess(token);   // attaches { sub, role, iat, exp }
        next();
    } catch (err) {
        const message = err.name === 'TokenExpiredError'
            ? 'Token expired'
            : 'Invalid token';
        next(createError(401, message));
    }
}

// Optional: role-based guard that sits after authenticate()
const requireRole = (...roles) => (req, res, next) =>
    roles.includes(req.user?.role)
        ? next()
        : next(createError(403, 'Insufficient permissions'));

module.exports = { authenticate, requireRole };

// Usage in a router
const { authenticate, requireRole } = require('../middleware/authenticate');
router.get('/profile',       authenticate, controller.getProfile);
router.delete('/users/:id',  authenticate, requireRole('admin'), controller.deleteUser);
```

Token Storage Security

Storage	XSS risk	CSRF risk	Verdict
JS memory (variable)	Low	None	Best for access tokens in SPAs — lost on page refresh, which is fine at 15 min lifetime.
localStorage	High	None	Avoid for auth tokens — any injected script can read it.
sessionStorage	High	None	Same XSS risk as localStorage; cleared on tab close but still accessible to scripts.
httpOnly cookie	None	Yes (needs SameSite or CSRF token)	Best for refresh tokens — JavaScript cannot read it. Use `SameSite: 'strict'` to eliminate CSRF risk on same-origin requests.

🔒 Refresh Token Rotation

Every time a refresh token is used, immediately invalidate it and issue a new one. If a stolen refresh token is used by an attacker, the legitimate user's next refresh will find their token already revoked — this detects the theft. Without rotation, a stolen refresh token is valid for its full 30-day lifetime with no way to detect misuse.

Store refresh tokens in a DB table (`id, user_id, token_hash, expires_at, revoked`). Hash the token before storing — the DB never holds the raw value.

⚠ The alg:none Attack

JWTs carry the algorithm in the header. Early libraries trusted this header — so an attacker could set `"alg": "none"` and send an unsigned token that would verify successfully. Always pass `{ algorithms: ['HS256'] }` to `jwt.verify()` . Modern libraries default to rejecting `alg:none` , but being explicit costs nothing and is a habit worth building.

Quick Reference

Operation	API
`jwt.sign(payload, secret, options)`	Creates a signed token. Options: `expiresIn` , `algorithm` , `issuer` , `audience` .
`jwt.verify(token, secret, options)`	Validates and decodes. Throws `TokenExpiredError` , `JsonWebTokenError` , `NotBeforeError` .
`jwt.decode(token)`	Decodes without verifying. Use only for logging/debugging — never for authentication decisions.
`err.name === 'TokenExpiredError'`	Token signature was valid but `exp` is in the past. Client should refresh.
`err.name === 'JsonWebTokenError'`	Signature mismatch, malformed token, or wrong algorithm. Treat as 401.
`Authorization: Bearer <token>`	Standard header for sending access tokens. Parse with `req.get('Authorization').slice(7)` .

Coding Challenges

Challenge 1 — Login & Protected Route

Build an auth system with three routes: `POST /auth/login` (accept `{ email, password }` , validate against a hardcoded users map with pre-hashed passwords using bcrypt; return `{ accessToken }` and set a `refreshToken` httpOnly cookie); `GET /me` (protected — validate the Bearer access token, return `{ id, email, role }` from the decoded payload); `POST /auth/logout` (clear the refresh cookie, 204). Use two separate env-var secrets for access and refresh tokens. Write supertest tests: 400 for missing credentials, 401 for wrong password, successful login returns accessToken and sets Set-Cookie, `/me` without token returns 401, `/me` with valid token returns 200, `/me` with expired token returns 401 with "Token expired" message (create an already-expired token in the test by signing with `expiresIn: -1`).

View sample solution ↗

Challenge 2 — Refresh Token Rotation

Extend the auth system with a `POST /auth/refresh` route that reads the `refreshToken` cookie, verifies it with the refresh secret, revokes the old token (in-memory Set), issues a new access token and a new refresh token (rotation), and sets the new cookie. Add a `GET /protected` route that requires a valid access token and returns a confirmation. Write supertest tests using `supertest.agent()` to maintain cookies: full login → protected route works → simulate access token expiry → refresh returns a new access token → new access token works on protected route → old refresh token is now revoked (second refresh with old cookie returns 401) → logout clears cookie → protected route returns 401.

View sample solution ↗

Challenge 3 — Role-Based Access Control

Build a multi-role API with three roles: `user`, `editor`, `admin`. Write an `authenticate` middleware that extracts and verifies the Bearer token and attaches `req.user`. Write a `requireRole(...roles)` middleware factory that checks `req.user.role` — 401 if not authenticated (missing req.user), 403 if authenticated but wrong role. Routes: `GET /public` (no auth), `GET /dashboard` (any authenticated user), `POST /articles` (editor or admin only), `DELETE /articles/:id` (admin only). Generate test tokens directly (skip the login flow) by calling `jwt.sign()` in the test file. Write supertest tests: public route works unauthenticated, dashboard returns 401 with no token, dashboard returns 200 for user/editor/admin tokens, articles POST returns 403 for user-role token and 201 for editor/admin, articles DELETE returns 403 for user and editor, 204 for admin.

View sample solution ↗

Database Integration

> *Every real Express application eventually needs to read and write persistent data. This chapter covers integrating MySQL with Express using the `mysql2` package — the fastest, most actively maintained MySQL driver for Node.js. The three things that matter most: always use a **connection pool** (not a single connection), always use **parameterized queries** (never string-concat SQL), and always **release connections** back to the pool when you're done with them.*

mysql2 vs mysql

mysql (legacy)

Callback-based only. Requires wrapping in `util.promisify` for async/await. No built-in prepared statements — uses client-side escaping (still safe but different semantics). Actively receives only security fixes.

mysql2 (use this)

Native Promise support via `mysql2/promise`. Server-side prepared statements via `execute()` — the SQL is parsed once, parameters are sent separately, SQL injection is structurally impossible. Faster than `mysql`, drop-in compatible API.

```
// npm install mysql2

// Two import paths — you almost always want the promise one
const mysql = require('mysql2/promise');    // ✓ async/await
const mysql = require('mysql2');            // ✗ callback-based — avoid
```

Connection Pools

A single connection can handle only one query at a time. Under real load — multiple requests arriving simultaneously — they would queue behind each other. A pool maintains multiple open connections and assigns them to incoming queries on demand, then returns them when done. Creating a connection is expensive (TCP handshake + MySQL auth). A pool reuses them.

```
// src/db/pool.js — create once, export, import everywhere
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
    host:             process.env.DB_HOST     || 'localhost',
    port:             process.env.DB_PORT     || 3306,
    user:             process.env.DB_USER,
    password:         process.env.DB_PASS,
    database:         process.env.DB_NAME,
    waitForConnections: true,   // queue requests when pool is exhausted
    connectionLimit:    10,     // max simultaneous connections
    queueLimit:         0,      // 0 = unlimited queue (requests wait forever)
    timezone:           'Z',     // store/retrieve dates as UTC
```

```
});

module.exports = pool;
```

Option	Default	Notes
connectionLimit	10	Total open connections to MySQL. A good starting point — raise under sustained high traffic.
waitForConnections	true	If false, throws immediately when pool is full instead of queuing.
queueLimit	0	Max queued requests. 0 = unlimited. Set a number (e.g. 100) to shed load instead of building an infinite queue.
idleTimeout	60000	Idle connections are closed after this many ms. Keeps the pool healthy.
enableKeepAlive	false	Set true + keepAliveInitialDelay: 10000 to prevent MySQL's 8-hour timeout from dropping long-lived connections.

execute() vs query()

This is the most important distinction in the whole chapter. `pool.execute()` sends the SQL template and the parameters to MySQL in separate messages. MySQL parses the SQL first, then receives the parameters as raw binary data — they are never treated as SQL, making injection structurally impossible. `pool.query()` sends a complete string after client-side escaping. Both work, but `execute()` is safer and faster (MySQL caches the parsed statement).

✅ execute() — always use for user data

```
const [rows] = await pool.execute(
    'SELECT * FROM users WHERE email = ?',
    [email]     // ← separate, never becomes SQL
);
```

⚠ query() — safe only for static SQL

```
// Fine for schema queries with no user input
const [tables] = await pool.query(
    'SHOW TABLES'
);
// Do NOT do this:
// pool.query(`SELECT * FROM users WHERE email = '${emai
```

Basic CRUD Pattern

```
// src/models/product.js
const pool = require('../db/pool');

async function findAll() {
    const [rows] = await pool.execute('SELECT * FROM products ORDER BY id');
    return rows;
}

async function findById(id) {
    const [rows] = await pool.execute(
        'SELECT * FROM products WHERE id = ?',
        [id]
    );
```

```
    return rows[0] ?? null;   // undefined → null
}

async function create({ name, price, stock = 0 }) {
    const [result] = await pool.execute(
        'INSERT INTO products (name, price, stock) VALUES (?, ?, ?)',
        [name, price, stock]
    );
    return { id: result.insertId, name, price, stock };
}

async function update(id, fields) {
    // Build SET clause dynamically from only the supplied fields
    const allowed  = ['name', 'price', 'stock'];
    const entries  = Object.entries(fields).filter(([k]) => allowed.includes(k));
    if (entries.length === 0) return null;
    const setClauses = entries.map(([k]) => `${k} = ?`).join(', ');
    const values     = [...entries.map(([, v]) => v), id];
    const [result]   = await pool.execute(
        `UPDATE products SET ${setClauses} WHERE id = ?`,
        values
    );
    return result.affectedRows > 0 ? findById(id) : null;
}

async function remove(id) {
    const [result] = await pool.execute(
        'DELETE FROM products WHERE id = ?',
        [id]
    );
    return result.affectedRows > 0;
}

module.exports = { findAll, findById, create, update, remove };
```

What execute() returns

Always an array of two elements: `[rows, fields]` for SELECT, `[ResultSetHeader, fields]` for INSERT/UPDATE/DELETE. The `ResultSetHeader` has `insertId` (last auto-increment ID), `affectedRows`, and `changedRows`. You almost always destructure just the first: `const [rows] = await pool.execute(...)`.

Mapping MySQL Errors to HTTP Status Codes

MySQL throws errors with a `code` string property. Your service or error handler should catch these and map them to meaningful HTTP responses before they bubble up as 500s.

MySQL code	HTTP status	When it happens
ER_DUP_ENTRY	409 Conflict	INSERT or UPDATE violates a UNIQUE constraint (duplicate email, SKU, etc.)
ER_NO_REFERENCED_ROW_2	422 Unprocessable	INSERT/UPDATE references a foreign key that doesn't exist (e.g. invalid category_id)
ER_ROW_IS_REFERENCED_2	409 Conflict	DELETE would orphan child rows that reference this record

MySQL code	HTTP status	When it happens
ER_DATA_TOO_LONG	422 Unprocessable	Value exceeds column length — usually a bug in validation, but can come from user input
ER_BAD_NULL_ERROR	422 Unprocessable	NULL inserted into a NOT NULL column without a default
ECONNREFUSED	503 Service Unavailable	MySQL server is down or unreachable — always log this, never expose to clients

```
// src/utils/dbErrors.js — centralised MySQL → HTTP mapping
function handleDbError(err) {
    switch (err.code) {
        case 'ER_DUP_ENTRY': {
            // MySQL includes the duplicate value in err.message — extract the key name
            const match = err.message.match(/for key '(.+?)'/);
            const key   = match ? match[1] : 'unknown';
            const e     = new Error(`Duplicate value for key: ${key}`);
            e.status    = 409;
            return e;
        }
        case 'ER_NO_REFERENCED_ROW_2': {
            const e  = new Error('Referenced record does not exist');
            e.status = 422;
            return e;
        }
        case 'ER_ROW_IS_REFERENCED_2': {
            const e  = new Error('Cannot delete: other records depend on this one');
            e.status = 409;
            return e;
        }
        default:
            return err;   // unknown DB error → will become 500
    }
}

module.exports = { handleDbError };

// Usage in a service
async function create(data) {
    try {
        return await ProductModel.create(data);
    } catch (err) {
        throw handleDbError(err);
    }
}
```

Transactions

A transaction groups multiple SQL statements so they either all succeed together or all roll back together. You need a transaction any time you are making changes that must be atomic — a bank transfer, decrementing stock while creating an order, or inserting related rows into two tables simultaneously.

