Node.js Backend Best Practices

Production-Ready Patterns for 2026

JANUARY 30, 2026

THEDECIPHERIST.COM

TABLE OF CONTENTS

2026 EDITION

Based on 10 Years of Production Experience

Workers, Servers, APIs, MongoDB, Docker, Secrets Management

TABLE OF CONTENTS

1. INTRODUCTION AND PHILOSOPHY

1.1 Why This Guide Exists

This guide distills 10 years of production Node.js experience into actionable patterns. It focuses exclusively on backend development: workers, servers, APIs, database management, and containerized deployments.

1.2 Core Principles

- **Async/Await Everywhere**: Callback hell is behind us. Modern Node.js requires async/await as the default.
- **Graceful Degradation**: Every process must handle shutdown signals properly.
- **Abstraction Layers**: Wrap third-party dependencies (especially databases) to isolate breaking changes.
- **Structured Logging**: Every log entry should be machine-parseable with context.
- **Stateless Design**: Design for horizontal scaling from day one.
- **Lock Your Dependencies**: Use package-lock.json and npm ci in production.

1.3 Node.js Version Strategy

Always use LTS (Long Term Support) releases in production. As of 2026, Node.js 22.x LTS is the recommended version. Never use odd-numbered releases (23.x, 25.x) in production—these are experimental.

```
# Lock your Node version with .nvmrc
echo "22.12.0" > .nvmrc
nvm use

# Or use Volta for version management
volta pin node@22
```

---

2. PROJECT STRUCTURE AND ARCHITECTURE

2.1 Recommended Directory Structure

```
project-root/
├── globals/                    # Shared modules across projects
│   └── server_modules/
│       ├── db.js              # Database wrapper (singleton)
│       ├── log.js             # Structured logger
│       ├── worker.js          # Worker thread abstraction
│       ├── module_loader.js   # Dynamic module loading
│       ├── middleware/        # Express middleware
│       │   ├── auth.js
│       │   ├── routeLogger.js
│       │   └── hasValidBrand.js
│       ├── utils/             # Utility functions
│       └── tasks/             # Background task utilities
├── app/
│   └── _webserver/
│       ├── server.js          # Main Express server
│       ├── routes/            # API route handlers
│       │   └── __template__.js
│       ├── tasks/             # Cron-style background jobs
│       ├── watchers/          # MongoDB change stream watchers
│       ├── workers/           # CPU-intensive worker scripts
│       └── modules/           # App-specific modules
├── secrets/                   # Credentials (NEVER in git)
│   ├── dbconn.js
│   ├── sessions.js
│   └── aws-config.json
├── documentation/
├── docker-compose.yaml
├── Dockerfile
└── package.json
```

2.2 Module Loading Pattern

Use a centralized module loader to dynamically load all modules from a directory. This enables plugin-like architecture and reduces boilerplate.

```
// globals/server_modules/module_loader.js
var path = require('path');
var files = require(__dirname + "/files");

module.exports = function module_loader(directory, filesArray) {
    if (!directory || directory == "") {
        console.error("Fatal error: no directory given to module_loader");
        process.exit(1);
    }

    var dir = path.join(__dirname, "/", directory);
    var modules = files.getFilesSync(dir);

    if (modules.length == 0) {
        console.error("Error: module_loader found no modules in '" + dir + "'")
        process.exit(1);
    }

    var filesToLoad = filesArray && Array.isArray(filesArray) ? filesArray : fa
    var functions = {};

    modules.forEach(function(file) {
        if (!filesToLoad || filesToLoad.indexOf(file.name) !== -1) {
            functions[file.name] = require(file.relativePath);
        }
    });

    return functions;
};

// Usage
var middleware = require("/globals/server_modules/module_loader")("middleware")
// middleware.auth, middleware.routeLogger, etc. are now available
```

---

3. MONGODB NATIVE DRIVER PATTERNS

3.1 The Database Wrapper Philosophy

NEVER use the MongoDB driver directly throughout your application. Create a single abstraction layer (db.js) that wraps all MongoDB operations. This single decision will save you hundreds of hours during driver upgrades.

Why: Between MongoDB driver versions 3.x, 4.x, 5.x, and 6.x, nearly every API changed. Connection strings, method names, option names, and return values all shifted. A wrapper lets you update ONE file instead of hunting through your entire codebase.