```
── Transaction lifecycle ──────────────────────────────────── 1. pool.getConnection() ← borrow one specific connection 2. conn.beginTransaction() ← START TRANSACTION 3. conn.execute(...) ← first statement 4. conn.execute(...) ← second statement 5a. conn.commit() ← success path 5b. conn.rollback() ← error path (undo everything) 6. conn.release() ← ALWAYS — returns connection to pool (goes in finally{} so it runs even if commit/rollback throws)
```

```
// Transfer balance between two accounts — must be atomic
async function transfer(fromId, toId, amount) {
    const conn = await pool.getConnection();
    try {
        await conn.beginTransaction();

        // Debit — check balance first (use FOR UPDATE to lock the row)
        const [[from]] = await conn.execute(
            'SELECT balance FROM accounts WHERE id = ? FOR UPDATE',
            [fromId]
        );
        if (!from) throw createError(404, 'Source account not found');
        if (from.balance < amount) throw createError(422, 'Insufficient funds');

        await conn.execute(
            'UPDATE accounts SET balance = balance - ? WHERE id = ?',
            [amount, fromId]
        );

        // Credit
        const [creditResult] = await conn.execute(
            'UPDATE accounts SET balance = balance + ? WHERE id = ?',
            [amount, toId]
        );
        if (creditResult.affectedRows === 0) throw createError(404, 'Target account not found');

        await conn.commit();
        return { success: true, amount };

    } catch (err) {
        await conn.rollback();
        throw err;
    } finally {
        conn.release();   // ← MUST run even when catch re-throws
    }
}
```

⚠ Always release() in a finally block

If you forget `conn.release()`, the connection is permanently checked out from the pool. After `connectionLimit` requests that hit this path, the pool is exhausted and every subsequent request hangs forever (or until `queueLimit` is exceeded). The bug is silent until the pool runs out — and it will run out. Put `release()` in `finally {}` and it cannot be skipped.

Query Helpers

Rather than writing the same execute-and-destructure boilerplate on every model, you can wrap common patterns in helper functions once. These are thin utilities — not an ORM — so you keep full control over your SQL.

```
// src/db/helpers.js
const pool = require('./pool');

// Run a query, return just the rows array
async function query(sql, params = []) {
    const [rows] = await pool.execute(sql, params);
    return rows;
}

// SELECT that returns one row or null
async function queryOne(sql, params = []) {
    const rows = await query(sql, params);
    return rows[0] ?? null;
}

// INSERT/UPDATE/DELETE — returns the ResultSetHeader
async function run(sql, params = []) {
    const [result] = await pool.execute(sql, params);
    return result;  // { insertId, affectedRows, changedRows, ... }
}

module.exports = { query, queryOne, run };

// Usage — much less boilerplate per model
const { query, queryOne, run } = require('../db/helpers');

const getAll   = ()     => query('SELECT * FROM products ORDER BY id');
const getById  = (id)   => queryOne('SELECT * FROM products WHERE id = ?', [id]);
const insert   = (data) => run('INSERT INTO products (name, price) VALUES (?, ?)', [data.name, d
const destroy  = (id)   => run('DELETE FROM products WHERE id = ?', [id]);
```

Wiring into MVC

```
// src/services/productsService.js — business logic, no HTTP
const ProductModel          = require('../models/product');
const { handleDbError }     = require('../utils/dbErrors');
const createError           = require('../utils/createError');

async function list()       { return ProductModel.findAll(); }

async function get(id) {
    const p = await ProductModel.findById(id);
    if (!p) throw createError(404, `Product ${id} not found`);
    return p;
```

```
}

async function create(data) {
    if (!data?.name)  throw createError(422, 'name is required');
    if (data.price == null || data.price < 0) throw createError(422, 'price must be >= 0');
    try {
        return await ProductModel.create(data);
    } catch (err) {
        throw handleDbError(err);
    }
}

module.exports = { list, get, create };

// src/controllers/productsController.js — thin, HTTP-only
const svc = require('../services/productsService');

module.exports = {
    list:   async (req, res) => res.json(await svc.list()),
    show:   async (req, res) => res.json(await svc.get(req.params.id)),
    create: async (req, res) => {
        const p = await svc.create(req.body ?? {});
        res.status(201).set('Location', `/api/v1/products/${p.id}`).json(p);
    },
};
```

Quick Reference

Operation	API
`pool.execute(sql, params)`	Prepared statement — always use for user data. Returns `[rows, fields]`.
`pool.query(sql)`	Plain string query — only for static SQL with no user input.
`result.insertId`	The auto-increment ID of the row just inserted.
`result.affectedRows`	Rows affected by UPDATE or DELETE. 0 means the WHERE matched nothing.
`pool.getConnection()`	Borrow a specific connection from the pool (needed for transactions).
`conn.beginTransaction()`	Start a transaction on the borrowed connection.
`conn.commit()`	Commit all changes made since beginTransaction().
`conn.rollback()`	Undo all changes since beginTransaction().
`conn.release()`	Return the connection to the pool. Goes in finally{} — never skip it.
`err.code`	MySQL error code string (e.g. `'ER_DUP_ENTRY'`). Check this in catch blocks to map to HTTP status codes.

Coding Challenges

Challenge 1 — Products CRUD with Real MySQL

Create a `products` table (`id INT AUTO_INCREMENT PRIMARY KEY, name VARCHAR(255) NOT NULL UNIQUE, price DECIMAL(10,2) NOT NULL, stock INT DEFAULT 0`). Write a `ProductModel` module using `mysql2/promise` that implements `findAll()` , `findById(id)` , `create({name, price, stock})` , `update(id, fields)` (only update supplied fields using a dynamic SET clause — filter against an allowlist of `['name','price','stock']`), and `remove(id)` . Write integration tests using a dedicated test database (`express_test`). In `beforeAll` : create the table. In `afterAll` : drop the table and call `pool.end()` to close all connections (required for Jest to exit cleanly). In `beforeEach` : truncate the table and reset the auto-increment counter. Tests: `findAll` returns empty array initially; `create` returns the new record with `insertId` ; `findById` returns null for a missing id; `update` only changes supplied fields; `remove` returns true on success and false for a missing id.

View sample solution ↗

Challenge 2 — Query Helpers & Error Mapping

Write a `src/db/helpers.js` module exporting `query(sql, params)` , `queryOne(sql, params)` , and `run(sql, params)` as described in the chapter. Write a `src/utils/dbErrors.js` module with a `handleDbError(err)` function that maps `ER_DUP_ENTRY` → 409, `ER_NO_REFERENCED_ROW_2` → 422, `ER_ROW_IS_REFERENCED_2` → 409, and leaves other errors unchanged. Rewrite the `ProductModel` from Challenge 1 to use the helpers (each model function becomes a one-liner). Add a `categories` table with a foreign key from `products.category_id` to `categories.id` . Write integration tests verifying: creating a product with a duplicate name returns 409 with a message containing "Duplicate"; creating a product with a non-existent `category_id` returns 422; deleting a category that has products returns 409; unknown MySQL errors pass through unchanged.

View sample solution ↗

Challenge 3 — Transactions: Stock Reservation

Add an `orders` table and an `order_items` table. Write a `createOrder(items)` function where `items` is an array of `{productId, quantity}` . The function must run as a single transaction: for each item, SELECT the product with `FOR UPDATE` (to lock the row against concurrent modifications), check that `stock >= quantity` (throw 422 with a clear message if not), then UPDATE `products SET stock = stock - quantity` . After all items are processed, INSERT a row into `orders` and one row per item into `order_items` . Commit on success, rollback on any error, and always release the connection. Write integration tests: a successful order reduces product stock; an order with insufficient stock for any item rolls back all stock changes (verify no stock was decremented); an order referencing a non-existent product throws 404 and rolls back; ordering from two items where the second is out of stock leaves the first item's stock unchanged.

View sample solution ↗

File Uploads

> *File uploads use `Content-Type: multipart/form-data` — a completely different encoding from JSON. `express.json()` ignores it, so you need dedicated middleware. `multer` is the standard choice: it parses multipart requests, runs your validation callbacks, and either buffers the file in memory or writes it to disk before your route handler runs. The three things to nail: never trust the client's filename, always validate MIME type server-side, and always handle `MulterError` separately from application errors.*

The Two Storage Engines

memoryStorage()

File is buffered entirely in RAM as a `Buffer` on `req.file.buffer`.

✓ Good for: transient processing — resize, validate, then stream to cloud storage (S3, GCS).

✗ Bad for: large files or high concurrency — RAM fills up fast. No disk I/O fallback.

diskStorage()

File is streamed to disk as it arrives. `req.file.path` is the saved path.

✓ Good for: files that stay on this server long-term; large files that would exhaust RAM.

✗ Bad for: multi-server deployments — each server has its own disk, no shared access.

Memory Storage

```
// npm install multer
const multer = require('multer');

const upload = multer({ storage: multer.memoryStorage() });

// Single file: field name must match the HTML input name / form-data key
app.post('/upload', upload.single('avatar'), (req, res) => {
    // req.file is undefined if no file was sent
    if (!req.file) return res.status(400).json({ error: 'No file uploaded' });

    console.log(req.file.originalname);  // client-supplied filename — do NOT use for saving
    console.log(req.file.mimetype);      // 'image/jpeg', 'application/pdf', etc.
    console.log(req.file.size);          // bytes
    console.log(req.file.buffer);        // Buffer — the raw file bytes

    res.json({ size: req.file.size });
});
```

Disk Storage

```
const path   = require('node:path');
const crypto = require('node:crypto');
```

```
const multer = require('multer');

const storage = multer.diskStorage({
    destination(req, file, cb) {
        cb(null, './uploads');   // directory must exist — multer won't create it
    },
    filename(req, file, cb) {
        // NEVER use file.originalname — it's client-controlled
        // Generate a random name, keep only the extension
        const ext      = path.extname(file.originalname).toLowerCase();
        const safeName = crypto.randomBytes(16).toString('hex') + ext;
        cb(null, safeName);   // e.g. 'a3f8c2d1...b9e4.jpg'
    },
});

const upload = multer({ storage });

app.post('/upload', upload.single('photo'), (req, res) => {
    console.log(req.file.filename);     // the name we generated (safe)
    console.log(req.file.path);          // './uploads/a3f8c2d1...b9e4.jpg'
    console.log(req.file.destination);   // './uploads'
    res.json({ filename: req.file.filename });
});
```

Handling Single vs Multiple Files

Method	req.file / req.files	Use when
`upload.single('name')`	`req.file` — one object or undefined	One file from one field (avatar, document)
`upload.array('name', max)`	`req.files` — array, same field	Multiple files from one field (gallery, attachments)
`upload.fields([{name, maxCount}])`	`req.files` — object keyed by field name	Different files from different fields (avatar + gallery)
`upload.none()`	Neither — text fields only	Multipart form with no file fields

```
// Multiple files — same field
app.post('/gallery', upload.array('images', 5), (req, res) => {
    // req.files → [{ fieldname, originalname, size, ... }, ...]
    res.json({ count: req.files.length });
});

// Different files from different fields
app.post('/profile', upload.fields([
    { name: 'avatar',  maxCount: 1 },
    { name: 'gallery', maxCount: 4 },
]), (req, res) => {
    const avatar  = req.files.avatar?.[0];    // single file object
    const gallery = req.files.gallery ?? [];  // array (may be absent)
    res.json({ avatar: avatar?.filename, galleryCount: gallery.length });
```

```
});
```

Validation: fileFilter and limits

```
const ALLOWED_MIME = new Set(['image/jpeg', 'image/png', 'image/webp']);
const ALLOWED_EXT  = new Set(['.jpg', '.jpeg', '.png', '.webp']);

const upload = multer({
    storage: multer.diskStorage({ /* ... */ }),

    limits: {
        fileSize:  5 * 1024 * 1024,  // 5 MB — enforced during streaming, before route runs
        files:     5,                  // max number of files per request
        fieldSize: 1 * 1024 * 1024,  // max size of non-file fields (1 MB)
    },

    fileFilter(req, file, cb) {
        // cb(error)         → reject with an error (route sees it as next(err))
        // cb(null, true)    → accept the file
        // cb(null, false)   → silently skip (req.file will be undefined)

        const ext = path.extname(file.originalname).toLowerCase();

        if (!ALLOWED_MIME.has(file.mimetype) || !ALLOWED_EXT.has(ext)) {
            // Use a custom error so the error handler can map it to 415
            const err    = new Error('Only JPEG, PNG and WebP images are allowed');
            err.status   = 415;
            return cb(err);
        }
        cb(null, true);
    },
});
```

> **MIME type vs extension — check both**
>
> MIME type (`file.mimetype`) comes from the client's `Content-Type` header for that part — trivial to spoof. File extension (`path.extname(file.originalname)`) is also client-supplied — also trivial to spoof. Neither alone is sufficient. Checking both raises the bar: an attacker must get two things right instead of one. For critical security (e.g. preventing SVG XSS), also read the first few bytes of the file and check the magic bytes signature server-side.

Handling MulterError

When multer hits a limit or an unexpected field, it throws a `MulterError` (a subclass of `Error`) and calls `next(err)`. These errors are *not* application errors — they need their own handling branch in the error middleware, otherwise they all become 500s.

```
const multer = require('multer');

// eslint-disable-next-line no-unused-vars
app.use((err, req, res, next) => {
    if (err instanceof multer.MulterError) {
        const statusMap = {
            LIMIT_FILE_SIZE:       413,   // Payload Too Large
            LIMIT_FILE_COUNT:      400,   // too many files
            LIMIT_UNEXPECTED_FILE:400,   // wrong field name
            LIMIT_FIELD_COUNT:     400,
            LIMIT_FIELD_KEY_SIZE: 400,
            LIMIT_FIELD_VALUE_SIZE:413,
        };
        const status  = statusMap[err.code] ?? 400;
        const message = err.code === 'LIMIT_FILE_SIZE'
            ? `File too large (max 5 MB)`
            : err.message;
        return res.status(status).json({ error: message, code: err.code });
    }

    // fileFilter errors (status set by us) and other app errors
    const status  = err.status ?? 500;
    const message = status < 500 ? err.message : 'Internal server error';
    res.status(status).json({ error: message });
});
```

MulterError.code	Meaning	Suggested status
LIMIT_FILE_SIZE	File exceeds the `fileSize` limit	413
LIMIT_FILE_COUNT	More files than the `files` limit	400
LIMIT_UNEXPECTED_FILE	Field name not in the accepted list	400
LIMIT_FIELD_COUNT	Too many non-file fields	400
LIMIT_FIELD_VALUE_SIZE	Non-file field exceeds `fieldSize`	413

Storage Strategies

Local Disk

Use `diskStorage()`. Serve with `express.static('./uploads')`.

✓ Zero dependencies, simple setup, fast reads.

✗ Doesn't scale past one server. Files lost if server is replaced.

Use for: single-server apps, dev environments.

Memory → Cloud

Use `memoryStorage()`. In the route, stream `req.file.buffer` to S3/GCS/Azure Blob.

✓ Files accessible from any server. Durable, scalable.

✗ Extra round-trip latency. RAM spikes under load.

Use for: production multi-server apps.

Disk → Cloud (temp)

Use `diskStorage()` as a temp buffer, then stream the file to cloud and delete the temp file.

✓ Handles files too large to fit in RAM.

✗ Must clean up temp files on success AND error paths.

Use for: large file uploads (video, archives).

Memory → S3 Pattern

```
// npm install @aws-sdk/client-s3
const { S3Client, PutObjectCommand } = require('@aws-sdk/client-s3');
const multer = require('multer');
const crypto = require('node:crypto');
const path   = require('node:path');

const s3     = new S3Client({ region: process.env.AWS_REGION });
const BUCKET = process.env.S3_BUCKET;

const upload = multer({
    storage: multer.memoryStorage(),
    limits:  { fileSize: 5 * 1024 * 1024 },
});

app.post('/upload', upload.single('photo'), async (req, res) => {
    if (!req.file) throw createError(400, 'No file uploaded');

    const ext = path.extname(req.file.originalname).toLowerCase();
    const key = `uploads/${crypto.randomBytes(16).toString('hex')}` + ext;

    await s3.send(new PutObjectCommand({
        Bucket:      BUCKET,
        Key:         key,
        Body:        req.file.buffer,    // the in-memory bytes
        ContentType: req.file.mimetype,
    }));

    const url = `https://${BUCKET}.s3.amazonaws.com/${key}`;
    res.status(201).json({ url });
});
```

🔒 Upload Security Checklist

Never use `file.originalname` as the saved filename. A client can send `../../../etc/passwd` as the filename — using it directly is a path traversal vulnerability. Always generate a random name.

Never serve uploads from inside your app directory. If a malicious `.html` or `.js` file is uploaded and served from the same origin as your app, it can run with your app's cookies and permissions. Serve uploads from a separate subdomain, a CDN, or a storage bucket with its own domain.

Validate MIME type server-side. The `Content-Type` in the multipart request is supplied by the client — never trust it alone. Check magic bytes for critical file types.

Set `limits.fileSize` always. Without it, a malicious client can stream an unlimited file, exhausting disk or RAM.

Quick Reference

API	Notes
`multer({ storage, limits, fileFilter })`	Create an upload middleware instance. Reuse it across routes.
`upload.single('field')`	Accepts one file from field `'field'`. Puts it on `req.file`.
`upload.array('field', max)`	Accepts up to `max` files from one field. Result in `req.files`.

API	Notes
`upload.fields([...])`	Multiple field definitions. Result in `req.files` keyed by field name.
`req.file.buffer`	Raw bytes (memoryStorage only). Use to stream to S3 or process in memory.
`req.file.path`	Full path to the saved file (diskStorage only).
`req.file.mimetype`	Client-supplied MIME type. Validate but do not trust alone.
`req.file.size`	Bytes. Available after the file is received.
`err instanceof multer.MulterError`	True for limit violations. Check `err.code` for the specific limit.
`limits.fileSize`	Bytes. Enforced by multer during streaming — the route handler never runs for oversized files.

Coding Challenges

Challenge 1 — Single File Upload with Disk Storage

Build a `POST /upload/avatar` route using `diskStorage()` that saves uploaded files to an `uploads/` directory. Generate a random hex filename (preserving the original extension). Apply a `fileFilter` that accepts only JPEG, PNG, and WebP images (validate both MIME type and extension; reject with status 415). Set a 2 MB `fileSize` limit. The route should return `{ filename, size, mimetype }` on success. Add a `GET /uploads/:filename` route that serves the file using `res.sendFile()` with an absolute path. Write supertest tests using a real small JPEG buffer (construct a minimal valid JPEG: `Buffer.from([0xFF, 0xD8, 0xFF, 0xE0, ...])` or use a fixture file): 200 with a JSON body for a valid upload; 415 for a `.txt` file; 413 for a file exceeding 2 MB; 400 when no file is sent; the returned filename is NOT the original filename; a GET to `/uploads/:filename` with the returned filename returns 200.

View sample solution ↗

Challenge 2 — Multiple Fields with Different Validation Rules

Build a `POST /upload/profile` route using `upload.fields()` that accepts two fields: `avatar` (max 1 file, images only, 2 MB limit) and `documents` (max 3 files, PDFs only, 5 MB each). Since multer applies one `fileFilter` to all fields, inspect `file.fieldname` inside the filter to apply field-specific MIME type rules. After the upload middleware, validate in the route handler that at least one field was provided (reject 400 if both are absent). Return `{ avatar: { filename, size } | null, documents: [{ filename, size }] }`. Write supertest tests: 400 when no files are sent; valid avatar upload succeeds; valid documents upload (up to 3 PDFs) succeeds; avatar field rejects a PDF (415); documents field rejects a JPEG (415); uploading 4 documents triggers LIMIT_UNEXPECTED_FILE or LIMIT_FILE_COUNT (400).

View sample solution ↗

Challenge 3 — Memory Storage with Mock Cloud Upload

Build a `POST /upload/photo` route using `memoryStorage()`. After multer processes the file, pass `req.file.buffer`, `req.file.mimetype`, and a generated key to a `uploadToCloud(key, buffer, mimeType)` function (write this as a separate module so it can be mocked in tests). The function should return a `{ url }` object. If the cloud upload fails, delete nothing (memory storage has no disk file to clean up) and propagate a 502 error. Return `201 { url, size }` on success. Write tests using `jest.mock()` to replace the cloud upload module: mock resolves → 201 with url and size; mock rejects → route returns 502; no file sent → 400 before cloud upload is even called (assert the mock was NOT called); file exceeds 3 MB limit → 413 and mock not called. Use `jest.clearAllMocks()` in `beforeEach`.

View sample solution ↗

Security

> *Security in Express breaks down into four independent layers, each defending against a different class of attack: **helmet** sets HTTP headers that tell browsers how to behave (preventing clickjacking, MIME sniffing, and protocol downgrade); **CORS** controls which external origins may call your API (preventing cross-site data theft); **rate limiting** throttles how many requests an IP can make (preventing brute-force and abuse); and **input validation/sanitisation** ensures your application only processes well-formed data (preventing injections and crashes). None of these is sufficient alone — they stack.*

helmet — Security Headers

`helmet` is a collection of small middleware functions, each setting one security-related HTTP response header. Calling `helmet()` enables a sensible default set. Every header it sets tells the browser to restrict some potentially dangerous behavior — none of them affect your API logic.

```
// npm install helmet
const helmet = require('helmet');

// Use as early as possible — before routes, after app.use(express.json())
app.use(helmet());

// Disable or configure individual headers
app.use(helmet({
    // Content-Security-Policy — the most complex header; controls what the page can load
    contentSecurityPolicy: {
        directives: {
            defaultSrc:  ["'self'"],
            scriptSrc:   ["'self'", 'https://cdn.example.com'],
            styleSrc:    ["'self'", "'unsafe-inline'"],
            imgSrc:      ["'self'", 'data:', 'https:'],
            connectSrc:  ["'self'"],
            fontSrc:     ["'self'"],
            objectSrc:   ["'none'"],
            upgradeInsecureRequests: [],
        },
    },
    // HSTS — tell browsers to only connect over HTTPS for 1 year
    strictTransportSecurity: {
        maxAge:            31536000,
        includeSubDomains: true,
        preload:           true,
    },
    // Disable a header (pass false)
    xPoweredBy:       false,   // helmet already removes X-Powered-By by default
    frameguard:       { action: 'deny' },   // X-Frame-Options: DENY
    referrerPolicy:   { policy: 'strict-origin-when-cross-origin' },
```

```
}));
```

Header set by helmet	Value (default)	Prevents
`X-Frame-Options`	SAMEORIGIN	Clickjacking — embedding your page in another site's iframe
`X-Content-Type-Options`	nosniff	MIME sniffing — browser interpreting a JS file as an image or vice versa
`Strict-Transport-Security`	max-age=15552000	Protocol downgrade — forces HTTPS for future visits
`Content-Security-Policy`	default-src 'self'; ...	XSS — restricts which scripts, styles, images the page can load
`Referrer-Policy`	no-referrer	Leaking URL paths in the Referer header to third parties
`Permissions-Policy`	disables many APIs	Malicious use of camera, microphone, geolocation from injected scripts
`X-DNS-Prefetch-Control`	off	DNS prefetching leaking visited URLs
`Cross-Origin-Opener-Policy`	same-origin	Cross-origin window.opener access (Spectre side-channels)

CORS — Cross-Origin Resource Sharing

Browsers block JavaScript from reading responses from a different origin (protocol + host + port). That's the same-origin policy. CORS is the mechanism that lets servers opt specific origins back in. Without CORS configuration, your API is only callable from the same origin — any SPA on a different domain gets a network error, even for valid requests.

Simple request (no preflight)

GET or POST with basic Content-Types. Browser sends request directly; server responds with `Access-Control-Allow-Origin`. Browser decides whether to expose the response to JS.

Preflight request (OPTIONS first)

Any request with custom headers (e.g. `Authorization`, `Content-Type: application/json`). Browser sends OPTIONS first. Server must respond with the CORS headers. If approved, the real request follows.