3.2 Complete Database Wrapper Implementation

```
// globals/server_modules/db.js
var MongoClient = require('mongodb').MongoClient;
var ObjectID = require('mongodb').ObjectID;
var EventEmitter = require('events').EventEmitter;
var logger = require('./log');
var dbconn = require('/secrets/dbconn');

var mongoDbUrl = dbconn.getUrl();
var isProd = process.env.NODE_ENV === "production";
var dbClient;
var dbPool;
var dbEvents = new EventEmitter();

// Singleton connection pattern
async function db() {
    if (dbPool) {
        return dbPool;
    }

    try {
        dbClient = await MongoClient.connect(mongoDbUrl, {
            maxPoolSize: 5,  // Was 'poolSize' in v3.x
            // For MongoDB 6.x+:
            // serverApi: { version: '1', strict: true, deprecationErrors: true
        });

        dbPool = dbClient.db();

        // Monitor topology events
        dbClient.on('serverDescriptionChanged', (event) => {
            logger.prod().debug("MongoDB topology change", { event });
        });

        dbClient.on('close', () => {
            logger.prod().warn("MongoDB connection closed");
            dbEvents.emit("closed");
        });

        dbEvents.emit("connected");
        return dbPool;

    } catch (err) {
        logger.prod().error("MongoDB connection failed", { error: err });
        throw err;
    }
}
```

```
// Collection helper with aggregation pipeline
async function getCollection(collectionName, conditions = []) {
    var database = await db();
    var collection = database.collection(collectionName);

    if (conditions.length > 0) {
        return await collection.aggregate(conditions).toArray();
    }
    return collection;
}

// CRUD Operations with error handling
async function insertOne(collectionName, document) {
    var database = await db();
    var collection = database.collection(collectionName);
    return await collection.insertOne(document);
}

async function updateOne(collectionName, filter, update, upsert = false) {
    var database = await db();
    var collection = database.collection(collectionName);
    var updateDoc = update["$set"] ? update : { "$set": update };
    return await collection.updateOne(filter, updateDoc, { upsert });
}

async function updateMany(collectionName, filter, update, upsert = false) {
    var database = await db();
    var collection = database.collection(collectionName);
    var updateDoc = update["$set"] ? update : { "$set": update };
    return await collection.updateMany(filter, updateDoc, { upsert });
}

async function deleteOne(collectionName, filter) {
    var database = await db();
    var collection = database.collection(collectionName);
    return await collection.deleteOne(filter);
}

async function bulkWrite(collectionName, operations) {
    var database = await db();
    var collection = database.collection(collectionName);
    return await collection.bulkWrite(operations, { ordered: false });
}

// Change Streams for real-time watching
async function watchCollection(collectionName, pipeline = [], options = {}) {
```

```
    var database = await db();
    var collection = database.collection(collectionName);
    return collection.watch(pipeline, {
        fullDocument: 'updateLookup',
        ...options
    });
}

// Utility functions
function isRealMongoObjectId(id) {
    if (!id) return false;
    return ObjectID.isValid(id) && String(new ObjectID(id)) === String(id);
}

function safeString(str, maxLength = 1000) {
    if (!str) return '';
    return String(str).substring(0, maxLength).replace(/[<>]/g, '');
}

function safeInt(val, defaultVal = 0) {
    var parsed = parseInt(val, 10);
    return isNaN(parsed) ? defaultVal : parsed;
}

// Graceful shutdown
function cleanupDB() {
    if (dbClient) {
        dbClient.close();
        dbPool = null;
    }
}

// Export everything
module.exports = {
    db, getCollection, insertOne, updateOne, updateMany,
    deleteOne, bulkWrite, watchCollection, cleanupDB,
    ObjectID, isRealMongoObjectId, safeString, safeInt,
    events: dbEvents
};
```

3.3 MongoDB 6.x Migration Notes

When upgrading from MongoDB driver 3.x to 6.x, these are the critical changes:

OLD (3.X)	NEW (6.X)
`poolSize`	`maxPoolSize`
`useNewUrlParser: true`	Remove (now default)
`useUnifiedTopology: true`	Remove (now default)
`collection.insert()`	`collection.insertOne()` or `insertMany()`
`collection.remove()`	`collection.deleteOne()` or `deleteMany()`
`collection.update()`	`collection.updateOne()` or `updateMany()`
Callbacks	All methods return Promises natively

---

4. ERROR HANDLING AND PROCESS MANAGEMENT

4.1 The Golden Rule

EVERY Node.js process must handle these signals: exit, SIGINT, SIGTERM, uncaughtException, unhandledRejection. Failure to do so causes zombie processes, connection leaks, and data corruption.