```
// npm install cors
const cors = require('cors');

// ① Development — allow everything (NEVER in production)
app.use(cors());

// ② Production — static allowlist
const ALLOWED_ORIGINS = new Set([
    'https://app.example.com',
    'https://admin.example.com',
]);

app.use(cors({
    origin(requestOrigin, cb) {
        // requestOrigin is undefined for non-browser requests (curl, server-to-server)
        if (!requestOrigin || ALLOWED_ORIGINS.has(requestOrigin)) {
            cb(null, true);    // allow
        } else {
            cb(new Error(`Origin ${requestOrigin} not allowed`));
        }
    },
```

```
    methods:          ['GET', 'POST', 'PUT', 'PATCH', 'DELETE'],
    allowedHeaders:   ['Content-Type', 'Authorization'],
    credentials:      true,   // allow cookies/auth headers in cross-origin requests
    maxAge:           86400,  // cache preflight result for 24 h (reduces OPTIONS requests)
}));

// Must explicitly handle OPTIONS for all routes when credentials: true
app.options('*', cors());
```

⚠ credentials: true requires an explicit origin

If you set `credentials: true`, you cannot use `origin: '*'`. Browsers will block the request — the wildcard and credentials cannot be combined per the CORS spec. You must return the specific requesting origin (or a specific string) in `Access-Control-Allow-Origin`. The dynamic origin callback above handles this correctly.

Rate Limiting

Rate limiting caps how many requests a single IP address can make in a time window. Without it, a single client can hammer a login endpoint millions of times trying passwords (brute force), exhaust your server CPU, or run up your API costs. The key insight is that different routes warrant different limits: a login endpoint should be very strict, a public read endpoint can be generous.

```
// npm install express-rate-limit
const rateLimit = require('express-rate-limit');

// General API limit — 100 requests per 15 minutes per IP
const apiLimiter = rateLimit({
    windowMs:        15 * 60 * 1000,   // 15 min in ms
    max:             100,
    standardHeaders: true,    // RateLimit-* headers (RFC draft standard)
    legacyHeaders:   false,   // don't also send deprecated X-RateLimit-* headers
    message:         { error: 'Too many requests, please try again later', status: 429 },
});

// Strict login limit — 5 attempts per 15 minutes per IP
const loginLimiter = rateLimit({
    windowMs: 15 * 60 * 1000,
    max:      5,
    message:  { error: 'Too many login attempts, please try again in 15 minutes', status: 429 },
    // Don't count successful logins against the limit
    skipSuccessfulRequests: true,
});

// Apply generally, override for sensitive routes
app.use('/api', apiLimiter);
app.use('/auth/login', loginLimiter);  // applied first = stricter wins
```

```
// Behind a reverse proxy (nginx, load balancer) — use X-Forwarded-For for real IPs
app.set('trust proxy', 1);   // trust first hop (nginx → Express)

// Custom key generator — rate-limit by user ID after authentication instead of IP
const authenticatedLimiter = rateLimit({
    windowMs: 60 * 1000,
    max:      30,
    keyGenerator: (req) => req.user?.sub ?? req.ip,   // fall back to IP if not authed
    skip:         (req) => req.user?.role === 'admin', // admins bypass the limit
});
```

Headers sent by rate limiting

With `standardHeaders: true`, every response includes `RateLimit-Limit` (the cap), `RateLimit-Remaining` (requests left in the window), and `RateLimit-Reset` (Unix timestamp when the window resets). When the limit is hit, the response is 429 and includes `Retry-After` (seconds to wait). Client code can read these to implement back-off.

Input Validation & Sanitisation

Validation asks: "is this value acceptable?" **Sanitisation** asks: "can I clean this value so it's safe to use?" They solve different problems. Validation rejects bad data at the boundary. Sanitisation normalises data — trimming whitespace, lowercasing emails, converting strings to integers. The critical rule: *sanitise input to normalise it, not to make it "safe" from injection* — that's what parameterised queries (SQL) and output encoding (XSS) are for.

```
// npm install express-validator
const { body, param, query, validationResult } = require('express-validator');

// Validation chain — each method is a step in the pipeline
const registerRules = [
    body('email')
        .trim().toLowerCase()       // sanitise first
        .notEmpty().withMessage('email is required')
        .isEmail().withMessage('must be a valid email address')
        .isLength({ max: 254 }).withMessage('email too long'),

    body('password')
        .notEmpty().withMessage('password is required')
        .isLength({ min: 8, max: 72 }).withMessage('password must be 8–72 characters')
        .matches(/[A-Z]/).withMessage('password must contain an uppercase letter')
        .matches(/[0-9]/).withMessage('password must contain a number'),

    body('age')
        .optional()
        .toInt()                    // sanitise: string '25' → number 25
        .isInt({ min: 13, max: 120 }).withMessage('age must be between 13 and 120'),

    body('username')
        .trim()
        .notEmpty().withMessage('username is required')
        .isLength({ min: 3, max: 30 }).withMessage('username must be 3–30 characters')
```

```
        .matches(/^[a-zA-Z0-9_]+$/).withMessage('username may only contain letters, numbers and
];

// Middleware that reads the result and returns 422 if any rule failed
function validate(req, res, next) {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(422).json({
            error:  'Validation failed',
            errors: errors.array().map(({ path, msg }) => ({ field: path, message: msg })),
        });
    }
    next();
}

// Route: validation rules run as middleware before the handler
app.post('/auth/register', registerRules, validate, async (req, res) => {
    // By this point: req.body.email is trimmed+lowercased, req.body.age is a number
    const { email, password, username, age } = req.body;
    // ... create user
    res.status(201).json({ id: 1, email, username });
});
```

Custom Validators

```
// Check something that built-in validators can't — e.g. uniqueness in the DB
body('email')
    .trim().toLowerCase().isEmail()
    .custom(async (value) => {
        const existing = await db.findByEmail(value);
        if (existing) throw new Error('Email already registered');
    }),

// Cross-field validation — confirm password must match password
body('confirmPassword')
    .custom((value, { req }) => {
        if (value !== req.body.password) throw new Error('Passwords do not match');
        return true;
    }),

// Validate a URL param
param('id')
    .isInt({ min: 1 }).withMessage('id must be a positive integer')
    .toInt(),    // converts req.params.id from string '5' to number 5

// Validate a query string param
query('page')
    .optional()
    .isInt({ min: 1 }).withMessage('page must be a positive integer')
    .toInt()
```

```
    .default(1),  // if absent, set req.query.page = 1
```

Sanitiser	What it does
`.trim()`	Remove leading/trailing whitespace
`.toLowerCase()`	Normalise to lowercase (email, username)
`.toInt()`	Parse to integer — NaN if not a valid integer
`.toFloat()`	Parse to float
`.toBoolean()`	"true"/"1" → true, anything else → false
`.escape()`	HTML-encode `< > & " '` — use for rendering user content in HTML templates, not as an XSS defence for APIs
`.normalizeEmail()`	Normalise email per RFC: lowercase, strip dots from Gmail, etc.
`.default(value)`	Set a default if the field is absent

Putting It All Together

```
// app.js — the four layers in registration order
require('express-async-errors');
const express   = require('express');
const helmet    = require('helmet');
const cors      = require('cors');
const rateLimit = require('express-rate-limit');

const app = express();

// 1. Security headers — first, before anything that touches the response
app.use(helmet());

// 2. CORS — before routes, but after helmet (helmet runs on all requests)
app.use(cors({ origin: ALLOWED_ORIGINS, credentials: true }));
app.options('*', cors());

// 3. Body parsing
app.use(express.json({ limit: '10kb' }));  // limit body size too

// 4. Rate limiting
app.use('/api',        apiLimiter);
app.use('/auth/login', loginLimiter);

// 5. Routes (with validation middleware inline per route)
app.use('/api/v1', apiRoutes);
app.use('/auth',    authRoutes);

// 6. Error handler — last
app.use(errorHandler);
```

🔒 What each layer does NOT protect against

helmet doesn't prevent bad data — it sets headers. It has no effect on server-side logic or database operations.

CORS is a browser-enforcement mechanism — it doesn't block server-to-server calls, curl, or Postman. Never treat CORS as an authentication or authorisation mechanism.

Rate limiting by IP is defeated by distributed botnets. It slows attackers but doesn't stop determined ones. Always combine with account lockouts and MFA.

Input sanitisation is not an XSS defence — output encoding is. Sanitisation is for normalisation (trimming whitespace, lowercasing) and validation is for rejection. SQL injection is defeated by parameterised queries, not escaping.

Coding Challenges

Challenge 1 — helmet & CORS

Build an Express app with both `helmet` and `cors` configured for production. The CORS allowlist should contain two hardcoded origins: `https://app.example.com` and `https://admin.example.com`. Use a dynamic origin callback so `credentials: true` works correctly. Set `maxAge: 86400` and allow `Authorization` and `Content-Type` headers. Register `app.options('*', cors())` for preflight support. Add a `GET /api/data` route that returns `{ message: 'ok' }`. Write supertest tests: a request from an allowed origin gets `Access-Control-Allow-Origin` matching that origin; a request from a disallowed origin does NOT get `Access-Control-Allow-Origin` (or gets an error); the response includes the security headers set by helmet (`X-Frame-Options`, `X-Content-Type-Options`); an OPTIONS preflight to `/api/data` from an allowed origin returns 204 with `Access-Control-Allow-Methods`; the response does NOT include `X-Powered-By: Express`.

View sample solution ↗

Challenge 2 — Rate Limiting

Add two rate limiters to an app: a general `apiLimiter` (10 requests per minute per IP — low cap makes it easy to test) applied to `/api`, and a strict `loginLimiter` (3 attempts per minute) applied to `POST /auth/login`. Both should use `standardHeaders: true` and return a JSON error with `status: 429` when hit. Write supertest tests: the first N requests to `/api/data` return 200; the (N+1)th request returns 429 with a JSON error body; the response headers include `RateLimit-Limit` and `RateLimit-Remaining`; the login endpoint has a tighter limit (hits 429 sooner than the API limiter); a `skip` function that exempts requests with header `X-Admin: true` allows unlimited requests from that header. (Hint: supertest fires requests against the same app instance in the same process — the in-memory counter persists between requests in one test, so you can exhaust it by looping.)

View sample solution ↗

Challenge 3 — Input Validation & Sanitisation

Build a `POST /auth/register` endpoint with a `express-validator` chain. Rules: `email` — trim, lowercase, required, valid email format, max 254 chars; `password` — required, 8–72 chars, must contain at least one uppercase letter and one digit; `username` — trim, required, 3–30 chars, only letters/numbers/underscores; `age` — optional, convert to integer, must be 13–120. A reusable `validate` middleware reads `validationResult(req)` and returns 422 with `{ errors: [{ field, message }] }` if any rule fails. On success return 201 with the sanitised values (email lowercased, username trimmed, age as integer). Write supertest tests: 201 for valid input; 422 for missing email; 422 for invalid email format; 422 for password under 8 chars; 422 for password without a digit; 422 for username with spaces; 422 for age of 10 (under minimum); the response body for a valid POST contains the email in lowercase (even if submitted in mixed case); `age` in the response is a number, not a string.

View sample solution ↗

Testing Express APIs

> *Testing a REST API has two distinct layers. **Unit tests** call your service and utility functions directly — no HTTP involved, fast, easy to isolate. **Integration tests** fire real HTTP requests at your Express app using supertest and verify the full request/response cycle. The MVC split from express1-8 pays off here directly: because services have no HTTP dependency, they can be unit-tested without a server; because controllers are thin, integration tests cover them completely via supertest. The key enabling move is mocking the database layer so tests don't need a running MySQL instance.*

Test Types at a Glance

Unit Tests

Call functions directly. Mock all dependencies. No network, no DB.

Target: services, validators, utilities, error mappers

Tool: Jest alone

Fast — milliseconds each

Integration Tests

Fire HTTP via supertest. Mock the DB layer. Test routes end-to-end.

Target: controllers, routes, middleware

Tool: Jest + supertest + jest.mock()

Medium — tens of ms each

End-to-End Tests

Real DB, real server, real HTTP. Everything running together.

Target: critical user flows

Tool: Jest + supertest + test DB

Slow — real I/O on every request

supertest Basics

supertest lets you make HTTP requests against an Express app without binding to a port. You pass the app directly — supertest opens a temporary connection for the duration of the request and closes it. No `app.listen()` call needed in your test file.