4.2 Standard Cleanup Pattern

```
// Always use this pattern in every script
var db = require('./db');
var logger = require('./log');

async function cleanup(signal) {
    logger.prod().info("Cleanup initiated", { signal });

    try {
        // Close database connections
        db.cleanupDB();

        // Close any other resources (Redis, message queues, etc.)
        // await redis.quit();
        // await messageQueue.close();

        logger.prod().info("Cleanup completed", { signal });
    } catch (err) {
        logger.prod().error("Cleanup error", { error: err, signal });
    }

    process.exit(signal === 'uncaughtException' ? 1 : 0);
}

// Handle ALL shutdown signals
["exit", "SIGINT", "SIGTERM", "SIGUSR1", "SIGUSR2"].forEach(signal => {
    process.on(signal, () => cleanup(signal));
});

// Critical: Handle unhandled errors
process.on("uncaughtException", (err) => {
    logger.prod().error("Uncaught Exception", { error: err });
    cleanup("uncaughtException");
});

process.on("unhandledRejection", (reason, promise) => {
    logger.prod().error("Unhandled Rejection", { reason, promise });
    cleanup("unhandledRejection");
});
```

4.3 Database-Based Locking for Long-Running Tasks

For tasks that should never run concurrently (importers, batch processors), use database-based locking instead of file locks.

```
// Database lock pattern for critical sections
async function acquireLock(lockName) {
    var existing = await db.getCollection("__system_locks", [
        { $match: { [lockName]: true } },
        { $limit: 1 }
    ]);

    if (existing.length > 0) {
        throw new Error(`Lock '${lockName}' already held`);
    }

    await db.insertOne("__system_locks", {
        [lockName]: true,
        createdDate: new Date(),
        pid: process.pid,
        hostname: require('os').hostname()
    });
}

async function releaseLock(lockName) {
    await db.deleteOne("__system_locks", { [lockName]: true });
}

// Usage in a task
async function runImporter() {
    try {
        await acquireLock("x3OrderImporter");

        // Do the actual work
        await processOrders();

    } finally {
        await releaseLock("x3OrderImporter");
    }
}
```

---

5. LOGGING AND OBSERVABILITY

5.1 Structured Logging with Winston

Console.log is for development. Production requires structured, machine-parseable logs with context enrichment for tools like DataDog, Splunk, or ELK.

```
// globals/server_modules/log.js
var winston = require('winston');
var os = require('os');

var isProd = process.env.NODE_ENV === "production";

// Custom format for production
var prodFormat = winston.format.combine(
    winston.format.timestamp(),
    winston.format.json()
);

// Human-readable format for development
var devFormat = winston.format.combine(
    winston.format.colorize(),
    winston.format.timestamp({ format: 'HH:mm:ss' }),
    winston.format.printf(({ level, message, timestamp, ...meta }) => {
        return `${timestamp} [${level}]: ${message} ${Object.keys(meta).length
    })
);

var logger = winston.createLogger({
    level: isProd ? 'info' : 'debug',
    format: isProd ? prodFormat : devFormat,
    defaultMeta: {
        service: process.env.DD_SERVICE || 'nodeserver',
        env: process.env.DD_ENV || process.env.NODE_ENV || 'development',
        version: process.env.DD_VERSION || 'unknown',
        hostname: os.hostname()
    },
    transports: [
        new winston.transports.Console()
    ]
});

// HTTP status code to log level mapping
function getLogLevel(statusCode) {
    if (statusCode >= 500) return 'error';
    if (statusCode >= 400) return 'warn';
    if (statusCode >= 300) return 'notice';
    return 'info';
}

// Request context extractor
function extractRequestContext(req) {
    return {
```

```
        method: req.method,
        url: req.originalUrl || req.url,
        ip: req.headers['x-forwarded-for'] || req.connection?.remoteAddress,
        userAgent: req.headers['user-agent'],
        referer: req.headers['referer'],
        userId: req.user?.id,
        sessionId: req.session?.id
    };
}

module.exports = {
    dev: () => logger,
    prod: () => logger,
    getLogLevel,
    extractRequestContext
};
```