```
// npm install --save-dev supertest jest
const request = require('supertest');
const app     = require('./app');   // app.js exports the app, never calls listen()

// One-shot request — new connection each call
const res = await request(app)
    .get('/api/products')
    .set('Authorization', `Bearer ${token}`)
    .expect(200)                    // built-in assertion (throws on mismatch)
    .expect('Content-Type', /json/);  // also works with regex

// Persistent agent — keeps cookies between requests (auth sessions, refresh tokens)
const agent = request.agent(app);
await agent.post('/auth/login').send({ email: 'a@b.com', password: 'pass' });
const profile = await agent.get('/me');   // cookie sent automatically

// Common assertions on the result object
expect(res.status).toBe(201);
expect(res.body.id).toBeDefined();
expect(res.headers['location']).toMatch(/\/api\/products\/\d+/);
```

```
expect(res.body.error).toMatch(/not found/i);
```

> **app.js must NOT call listen()**
>
> supertest creates its own temporary server. If `app.js` calls `app.listen()` at module-load time, you get port conflicts and Jest won't exit cleanly. The pattern: `app.js` exports the app only; `server.js` (never imported in tests) calls `listen()`. This is the pattern established in express1-8 and pays off fully here.

Mocking the Database Layer

`jest.mock('./path/to/module')` replaces every export of that module with `jest.fn()` before any test file imports it. Jest hoists the call above all `require()`s, so even modules that are imported before the mock call see the mocked version. This is the standard way to test controllers without a real database.

```
// Suppose app.js → controller → productsService → ProductModel (the DB layer)
// We mock ProductModel so nothing ever touches MySQL

jest.mock('./src/models/product');   // hoisted — runs before any require()

const request      = require('supertest');
const app          = require('./app');
const ProductModel = require('./src/models/product');  // now all jest.fn()s

// Reset call counts and return values before each test
beforeEach(() => jest.clearAllMocks());

describe('GET /api/products', () => {
    test('200 and returns the list from the model', async () => {
        // Arrange — tell the mock what to return
        ProductModel.findAll.mockResolvedValue([
            { id: 1, name: 'Widget', price: 9.99 },
            { id: 2, name: 'Gadget', price: 24.99 },
        ]);

        // Act
        const res = await request(app).get('/api/products');

        // Assert
        expect(res.status).toBe(200);
        expect(res.body).toHaveLength(2);
        expect(ProductModel.findAll).toHaveBeenCalledTimes(1);
    });

    test('500 when the model throws an unexpected error', async () => {
        ProductModel.findAll.mockRejectedValue(new Error('DB connection lost'));
        const res = await request(app).get('/api/products');
        expect(res.status).toBe(500);
        // Internal message must NOT be in the response body
        expect(JSON.stringify(res.body)).not.toMatch(/DB connection lost/);
    });
```

```
});
```

jest.fn() Cheat Sheet

Call	Behaviour
`jest.fn()`	Creates a mock function that returns `undefined` and records all calls.
`fn.mockReturnValue(x)`	Returns `x` synchronously every time called.
`fn.mockResolvedValue(x)`	Returns `Promise.resolve(x)` — use for async model methods.
`fn.mockRejectedValue(err)`	Returns `Promise.reject(err)` — simulates DB errors.
`fn.mockResolvedValueOnce(x)`	Returns `x` on the next call only, then falls through to the default.
`fn.mockImplementation(impl)`	Replaces the function body — for complex conditional behaviour.
`jest.clearAllMocks()`	Resets call counts and return values on all mocks. Put in `beforeEach`.
`jest.spyOn(obj, 'method')`	Wraps an existing method without replacing it — call `.mockRestore()` to undo.
`fn.mock.calls`	Array of argument arrays for each call: `fn.mock.calls[0][1]` = first call, second arg.
`expect(fn).toHaveBeenCalledWith(a, b)`	Asserts the mock was called with specific arguments.

Unit Testing Services

Services contain business logic — validation, error mapping, orchestration. They're pure functions of their inputs and their model dependencies. Mock the model, call the service directly, assert the result or the thrown error. No HTTP, no supertest.

```
jest.mock('./src/models/product');

const ProductModel     = require('./src/models/product');
const productsService  = require('./src/services/productsService');

beforeEach(() => jest.clearAllMocks());

describe('productsService.get(id)', () => {
    test('returns the product when found', async () => {
        ProductModel.findById.mockResolvedValue({ id: 1, name: 'Widget' });
        const result = await productsService.get(1);
        expect(result.name).toBe('Widget');
    });

    test('throws 404 when model returns null', async () => {
        ProductModel.findById.mockResolvedValue(null);
        await expect(productsService.get(99))
            .rejects.toMatchObject({ status: 404 });
    });
});

describe('productsService.create(data)', () => {
    test('throws 422 for missing name — without calling the model', async () => {
        await expect(productsService.create({ price: 5 }))
```

```
            .rejects.toMatchObject({ status: 422 });
        expect(ProductModel.create).not.toHaveBeenCalled();
    });

    test('calls model.create with trimmed name', async () => {
        ProductModel.create.mockResolvedValue({ id: 3, name: 'Trimmed', price: 10 });
        await productsService.create({ name: '  Trimmed  ', price: 10 });
        expect(ProductModel.create).toHaveBeenCalledWith(
            expect.objectContaining({ name: 'Trimmed' })
        );
    });
});
```

Testing Middleware

Authentication middleware is best tested in two ways: unit tests that call the middleware function directly with mock `req` / `res` / `next` objects, and integration tests via supertest that send a real token to a protected route. The integration path is simpler and covers more — use it for the main cases, unit tests for edge cases.

```
// Integration approach — generate a real token with jwt.sign() in the test
const jwt = require('jsonwebtoken');
const SECRET = process.env.JWT_SECRET || 'test-secret';

function tokenFor(payload) {
    return `Bearer ${jwt.sign(payload, SECRET, { algorithm: 'HS256', expiresIn: '15m' })}`;
}
function expiredToken() {
    return `Bearer ${jwt.sign({ sub: 1, role: 'user' }, SECRET, { expiresIn: '-1s' })}`;
}

// In tests — no login route needed at all
const res = await request(app)
    .get('/api/me')
    .set('Authorization', tokenFor({ sub: 1, role: 'admin' }));
expect(res.status).toBe(200);

// Unit approach — test the middleware function itself
const authenticate = require('./src/middleware/authenticate');

test('calls next(err) with 401 for missing header', () => {
    const req  = { get: () => undefined };
    const res  = {};
    const next = jest.fn();
    authenticate(req, res, next);
    expect(next).toHaveBeenCalledWith(
        expect.objectContaining({ status: 401 })
    );
});
```

Test Organisation

```
// jest.config.js — recommended settings for Express API projects
module.exports = {
    testEnvironment: 'node',        // not 'jsdom' — this is a server
    forceExit:       true,          // close after all tests (Express keeps event loop open)
    clearMocks:      true,          // automatically clear mock state between tests
    testMatch:       ['**/*.test.js'],
    coverageThreshold: {
        global: { lines: 80, functions: 80, branches: 70 },
    },
};

// ── Typical test file structure ──────────────────────────────────────
jest.mock('./src/models/product');   // hoisted to top by jest — always first

const request      = require('supertest');
const app          = require('./app');
const ProductModel = require('./src/models/product');

// Data factory — override only what each test needs to change
const makeProduct = (overrides = {}) => ({
    id: 1, name: 'Widget', price: 9.99, stock: 5, ...overrides,
});

beforeEach(() => jest.clearAllMocks());
beforeAll(() => jest.spyOn(console, 'error').mockImplementation(() => {}));
afterAll(() => jest.restoreAllMocks());

describe('GET /api/products', () => { /* ... */ });
describe('GET /api/products/:id', () => { /* ... */ });
describe('POST /api/products', () => { /* ... */ });
```

⚠ jest.mock() is hoisted — the path must be a literal string

Jest's Babel transform moves `jest.mock()` calls to the top of the file before any imports. This means you cannot use a variable in the path: `jest.mock(MODELS_PATH)` will fail because the variable hasn't been assigned yet. Always use a literal string: `jest.mock('./src/models/product')`. This is the most common "why isn't my mock working?" mistake.

Quick Reference

API	Notes
`request(app).get(url)`	Make a one-shot GET. Returns a thenable — await it to get the response object.
`.send(body)`	Set the request body. Also sets Content-Type: application/json automatically.
`.set(header, value)`	Set a request header. For auth: `.set('Authorization', 'Bearer ...')`.
`.attach(field, buffer, opts)`	Attach a file for multipart/form-data requests (multer).

API	Notes
`.expect(status)`	Built-in status assertion — throws if the status doesn't match.
`request.agent(app)`	Persistent agent — cookies are sent on every subsequent request.
`jest.mock('./path')`	Replace all module exports with jest.fn(). Must be a literal string.
`fn.mockResolvedValue(x)`	Make an async mock return x. Required for all async model methods.
`fn.mockRejectedValue(err)`	Simulate a rejected promise from the model.
`jest.clearAllMocks()`	Reset call counts and mock return values. Put in beforeEach.
`expect(fn).not.toHaveBeenCalled()`	Verify a mock was never called — useful to confirm short-circuit paths.
`expect(obj).toMatchObject(partial)`	Passes if obj contains all keys in partial. Used for error objects: `{ status: 404 }`.

Coding Challenges

Challenge 1 — Unit Testing a Service

Write a `productsService` with four methods: `list()`, `get(id)`, `create(data)`, `remove(id)`. The service validates input (`name` required, `price` must be a non-negative number) and maps model results to errors (model returns null → service throws 404). Use `jest.mock()` to replace the model and write unit tests for all four methods — no supertest, no HTTP. Tests must cover: `list()` returns whatever the model returns; `get(id)` returns the product when found and throws status 404 when model returns null; `create(data)` throws 422 for missing name without calling the model; `create(data)` calls the model with trimmed name; `create(data)` propagates a DB 409 error from the model; `remove(id)` returns true on success and throws 404 when model returns false. Use `jest.clearAllMocks()` in `beforeEach`. Use a data factory function for product fixtures.

View sample solution ↗

Challenge 2 — Integration Testing Routes with a Mocked Service

Take a complete products CRUD Express app (routes → controller → service → model). Use `jest.mock()` to mock the **service layer** (not the model — the service is the dependency boundary for controllers). Write integration tests using supertest for all five routes: `GET /api/products`, `GET /api/products/:id`, `POST /api/products`, `PATCH /api/products/:id`, `DELETE /api/products/:id`. Each route needs at least: the happy-path (correct status code, response body shape, headers like Location); the not-found case (service throws 404 → route returns 404 JSON); where applicable, the validation case (service throws 422 → route returns 422 with details). Also verify: the 500 case (service throws an unexpected error → route returns 500 with a sanitised message, not the raw error message); `console.error` is silenced with `jest.spyOn`. Use `expect(fn).toHaveBeenCalledWith()` on at least one test to confirm the controller passes the right arguments to the service.

View sample solution ↗

Challenge 3 — Testing Authentication Middleware

Write tests for an `authenticate` middleware and a `requireRole` factory (from express2-1). Use both approaches: (A) **Unit tests** — call the middleware directly with mock `req` / `res` / `next` objects. Test: missing Authorization header calls `next(err)` with status 401; non-Bearer token calls `next(err)` with status 401; a tampered token calls `next(err)` with status 401; an expired token calls `next(err)` with a message matching "expired"; a valid token sets `req.user` and calls `next()` with no arguments; `requireRole('admin')` calls `next(err)` with status 403 when `req.user.role` is 'user'. (B) **Integration tests via supertest** — mount the middleware on a test route in a minimal app. Test: no token → 401; expired token → 401; valid token with correct role → 200; valid token with wrong role → 403. Generate tokens directly with `jwt.sign()` — no login route needed.

View sample solution ↗

> *HTTP is half-duplex: the client asks, the server answers, and the connection closes. WebSockets flip that model — after a single HTTP upgrade handshake the connection stays open, and either side can push data at any moment. Socket.IO wraps raw WebSockets with rooms, namespaces, acknowledgements, automatic reconnection, and transparent fallbacks, making real-time features practical without re-inventing the protocol layer.*

HTTP vs WebSocket

HTTP (request / response)

→ Client always initiates

→ New TCP connection per request (HTTP/1.1: keep-alive helps, not the same)

→ Server cannot push unsolicited data

→ Polling: client hammers server every N seconds

→ Best for: CRUD APIs, file downloads, page loads

WebSocket (persistent channel)

→ Either side can send at any time

→ Single TCP connection, stays open

→ Server pushes events the moment they happen

→ No polling overhead — binary or text frames

→ Best for: chat, live scores, collaborative editing, notifications

The upgrade from HTTP to WebSocket happens via a standard header exchange: the client sends `Upgrade: websocket`, the server responds with `101 Switching Protocols`, and from that point the TCP socket is handed to the WebSocket layer.

Why Socket.IO over Raw WebSockets?

Feature	Raw `ws`	Socket.IO
`Transport fallback`	WebSocket only	Polling → WebSocket upgrade automatically
`Named events`	One `message` event — you parse JSON yourself	`socket.on('chat-message', handler)` built-in
`Rooms`	Manual Set tracking	`socket.join(room)` — one call
`Acknowledgements`	Roll your own request-ID map	Callback argument on `emit`
`Reconnection`	You write it	Built-in with exponential back-off
`Middleware`	Not built in	`io.use((socket, next) => {})`
`Namespaces`	Not built in	`io.of('/admin')`

Trade-off

Socket.IO is not a strict WebSocket implementation — a raw WebSocket client cannot connect to a Socket.IO server. Both sides must use Socket.IO's protocol. This is fine for apps you control end-to-end; if you're building a public API that third-party clients connect to, use raw WebSockets instead.

Server Setup

Socket.IO attaches to a plain Node `http.Server`, not to the Express app directly. Create the HTTP server first, pass it to Socket.IO, then listen on that server — not on `app.listen()`.