5.2 DataDog APM Integration

DataDog provides distributed tracing, APM, and log aggregation. Initialize it FIRST, before any other imports.

```
// MUST be at the very top of server.js
if (process.env.NODE_ENV !== "development") {
    const tracer = require('dd-trace').init({
        profiling: true,
        runtimeMetrics: true,
        logInjection: true,          // Correlate logs with traces
        env: process.env.DD_ENV || 'production',
        service: process.env.DD_SERVICE || 'nodeserver',
        version: process.env.DD_VERSION || '1.0.0',
        hostname: process.env.DD_AGENT_HOST || 'localhost',
        port: process.env.DD_TRACE_AGENT_PORT || 8126
    });
}

// Now import everything else
var express = require('express');
var logger = require('./log');
// ...
```

5.3 Event-Based Logging Pattern

Use consistent event structures for searchability and dashboards:

```
// Standard event structure
logger.prod().info("Survey completed", {
    survey: surveyObj,
    evt: {
        tags: ["survey", "customer-feedback"],
        category: "survey",
        name: "surveyCompleted",
        outcome: "success"
    },
    usr: {
        email: surveyObj.email,
        caseId: surveyObj.caseId,
        customerId: surveyObj.customerId
    }
});

// Error logging with stack traces
try {
    await riskyOperation();
} catch (err) {
    logger.prod().error("Operation failed", {
        error: {
            message: err.message,
            stack: err.stack,
            code: err.code
        },
        evt: {
            tags: ["error", "critical"],
            category: "system",
            name: "operationFailed",
            outcome: "failure"
        },
        context: { orderId, userId }
    });
}
```

---

6. WORKER THREADS AND BACKGROUND TASKS

6.1 Worker Thread Abstraction

Use Node's native worker_threads for CPU-intensive operations. Never block the main event loop with heavy computation.

```
// globals/server_modules/worker.js
const { Worker } = require('worker_threads');
const logger = require('./log');

function runWorker(file, workerData) {
    return new Promise((resolve, reject) => {
        const start = process.hrtime.bigint();

        logger.prod().info("Worker spawning", { file });

        const worker = new Worker(file, {
            workerData,
            resourceLimits: {
                maxOldGenerationSizeMb: 512  // Memory limit per worker
            }
        });

        worker.on("message", (msg) => {
            const duration = Number(process.hrtime.bigint() - start) / 1e6;
            logger.prod().info("Worker completed", {
                file,
                durationMs: duration.toFixed(2)
            });
            resolve(msg);
        });

        worker.on("error", (err) => {
            logger.prod().error("Worker error", { file, error: err });
            reject(err);
        });

        worker.on("exit", (code) => {
            if (code !== 0) {
                reject(new Error(`Worker exited with code ${code}`));
            }
        });
    });
}

module.exports = runWorker;
```

6.2 Worker Script Template

```
// workers/processOrders.js
const { workerData, parentPort } = require('worker_threads');
const db = require('/globals/server_modules/db');
const logger = require('/globals/server_modules/log');

async function processOrders() {
    const { batchId, orderIds } = workerData;

    logger.prod().info("Worker started", { batchId, orderCount: orderIds.length

    let processed = 0;
    let errors = [];

    for (const orderId of orderIds) {
        try {
            await processOrder(orderId);
            processed++;
        } catch (err) {
            errors.push({ orderId, error: err.message });
            logger.prod().error("Order processing failed", { orderId, error: er
        }
    }

    return { batchId, processed, errors };
}

processOrders()
    .then(result => parentPort.postMessage(result))
    .catch(err => {
        logger.prod().error("Worker crashed", { error: err });
        process.exit(1);
    });
```

6.3 MongoDB Change Stream Watchers

For real-time event processing, use MongoDB change streams instead of polling.

```
// watchers/watchSurveys.js
var db = require('/globals/server_modules/db');
var logger = require('/globals/server_modules/log');

var changeStream;

async function startWatcher() {
    logger.prod().info("Survey watcher starting");

    changeStream = await db.watchCollection("surveys", [], {
        fullDocument: 'updateLookup'
    });

    changeStream.on("change", async (change) => {
        const doc = change.fullDocument;

        if (doc.surveyCompleted && !doc.processed) {
            try {
                await processSurvey(doc);
                await db.updateOne("surveys",
                    { _id: doc._id },
                    { processed: true, processedDate: new Date() }
                );
            } catch (err) {
                logger.prod().error("Survey processing failed", {
                    surveyId: doc._id,
                    error: err
                });
            }
        }
    });

    changeStream.on("error", async (err) => {
        logger.prod().error("Change stream error", { error: err });
        // Restart the watcher after a delay
        setTimeout(startWatcher, 5000);
    });
}

// Graceful shutdown
async function cleanup() {
    if (changeStream) {
        await changeStream.close();
    }
    db.cleanupDB();
    process.exit(0);
```

```
}

["SIGINT", "SIGTERM", "uncaughtException", "unhandledRejection"].forEach(signal
    process.on(signal, cleanup);
});

startWatcher();
```

---

7. EXPRESS SERVER PATTERNS

7.1 Server Initialization

```
// server.js
// DataDog MUST be first
if (process.env.NODE_ENV !== "development") {
    require('dd-trace').init({
        profiling: true,
        runtimeMetrics: true,
        env: 'production',
        service: 'nodeserver',
        version: '1.0.0'
    });
}

var logger = require('/globals/server_modules/logger');
logger("Server starting", "system");

var express = require("express");
var session = require("express-session");
var MongoStore = require("connect-mongo");
var cors = require('cors');
var helmet = require('helmet');

var dbconn = require('/secrets/dbconn');
var sessionKeys = require('/secrets/sessions');

var app = express();
var port = process.env.PORT || 3030;

// Security middleware
app.use(helmet());
app.use(cors({
    origin: process.env.NODE_ENV === 'development' ? '*' : 'https://yourdomain.
    credentials: true
}));

// Session configuration
var sessionStore = MongoStore.create({
    mongoUrl: dbconn.getUrl(),
    touchAfter: 24 * 3600,  // Lazy session updates
    crypto: { secret: sessionKeys.getSecret() }
});

app.use(session({
    store: sessionStore,
    secret: sessionKeys.getSecret(),
    name: 'sessionId',
    resave: false,
```

```
    saveUninitialized: false,
    cookie: {
        secure: process.env.NODE_ENV === 'production',
        httpOnly: true,
        maxAge: 24 * 60 * 60 * 1000  // 24 hours
    }
}));

// Body parsing
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Load routes dynamically
var fs = require('fs');
var path = require('path');
var routesPath = path.join(__dirname, 'routes');

fs.readdirSync(routesPath).forEach(file => {
    if (file.endsWith('.js') && !file.startsWith('__')) {
        require(path.join(routesPath, file))(app);
    }
});

// Start server
var server = app.listen(port, () => {
    logger(`Server listening on port ${port}`, "system");
});

// Graceful shutdown
process.on('SIGTERM', () => {
    logger('SIGTERM received, shutting down gracefully', 'system');
    server.close(() => {
        sessionStore.close();
        process.exit(0);
    });
});
```

7.2 Route Template

```
// routes/__template__.js
var db = require("/globals/server_modules/db");
var middleware = require("/globals/server_modules/module_loader")("middleware")

module.exports = function registerRoutes(app) {

    // Public endpoint
    app.get('/api/health', (req, res) => {
        res.json({ status: 'healthy', timestamp: new Date() });
    });

    // Protected endpoint with auth middleware
    app.post('/api/getData', middleware.auth, async (req, res) => {
        try {
            const { filter, limit = 100 } = req.body;

            const data = await db.getCollection("myCollection", [
                { $match: filter || {} },
                { $limit: limit }
            ]);

            res.json({ success: true, data });

        } catch (err) {
            logger.prod().error("getData failed", { error: err, user: req.user
            res.status(500).json({ success: false, error: "Internal error" });
        }
    });

    // Endpoint with validation
    app.post('/api/createItem',
        middleware.auth,
        middleware.hasValidBrand,
        async (req, res) => {
            // Implementation
        }
    );
};
```