```
const express        = require('express');
const http           = require('http');
const { Server }     = require('socket.io');

const app        = express();
const httpServer = http.createServer(app);
const io         = new Server(httpServer, {
  cors: {
    origin:      'http://localhost:5173',  // your frontend origin
    credentials: true,
  },
});

// Express routes work exactly as before
app.get('/api/status', (_req, res) => res.json({ ok: true }));

// All Socket.IO logic under io.on()
io.on('connection', (socket) => {
  console.log('connected:', socket.id);
});

httpServer.listen(3000, () => console.log('Listening on :3000'));
```

Installation

`npm install socket.io` — server only.
For the browser: `npm install socket.io-client` (or load from the Socket.IO CDN).
For tests: `npm install --save-dev socket.io-client`.

Connection Lifecycle

`io.on('connection', socket => {})`	New client connected — `socket.id` is a unique string
`socket.on('my-event', handler)`	Register listeners for events the client emits
`socket.emit('event', data)`	Push an event to this specific client
`socket.on('disconnect', reason => {})`	Client disconnected — clean up state (rooms auto-cleared)

```
io.on('connection', (socket) => {
  console.log(`+ connected  ${socket.id}`);

  socket.on('ping', (callback) => {
    if (typeof callback === 'function') callback({ pong: Date.now() });
  });

  socket.on('disconnect', (reason) => {
    console.log(`- disconnected ${socket.id} (${reason})`);
  });
});
```

Each `socket` object inside `io.on('connection')` represents one client. `socket.id` is stable for the lifetime of the connection. Custom data tied to a socket (username, room, role) is stored on `socket.data` — a plain object that is not sent to the client.

Emitting Events — Who Receives What

The most important thing to get right with Socket.IO: every emit target is a different audience. Learn these five methods before writing any real-time feature.

Method	Receives it	Scope
`socket.emit('event', data)`	Only this client (the sender)	self
`io.emit('event', data)`	Every connected client including sender	all
`socket.broadcast.emit('event', data)`	Every connected client **except** sender	others
`io.to(room).emit('event', data)`	All sockets in room, **including** sender if they joined it	room (all)
`socket.to(room).emit('event', data)`	All sockets in room, **excluding** sender	room (others)

> **io.to() vs socket.to()**
>
> The only difference is whether the sender receives the event. Use `io.to(room).emit()` when the sender should see their own action (e.g. a new chat message appearing in the sender's own UI). Use `socket.to(room).emit()` to notify others without echoing back to the sender (e.g. "Alice joined the room").

Rooms

Rooms are named groups of sockets. Every socket is automatically in a private room named after its own `socket.id`, so you can DM a specific socket with `io.to(socket.id).emit()`. You can also create named rooms for chat channels, game lobbies, or any grouping concept.

```
io.on('connection', (socket) => {

  socket.on('join-room', ({ room, username }) => {
    socket.join(room);                  // socket is now in this room
    socket.data.username = username;
    socket.data.room     = room;

    // Notify everyone else in the room (not the joiner)
    socket.to(room).emit('user-joined', { username });
  });

  socket.on('send-message', ({ message }) => {
    const { room, username } = socket.data;
    if (!room) return;

    // io.to() includes the sender — they see their own message
    io.to(room).emit('new-message', { username, message, ts: Date.now() });
  });

  socket.on('leave-room', () => {
    const { room, username } = socket.data;
    socket.leave(room);
    socket.to(room).emit('user-left', { username });
    socket.data.room = null;
  });
```

```
  socket.on('disconnect', () => {
    // socket automatically leaves all rooms on disconnect
    const { room, username } = socket.data;
    if (room) socket.to(room).emit('user-left', { username });
  });
});
```

A socket can be in multiple rooms simultaneously. `socket.rooms` is a `Set` containing the current room names (always includes `socket.id`). `socket.leave(room)` removes it from that room without disconnecting.

Socket.IO Middleware — Authentication

`io.use()` works like Express middleware but runs once per connection attempt, before `io.on('connection')` fires. Calling `next(new Error(...))` rejects the connection; the client receives a `connect_error` event with the error message.

```
const jwt    = require('jsonwebtoken');
const SECRET = process.env.JWT_SECRET;

io.use((socket, next) => {
  // Client passes token at connection time:
  // const socket = io(URL, { auth: { token: 'Bearer eyJ...' } });
  const token = socket.handshake.auth.token;

  if (!token) {
    return next(new Error('Authentication required'));
  }

  try {
    socket.data.user = jwt.verify(token, SECRET, { algorithms: ['HS256'] });
    next();
  } catch (err) {
    const message = err.name === 'TokenExpiredError' ? 'Token expired' : 'Invalid token';
    next(new Error(message));
  }
});

io.on('connection', (socket) => {
  // socket.data.user is populated — safe to read
  console.log('authed user:', socket.data.user.sub);
});
```

Handshake property	What it contains
`socket.handshake.auth`	Object passed by client as the second arg to `io(url, { auth: {...} })`
`socket.handshake.headers`	HTTP headers from the upgrade request (cookies, user-agent, etc.)
`socket.handshake.query`	Query-string params from the connection URL
`socket.handshake.address`	Client IP address
`socket.data`	Plain object you own — store server-side state here (user, room, etc.)

Acknowledgements

Acknowledgements turn a fire-and-forget emit into a round-trip. The client passes a callback as the last argument to `socket.emit()`; the server calls it with a response. This replaces the need for a separate "response" event and removes the request-ID tracking problem entirely.

```
// ── Server ──────────────────────────────────────────────────
socket.on('create-item', async (data, callback) => {
  if (typeof callback !== 'function') return;  // guard: client may not pass one

  if (!data?.name?.trim()) {
    return callback({ error: 'name is required' });
  }

  const item = await db.insert(data);
  callback({ success: true, item });   // triggers the client callback
});


// ── Client ──────────────────────────────────────────────────
socket.emit('create-item', { name: 'Widget' }, (response) => {
  if (response.error) {
    console.error('Failed:', response.error);
  } else {
    console.log('Created:', response.item);
  }
});


// ── In tests (socket.io-client) ─────────────────────────────
const response = await new Promise((resolve) => {
  client.emit('create-item', { name: 'Widget' }, resolve);
});
expect(response.success).toBe(true);
```

> **Always guard typeof callback === 'function'**
>
> If the client emits an event without a callback and the server calls `callback()` unconditionally, Socket.IO throws `TypeError: callback is not a function`. The guard on line 2 above prevents this.

Namespaces

Namespaces multiplex multiple logical connections over a single TCP socket. The default namespace is `/`. A named namespace `io.of('/chat')` is a fully independent channel with its own connection events, middleware, and rooms — useful for separating concerns (chat vs admin vs notifications) without spinning up multiple servers.

```
// Server
const chat  = io.of('/chat');
const admin = io.of('/admin');

chat.use(requireAuth);   // middleware only on /chat
admin.use(requireAdmin);

chat.on('connection', (socket) => { /* chat events */ });
admin.on('connection', (socket) => { /* admin events */ });

// Client
const chatSocket  = io('http://localhost:3000/chat',  { auth: { token } });
const adminSocket = io('http://localhost:3000/admin', { auth: { token } });
```

> **Namespaces vs Rooms**

> Rooms are lightweight named groups within a namespace — use them for channels, lobbies, or any dynamic grouping. Namespaces are heavier — separate connection and event trees. Prefer rooms unless you genuinely need namespace-level isolation (different middleware, different transport options).

Testing with socket.io-client

Socket.IO tests use the `socket.io-client` package to create real in-process connections to a test server. Listen on port `0` (OS assigns a free port), read it back from `httpServer.address().port`, and connect the test client there.

```
// jest.config.js
module.exports = {
  testEnvironment: 'node',
  forceExit:       true,   // socket.io keeps the event loop open
  clearMocks:      true,
};


// chat.test.js
const { io: ioc }        = require('socket.io-client');
const { httpServer, io } = require('./server');

let port;

// Helper: resolve when an event fires once
function waitForEvent(socket, event) {
  return new Promise((resolve) => socket.once(event, resolve));
}

// Helper: resolve when the client is connected
function connect(opts = {}) {
  return new Promise((resolve) => {
    const client = ioc(`http://localhost:${port}`, {
      transports: ['websocket'],  // skip polling, faster tests
      forceNew:   true,           // each call = fresh socket (no reuse)
      ...opts,
    });
    client.once('connect', () => resolve(client));
  });
}

// Helper: small pause so a server event can propagate to recipients
const tick = () => new Promise((r) => setTimeout(r, 20));

beforeAll((done) => {
  httpServer.listen(() => {
    port = httpServer.address().port;
    done();
  });
});

afterAll((done) => {
  io.close();               // closes all sockets + the http server internally
  httpServer.close(done);
});

test('server echoes message to sender', async () => {
  const client = await connect();
  client.emit('join-room', { room: 'test', username: 'alice' });
  await tick();

  client.emit('send-message', { message: 'hello' });
```

```
  const msg = await waitForEvent(client, 'new-message');
  expect(msg.message).toBe('hello');
  client.disconnect();
});
```

forceNew: true — why it matters

Without `forceNew: true`, `socket.io-client` reuses an existing connection to the same URL. In tests, multiple `connect()` calls in the same file would share one socket, and tests that need independent clients would fail. `forceNew: true` guarantees a fresh socket each time.

The tick() pattern

After emitting an event that triggers a server broadcast to other clients, you need a brief pause to allow the message to propagate over the loopback connection. A 20–50ms `setTimeout` is idiomatic in Socket.IO integration tests. For stricter sequencing, use acknowledgements on the emitting event instead.

Coding Challenges

Challenge 1 — Chat Room

Build a Socket.IO server that supports named chat rooms. Clients emit `join-room`, `send-message`, and `leave-room`. Messages broadcast to the whole room (sender included). Joining and leaving emit `user-joined` / `user-left` to the room's other members. Write integration tests using `socket.io-client` verifying room isolation, echo to sender, cross-room silence, and cleanup on disconnect.

View solution

Challenge 2 — JWT Auth Middleware

Add an `io.use()` middleware that reads a JWT from `socket.handshake.auth.token`, verifies it, and stores the decoded payload on `socket.data.user`. Rejected connections should carry a descriptive error message. Add a `whoami` ack event that returns the authenticated user, and a `broadcast` event that sends an `announcement` to all connected clients. Test all four auth failure cases plus the authenticated happy paths.

View solution

Challenge 3 — Acknowledgements & Rate Limiting

Build a `submit-score` event that uses acknowledgements to confirm success or report validation errors. Add server-side rate limiting per socket (`RATE_LIMIT_MAX` events per `RATE_LIMIT_WINDOW` ms) using a `Map` of socket IDs to timestamp arrays. Rate-limited requests return an error ack — the client is *not* disconnected. On success, broadcast a `score-recorded` event to all. Test valid submissions, all validation error cases, the broadcast, rate-limit exhaustion, and per-socket isolation.

View solution

> *An undocumented API forces every consumer to read the source code. OpenAPI 3.0 is a standard format for describing REST APIs — what routes exist, what they accept, what they return, and how to authenticate. swagger-jsdoc reads JSDoc comments in your route files and produces a spec JSON automatically. swagger-ui-express serves an interactive browser UI from that JSON so clients can explore and test the API without touching a terminal.*

OpenAPI vs Swagger

"Swagger" and "OpenAPI" are often used interchangeably, which causes confusion. The history: Swagger was the original spec + tooling project. In 2016 it was donated to the OpenAPI Initiative and renamed **OpenAPI Specification (OAS)**. The Swagger brand now refers only to the tooling (Swagger UI, swagger-jsdoc, Swagger Editor). So:

Term	What it is
`OpenAPI 3.0 / OAS`	The vendor-neutral specification for describing REST APIs (a JSON/YAML document)
`Swagger UI`	A browser app that renders an OpenAPI spec as an interactive docs page
`swagger-jsdoc`	An npm package that extracts `@openapi` JSDoc blocks from source files and produces the spec JSON
`swagger-ui-express`	An npm package that serves Swagger UI at an Express route (e.g. `/api-docs`)



OpenAPI Document Anatomy

Top-level key	Purpose	Required
`openapi`	Version string — always `"3.0.0"` (or `"3.1.0"`)	✓
`info`	API metadata: `title`, `version`, `description`, `contact`	✓
`servers`	Array of base URLs (dev, staging, prod). Swagger UI uses the first one for "Try it out"	
`paths`	Every route, keyed by path (e.g. `/api/products/{id}`). Each path has method objects: `get`, `post`, etc.	✓
`components`	Reusable definitions: `schemas`, `responses`, `parameters`, `securitySchemes`	
`tags`	Named groups with descriptions — routes reference them to appear under a heading in the UI	
`security`	Default security applied to every path (can be overridden per-operation)	

Setup — swagger-jsdoc + swagger-ui-express