---

8. MIDDLEWARE ARCHITECTURE

8.1 Authentication Middleware

```
// middleware/auth.js
var db = require("/globals/server_modules/db");
var logger = require('/globals/server_modules/log');

module.exports = function auth(req, res, next) {
    if (!req.isAuthenticated || !req.isAuthenticated()) {
        logger.prod().warn("Auth failed", {
            url: req.originalUrl,
            ip: req.headers['x-forwarded-for'] || req.ip,
            userAgent: req.headers['user-agent'],
            evt: {
                category: "authentication",
                name: "authFailed",
                outcome: "failure"
            }
        });

        // Log failed auth attempts for security monitoring
        db.insertOne("auth_failures", {
            date: new Date(),
            url: db.safeString(req.url),
            ip: db.safeString(req.headers['x-forwarded-for'] || req.ip),
            userAgent: db.safeString(req.headers['user-agent'])
        }).catch(err => logger.prod().error("Auth logging failed", { error: err

        return res.sendStatus(401);
    }

    logger.prod().debug("Auth success", {
        userId: req.user?.id,
        evt: { category: "authentication", outcome: "success" }
    });

    next();
};
```

8.2 Rate Limiting Middleware

```
// middleware/rateLimit.js
var rateLimit = require('express-rate-limit');

// Different limits for different endpoint types
module.exports = {
    // Strict limit for auth endpoints
    auth: rateLimit({
        windowMs: 15 * 60 * 1000,  // 15 minutes
        max: 5,
        message: { error: 'Too many login attempts' }
    }),

    // Standard API limit
    api: rateLimit({
        windowMs: 60 * 1000,  // 1 minute
        max: 100,
        message: { error: 'Rate limit exceeded' }
    }),

    // Generous limit for public endpoints
    public: rateLimit({
        windowMs: 60 * 1000,
        max: 300
    })
};
```

8.3 Request Logging Middleware

```
// middleware/routeLogger.js
var logger = require('/globals/server_modules/log');

module.exports = function routeLogger(req, res, next) {
    const start = process.hrtime.bigint();

    res.on('finish', () => {
        const duration = Number(process.hrtime.bigint() - start) / 1e6;
        const level = logger.getLogLevel(res.statusCode);

        logger.prod()[level]("Request completed", {
            ...logger.extractRequestContext(req),
            statusCode: res.statusCode,
            durationMs: duration.toFixed(2),
            contentLength: res.get('Content-Length')
        });
    });

    next();
};
```

---

9. SECRETS MANAGEMENT

9.1 Never Commit Secrets

Secrets should NEVER be in version control. Use environment variables, Docker secrets, or vault products.

9.2 Secrets Module Pattern

```
// secrets/dbconn.js (NOT in git)
module.exports = {
    getUrl: function() {
        return process.env.MONGODB_URL ||
               'mongodb://user:pass@host:27017/database?authSource=admin';
    }
};

// secrets/sessions.js (NOT in git)
module.exports = {
    getSecret: function() {
        return process.env.SESSION_SECRET || 'your-256-bit-secret';
    }
};
```

```
// secrets/aws-config.json (NOT in git)
{
    "accessKeyId": "AKIA...",
    "secretAccessKey": "...",
    "region": "us-east-1"
}
```

9.3 Docker Secrets

```
# docker-compose.yml
version: "3.8"
services:
  nodeserver:
    image: myapp:latest
    secrets:
      - db_password
      - session_secret
    environment:
      - MONGODB_URL_FILE=/run/secrets/db_password

secrets:
  db_password:
    external: true
  session_secret:
    external: true
```

9.4 Environment-Based Configuration

```
// config/index.js
const env = process.env.NODE_ENV || 'development';

const configs = {
    development: {
        db: { url: 'mongodb://localhost:27017/dev' },
        redis: { url: 'redis://localhost:6379' },
        logLevel: 'debug'
    },
    production: {
        db: { url: process.env.MONGODB_URL },
        redis: { url: process.env.REDIS_URL },
        logLevel: 'info'
    }
};

module.exports = configs[env];
```