```
// npm install swagger-jsdoc swagger-ui-express

const swaggerJsdoc = require('swagger-jsdoc');
const swaggerUi    = require('swagger-ui-express');
const path         = require('path');

const options = {
```

```
  definition: {
    openapi: '3.0.0',
    info: {
      title:       'Products API',
      version:     '1.0.0',
      description: 'CRUD API for the products catalogue',
    },
    servers: [
      { url: 'http://localhost:3000', description: 'Development' },
    ],
    tags: [
      { name: 'Products', description: 'Product catalogue endpoints' },
    ],
    components: {
      schemas: { /* defined here — see Components section */ },
      securitySchemes: {
        BearerAuth: {
          type:         'http',
          scheme:       'bearer',
          bearerFormat: 'JWT',
        },
      },
    },
  },
  // Glob patterns pointing at files with @openapi JSDoc blocks
  apis: [path.join(__dirname, './src/routes/*.js')],
};

const spec = swaggerJsdoc(options);  // generates the OpenAPI object

// Serve the interactive UI
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(spec));

// Expose raw JSON for client generators and tests
app.get('/api-docs.json', (_req, res) => res.json(spec));
```

> **The apis glob is the most common source of bugs**
>
> swagger-jsdoc resolves globs relative to `process.cwd()` (where you run `node` / `jest`), not relative to the file. Using `path.join(__dirname, '...')` makes the path absolute and immune to where you run the process from.

Annotating Routes with @openapi

swagger-jsdoc scans the files in the `apis` glob for JSDoc comment blocks that contain the `@openapi` tag (also accepted: `@swagger`). Everything after the tag is parsed as YAML. The YAML path must use curly-brace params (`{id}`), not the Express colon syntax (`:id`).

```
// src/routes/products.js

const router = require('express').Router();

/**
 * @openapi
 * /api/products:
 *   get:
 *     summary: List all products
 *     tags: [Products]
 *     parameters:
 *       - in: query
 *         name: search
```

```
 *         schema:
 *           type: string
 *         description: Filter by product name
 *       - in: query
 *         name: limit
 *         schema:
 *           type: integer
 *           default: 20
 *         description: Maximum number of results
 *     responses:
 *       200:
 *         description: Array of products
 *         content:
 *           application/json:
 *             schema:
 *               type: array
 *               items:
 *                 $ref: '#/components/schemas/Product'
 */
router.get('/', controller.list);

/**
 * @openapi
 * /api/products/{id}:
 *   get:
 *     summary: Get a single product by ID
 *     tags: [Products]
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *         description: Product ID
 *     responses:
 *       200:
 *         description: A single product
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/Product'
 *       404:
 *         description: Product not found
 *         content:
 *           application/json:
 *             schema:
 *               $ref: '#/components/schemas/ErrorResponse'
 */
router.get('/:id', controller.show);

module.exports = router;
```

> **Express :id → OpenAPI {id}**
>
> Express routes use `/api/products/:id` but OpenAPI paths use `/api/products/{id}`. swagger-jsdoc does **not** convert them — if you write `:id` in the YAML, the UI will show it literally as a colon-param and path parameter "Try it out" will be broken. Always use `{id}` in your `@openapi` blocks.

Parameters

OpenAPI parameters are declared with an `in` field that specifies their location. Each has a name, schema, and optional description and required flag.

`in` value	Location	Required by default?	Example
`path`	`/products/{id}`	Yes — must be `required: true`	`{id}`
`query`	`?search=widget&limit=10`	No	`?search=`
`header`	`Authorization: Bearer ...`	No (use securitySchemes instead)	`X-Request-ID`
`body`	Request body (JSON)	N/A — use `requestBody`, not `parameters`	POST body

Request Body

```
/**
* @openapi
* /api/products:
*   post:
*     summary: Create a product
*     tags: [Products]
*     security:
*       - BearerAuth: []
*     requestBody:
*       required: true
*       content:
*         application/json:
*           schema:
*             $ref: '#/components/schemas/ProductInput'
*     responses:
*       201:
*         description: Product created
*         headers:
*           Location:
*             schema:
*               type: string
*             description: URL of the new resource
*         content:
*           application/json:
*             schema:
*               $ref: '#/components/schemas/Product'
*       422:
*         description: Validation error
*         content:
*           application/json:
*             schema:
*               $ref: '#/components/schemas/ValidationError'
*/
router.post('/', authenticate, controller.create);
```

Components and $ref

Defining a schema inline in every route response leads to duplication and drift. The `components/schemas` section lets you define a shape once and reference it with `$ref: '#/components/schemas/Product'` anywhere in the spec. swagger-jsdoc merges whatever is in `definition.components.schemas` with any schemas defined in `@openapi` blocks.

DEFINE ONCE (IN OPTIONS.DEFINITION)

REFERENCE EVERYWHERE (IN @OPENAPI BLOCKS)

```
components: {
  schemas: {
    Product: {
      type: 'object',
      properties: {
        id:    { type: 'integer' },
        name:  { type: 'string' },
        price: { type: 'number' },
        stock: { type: 'integer' },
      },
    },
    ProductInput: {
      type: 'object',
      required: ['name', 'price'],
      properties: {
        name:  { type: 'string', maxLength: 200 },
        price: { type: 'number', minimum: 0 },
        stock: { type: 'integer', minimum: 0 },
      },
    },
    ErrorResponse: {
      type: 'object',
      properties: {
        error:  { type: 'string' },
        status: { type: 'integer' },
      },
    },
  },
}
```

```
/**
* @openapi
* /api/products/{id}:
*   get:
*     responses:
*       200:
*         content:
*           application/json:
*             schema:
*               $ref: '#/components/schemas/Product'
*       404:
*         content:
*           application/json:
*             schema:
*               $ref: '#/components/schemas/ErrorResponse'
*   patch:
*     requestBody:
*       content:
*         application/json:
*           schema:
*             $ref: '#/components/schemas/ProductInput'
*     responses:
*       200:
*         content:
*           application/json:
*             schema:
*               $ref: '#/components/schemas/Product'
*/
```

Security Schemes

Security schemes are declared once in `components/securitySchemes` and then applied to individual operations with a `security` array. An empty array (`- BearerAuth: []`) means the scheme is required but no specific scopes are needed.

```
// In options.definition:
components: {
  securitySchemes: {
    BearerAuth: {
      type:         'http',
      scheme:       'bearer',
      bearerFormat: 'JWT',
    },
    ApiKeyAuth: {
      type: 'apiKey',
      in:   'header',
      name: 'X-API-Key',
    },
  },
}

// In a route's @openapi block:
/**
*   delete:
*     summary: Delete a product
*     security:
*       - BearerAuth: []    ← name must match the key in securitySchemes
*     responses:
*       204:
```

```
 *         description: Product deleted
 *       401:
 *         description: Unauthorized
 */
```

> **Global vs per-operation security**
>
> Setting `security: [{ BearerAuth: [] }]` at the document root applies to all operations. Individual operations can override this — including opting out with `security: []` (empty array = no security required). Most APIs set no global security and mark individual routes explicitly so it's obvious which routes are public.

Testing the Spec

Because `swaggerJsdoc(options)` returns a plain JavaScript object, you can test the spec's structure directly without HTTP — fast, no server needed. Test coverage goals: every path exists, every method has at least one response, all `$ref`s point to defined schemas, security is wired to protected routes.

```
// openapi.test.js
const spec = require('./src/openapi');  // exports swaggerJsdoc(options)

describe('OpenAPI spec structure', () => {
  test('is a valid OpenAPI 3.0 document', () => {
    expect(spec.openapi).toMatch(/^3\./);
    expect(spec.info.title).toBeTruthy();
    expect(spec.info.version).toBeTruthy();
  });

  test('contains all expected product paths', () => {
    expect(spec.paths).toHaveProperty('/api/products');
    expect(spec.paths).toHaveProperty('/api/products/{id}');
  });

  test('every operation has at least one response', () => {
    for (const [, methods] of Object.entries(spec.paths)) {
      for (const [method, op] of Object.entries(methods)) {
        if (method === 'parameters') continue;  // path-level params array
        expect(Object.keys(op.responses).length).toBeGreaterThan(0);
      }
    }
  });

  test('all $refs point to defined schemas', () => {
    const specStr = JSON.stringify(spec);
    const refRe   = /#\/components\/schemas\/(\w+)/g;
    let match;
    while ((match = refRe.exec(specStr)) !== null) {
      expect(spec.components?.schemas).toHaveProperty(match[1]);
    }
  });

  test('BearerAuth scheme is defined', () => {
    expect(spec.components.securitySchemes).toHaveProperty('BearerAuth');
  });

  test('POST /api/products requires BearerAuth', () => {
    const postOp   = spec.paths['/api/products'].post;
    const security = postOp.security ?? spec.security ?? [];
    expect(security.some((s) => 'BearerAuth' in s)).toBe(true);
```

```
  });
});
```

Also test via HTTP

For a full integration check, use supertest to call `GET /api-docs.json` and assert `res.status === 200` and `res.body.openapi` is set. This catches config wiring errors (wrong glob path, missing `app.use`) that the unit-style spec tests above can't see.

Coding Challenges

Challenge 1 — Basic Setup

Wire up `swagger-jsdoc` and `swagger-ui-express` for a products CRUD API (GET /api/products, GET /api/products/{id}, POST, PATCH /api/products/{id}, DELETE). Annotate all five routes with `@openapi` blocks — each route needs at least two response codes. Expose `/api-docs` (UI) and `/api-docs.json` (raw spec). Write tests that verify all five paths exist, each operation has responses, and the correct HTTP status codes are present.

View solution

Challenge 2 — Schemas & $ref

Define four reusable schemas in `components/schemas`: `Product`, `ProductInput`, `ErrorResponse`, and `ValidationError` (which has a `details` array of `{field, message}` objects). Update all route annotations to use `$ref` instead of inline schemas. Write a test that scans the entire spec for `$ref` strings, extracts the schema name from each, and asserts it exists in `components/schemas` — so a broken reference is caught automatically.

View solution

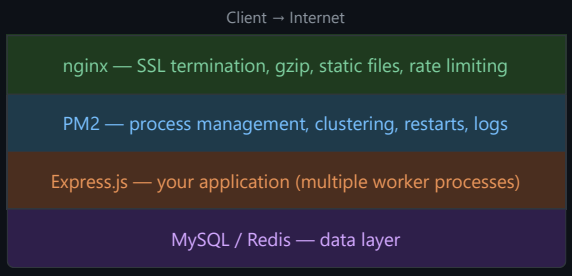
Challenge 3 — Security & Full Validation

Add a `BearerAuth` security scheme and mark POST, PATCH, and DELETE as requiring it. Add query parameters to `GET /api/products` (`search`, `limit`, `offset`) and a path parameter to `GET /api/products/{id}`. Write a comprehensive spec test suite: openapi version check, all paths present, every operation has responses, all `$ref`s resolve, protected routes have `BearerAuth` in their security array, query params are defined on GET /products, path param is defined on /products/{id}. End with an HTTP test via supertest.

View solution

> *Running `node app.js` is a development convenience — not a production deployment. A real deployment needs: environment secrets outside the codebase, a process manager that restarts the app on crash, a reverse proxy that terminates SSL and routes traffic, and ideally a container so the app runs identically everywhere. This chapter covers all four layers.*

The Production Stack



Each layer has a clear responsibility. nginx never runs application code; PM2 never touches HTTP; Express never manages its own restart. Keeping concerns separated means each piece can be scaled, replaced, or debugged independently.

Environment Management

Secrets and environment-specific config (database credentials, JWT secrets, API keys) must never be committed to source control. The standard pattern: store them in a `.env` file (gitignored), load them with `dotenv` in development, and inject them via the host environment or a secrets manager in production.

```
# .env  (gitignored — NEVER commit this)
NODE_ENV=development
PORT=3000
DB_HOST=localhost
DB_USER=myapp
DB_PASS=supersecret
DB_NAME=myapp_dev
JWT_SECRET=dev-secret-replace-in-prod


// src/config/env.js — validate at startup, fail loudly if anything is missing
const REQUIRED = [
  'NODE_ENV', 'PORT',
  'DB_HOST', 'DB_USER', 'DB_PASS', 'DB_NAME',
  'JWT_SECRET',
];

const missing = REQUIRED.filter((key) => !process.env[key]);
if (missing.length) {
  throw new Error(
    `Missing required environment variables: ${missing.join(', ')}\n` +
    `Add them to your .env file or the host environment.`
  );
}

module.exports = {
  nodeEnv:   process.env.NODE_ENV,
  port:      Number(process.env.PORT),
  db: {
```

```
    host:     process.env.DB_HOST,
    user:     process.env.DB_USER,
    password: process.env.DB_PASS,
    database: process.env.DB_NAME,
  },
  jwtSecret: process.env.JWT_SECRET,
};
```

Require `dotenv` at the very top of your entry point (before any other `require`) so the values are in `process.env` when other modules load. In production, don't use dotenv at all — set the variables directly in the host environment or via a secrets manager.

```
// src/server.js — entry point
if (process.env.NODE_ENV !== 'production') {
  require('dotenv').config();
}

const config = require('./config/env');  // throws if anything is missing
const app    = require('./app');

app.listen(config.port, () => {
  console.log(`Listening on :${config.port} [${config.nodeEnv}]`);
});
```

> **Validate at startup, not at request time**
>
> If a required secret is missing and you only discover it when a request hits the route that needs it, the app has been silently broken for every request before that. Validating all required variables during startup makes the failure immediate, obvious, and impossible to miss.

PM2 — Process Manager

PM2 keeps your app alive. If the process crashes, PM2 restarts it. In cluster mode it forks multiple worker processes — one per CPU core — so a single machine can handle load proportional to its CPU count, and a crash in one worker doesn't take down the others.

Command	What it does
`pm2 start ecosystem.config.js --env production`	Start using the production env block
`pm2 stop my-api`	Stop (keeps in process list)
`pm2 restart my-api`	Graceful restart (zero-downtime with cluster)
`pm2 reload my-api`	Rolling restart — each worker restarts one at a time
`pm2 delete my-api`	Remove from process list
`pm2 logs my-api`	Tail logs
`pm2 monit`	Terminal dashboard (CPU, RAM, restarts)
`pm2 startup`	Generate systemd/launchd script to auto-start on boot
`pm2 save`	Persist current process list across reboots

```
// ecosystem.config.js
module.exports = {
```

```
  apps: [{
    name:         'my-api',
    script:       'src/server.js',
    instances:    'max',       // one worker per CPU core
    exec_mode:    'cluster',   // share the port across all workers
    watch:        false,       // true in dev only — restarts on file change
    max_memory_restart: '500M',  // kill and restart if worker leaks memory
    merge_logs:   true,        // single log stream across workers
    log_date_format: 'YYYY-MM-DD HH:mm:ss Z',
    error_file:   'logs/err.log',
    out_file:     'logs/out.log',

    env: {                     // development defaults
      NODE_ENV: 'development',
    },
    env_production: {          // used with --env production flag
      NODE_ENV: 'production',
      PORT:     3000,
    },
  }],
};
```

Cluster mode and session state

In cluster mode, each request may land on a different worker. If your sessions or rate limiter state lives in process memory (a plain JS `Map`), workers won't share it — worker A's rate limit counter is invisible to worker B. Fix: move shared state to Redis (use `ioredis` as the store for `express-rate-limit`, Socket.IO's adapter, and sessions).

nginx as Reverse Proxy

nginx sits in front of Node, handling what Node shouldn't have to: SSL/TLS termination, gzip compression, static file serving, connection buffering, and coarse rate limiting. Node only sees decrypted HTTP on localhost — simpler, faster, and more secure.

```
upstream node_app {
    server    127.0.0.1:3000;
    keepalive 64;       # reuse connections to Node — avoids TCP handshake per request
}

# HTTP → HTTPS redirect
server {
    listen      80;
    server_name example.com;
    return      301 https://$host$request_uri;
}

server {
    listen      443 ssl http2;
    server_name example.com;

    # SSL — managed by Certbot / Let's Encrypt
    ssl_certificate     /etc/letsencrypt/live/example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/example.com/privkey.pem;
    ssl_protocols       TLSv1.2 TLSv1.3;
    ssl_ciphers         ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256;

    # Compression
    gzip       on;
    gzip_types application/json text/plain application/javascript text/css;

    # Security headers
```

```
    add_header X-Frame-Options         DENY;
    add_header X-Content-Type-Options   nosniff;
    add_header Referrer-Policy          strict-origin-when-cross-origin;
    add_header Strict-Transport-Security "max-age=63072000" always;

    # Connection-level rate limit (nginx zone; defined in http block)
    limit_req  zone=api burst=20 nodelay;

    location / {
        proxy_pass         http://node_app;
        proxy_http_version 1.1;

        # Required for WebSocket upgrade
        proxy_set_header Upgrade           $http_upgrade;
        proxy_set_header Connection        'upgrade';

        # Forward real client info to Node
        proxy_set_header Host              $host;
        proxy_set_header X-Real-IP         $remote_addr;
        proxy_set_header X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_cache_bypass                  $http_upgrade;
    }
}
```

With nginx forwarding `X-Forwarded-For`, Express must be told to trust the proxy so that `req.ip` returns the real client IP rather than nginx's loopback address:

```
// app.js — trust first proxy (nginx)
app.set('trust proxy', 1);
```

> **Never set trust proxy without nginx**
>
> `app.set('trust proxy', 1)` tells Express to read the real IP from `X-Forwarded-For`. Without a reverse proxy, any client can spoof that header to bypass IP-based rate limiting. Only set this when nginx (or another trusted proxy) is actually in front.

Docker

Docker packages the app and its exact runtime (Node version, npm dependencies) into an image that runs identically on any machine. A multi-stage build keeps the final image small: the builder stage installs all dependencies, then the production stage copies only what's needed to run the app.

```
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
# --omit=dev skips devDependencies; ci is deterministic (reads package-lock.json)
RUN  npm ci --omit=dev


FROM node:20-alpine AS production
ENV  NODE_ENV=production
WORKDIR /app

# Copy only the production node_modules from the builder stage
COPY --from=builder /app/node_modules ./node_modules
# Copy the application source
COPY . .
```

```
# Never run as root inside a container
RUN  addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser

EXPOSE 3000

# Container-level health check — Docker orchestrators use this
HEALTHCHECK --interval=30s --timeout=5s --retries=3 \
  CMD wget -qO- http://localhost:3000/health || exit 1

CMD ["node", "src/server.js"]
```

Instruction	Key point
`FROM … AS builder`	Named stage — referenced by `COPY --from=builder` in the next stage
`npm ci`	Deterministic install from `package-lock.json`; fails if it doesn't match
`--omit=dev`	Skips devDependencies (jest, supertest, etc.) — smaller image
`COPY . .`	Must come *after* the node_modules copy, not before
`USER appuser`	Drop root privileges; container escapes are much less dangerous
`EXPOSE`	Documentation only — does not actually publish the port
`CMD` vs `ENTRYPOINT`	`CMD` is the default command, easily overridden. `ENTRYPOINT` is the fixed executable. For Node apps, `CMD ["node", "src/server.js"]` is the standard choice.

The `.dockerignore` file prevents large or sensitive directories from being sent to the build context:

```
node_modules
.env
.env.*
logs
*.log
.git
coverage
.nyc_output
*.md
```

docker-compose — App + Database Together

```
services:
  app:
    build: .
    ports:
      - "3000:3000"
    env_file: .env.docker        # separate env file for Docker env
    environment:
      DB_HOST: db                # service name = hostname inside Docker network
    depends_on:
      db:
        condition: service_healthy  # wait until MySQL passes its health check
    restart: unless-stopped

  db:
    image: mysql:8.0
    env_file: .env.docker
    volumes:
```

```
      - db_data:/var/lib/mysql   # named volume — data survives container restart
    healthcheck:
      test:     ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout:  5s
      retries:  5
    restart: unless-stopped

volumes:
  db_data:
```

DB_HOST is the service name, not localhost

Inside a Docker Compose network, containers reach each other by service name. If your database service is named `db`, the app must connect to host `db`, not `localhost`. This is set in the `environment` block above and overrides whatever is in the env file.

Health Check Endpoint

A health check route lets Docker, load balancers, and monitoring systems verify the app is not just running but actually working — database connected, dependencies reachable. Returning 200 means healthy; 503 means the app is up but not ready to serve traffic.

```
// src/routes/health.js
const router = require('express').Router();
const pool   = require('../db/pool');

router.get('/', async (_req, res) => {
  try {
    await pool.execute('SELECT 1');
    res.json({ status: 'ok', db: 'ok', uptime: process.uptime() });
  } catch {
    res.status(503).json({ status: 'error', db: 'unreachable' });
  }
});

module.exports = router;


// app.js — register before auth middleware (health checks should always respond)
app.use('/health', require('./routes/health'));
```

Liveness vs Readiness

Kubernetes distinguishes two types of health checks. A *liveness* probe checks whether the process is still alive (if it fails, restart the container). A *readiness* probe checks whether it's ready to receive traffic (if it fails, stop routing to this pod). In Express you can implement both on separate paths — `/health/live` (just `res.json({ ok: true })`) and `/health/ready` (includes DB check).

Coding Challenges

Challenge 1 — Environment Validation & PM2

Challenge 2 — nginx Config & Trust Proxy

Challenge 3 — Dockerfile, docker-compose & Health Check

Build a `src/config/env.js` module that validates all required environment variables on startup, throws a descriptive error listing every missing key, and exports a typed config object. Write Jest tests: all vars present returns the config; each missing var produces an error message naming it; the error lists multiple missing vars at once; the exported types are correct (port is a number, not a string). Also write a complete `ecosystem.config.js` with cluster mode, memory restart threshold, and production env block.

View solution

Write a production-ready `nginx.conf`: HTTP → HTTPS redirect, SSL placeholders, upstream block, `proxy_pass`, all required proxy headers, gzip, security headers, and a rate limit zone. In Express, implement a `src/middleware/realIp.js` module that reads the real client IP from `X-Forwarded-For` when `app.set('trust proxy', 1)` is active. Write tests: correct IP extracted from forwarded header, falls back to `req.socket.remoteAddress` when header absent, and a config file test that reads `nginx.conf` and asserts required directives are present.

View solution

Write a multi-stage `Dockerfile` (builder + production), a `.dockerignore`, and a `docker-compose.yml` with app + MySQL + named volume + `depends_on: condition: service_healthy`. Implement `GET /health` that returns `{status:'ok', db:'ok', uptime}` on 200 when the DB query succeeds, and `{status:'error', db:'unreachable'}` on 503 when it fails. Write supertest tests for both states using a mocked pool, plus a test that reads the Dockerfile and asserts the non-root USER instruction is present.

View solution