---

10. DOCKER AND CONTAINER PATTERNS

10.1 Node.js Dockerfile

```
# Dockerfile
FROM node:22-alpine

# Security: Don't run as root
RUN addgroup -g 1001 -S nodejs && adduser -S nodeuser -u 1001

WORKDIR /app

# Copy package files first for layer caching
COPY package*.json ./

# Use npm ci for reproducible builds
RUN npm ci --only=production && npm cache clean --force

# Copy application code
COPY --chown=nodeuser:nodejs . .

USER nodeuser

# Memory limits for Node.js
ENV NODE_OPTIONS="--max-old-space-size=300"
ENV NODE_ENV=production

EXPOSE 3030

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

10.2 Docker Compose for Development

```
# docker-compose.yml
version: "3.8"
services:
  nodeserver:
    build: .
    ports:
      - "3030:3030"
    environment:
      - NODE_ENV=development
      - MONGODB_URL=mongodb://mongodb:27017/myapp
    volumes:
      - ./:/app
      - /app/node_modules
    depends_on:
      - mongodb

  mongodb:
    image: mongo:7
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

volumes:
  mongodb_data:
```

10.3 Docker Swarm Production Configuration

```
# docker-compose.prod.yml
version: "3.8"
services:
  nodeserver:
    image: myregistry/nodeserver:latest
    deploy:
      mode: replicated
      replicas: 6
      placement:
        max_replicas_per_node: 3
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
      resources:
        limits:
          cpus: '0.50'
          memory: 400M
        reservations:
          cpus: '0.20'
          memory: 150M
    environment:
      - NODE_ENV=production
      - DD_SERVICE=nodeserver
      - DD_ENV=production
    secrets:
      - db_credentials
    networks:
      - appnet

secrets:
  db_credentials:
    external: true

networks:
  appnet:
    driver: overlay
    attachable: true
```

---

11. PM2 PROCESS MANAGEMENT

11.1 PM2 Configuration

PM2 is the gold standard for Node.js process management in production when not using containers.

```
// ecosystem.config.js
module.exports = {
    apps: [{
        name: 'nodeserver',
        script: 'server.js',
        instances: 'max',           // Use all CPU cores
        exec_mode: 'cluster',       // Enable cluster mode
        max_memory_restart: '300M', // Auto-restart on memory leak

        // Environment variables
        env: {
            NODE_ENV: 'development',
            PORT: 3030
        },
        env_production: {
            NODE_ENV: 'production',
            PORT: 61339
        },

        // Logging
        error_file: '/var/log/pm2/error.log',
        out_file: '/var/log/pm2/out.log',
        log_date_format: 'YYYY-MM-DD HH:mm:ss Z',

        // Graceful shutdown
        kill_timeout: 5000,
        wait_ready: true,
        listen_timeout: 10000
    }]
};
```

11.2 PM2 Commands Reference

```
# Start with ecosystem file
pm2 start ecosystem.config.js --env production

# Monitor all processes
pm2 monit

# View logs
pm2 log

# Reload with zero downtime
pm2 reload nodeserver

# Save current process list (survives reboot)
pm2 save

# Setup startup script
pm2 startup
```

---

12. SECURITY BEST PRACTICES

12.1 Input Validation

Never trust user input. Always validate and sanitize.

```
// Always sanitize strings before database operations
function safeString(str, maxLength = 1000) {
    if (!str) return '';
    return String(str)
        .substring(0, maxLength)
        .replace(/[<>]/g, '')      // Prevent XSS
        .replace(/[$]/g, '');      // Prevent NoSQL injection
}

// Validate MongoDB ObjectIds
function isValidObjectId(id) {
    if (!id) return false;
    const ObjectID = require('mongodb').ObjectID;
    return ObjectID.isValid(id) && String(new ObjectID(id)) === String(id);
}

// Use parameterized queries (ORMs handle this automatically)
// NEVER use string concatenation for queries
const safeQuery = { email: safeString(userInput.email) };
```

12.2 Security Headers (via nginx or helmet)

```
# nginx.conf security headers
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" alwa
add_header X-Frame-Options "SAMEORIGIN";
add_header X-XSS-Protection "1; mode=block";
add_header X-Content-Type-Options nosniff;
```

```
// Or use helmet.js in Express
const helmet = require('helmet');
app.use(helmet());
```

12.3 Dependency Scanning

```
# Run regularly in CI/CD
npm audit
npm audit fix

# Use snyk for deeper scanning
npx snyk test

# Keep dependencies updated
npx npm-check-updates -u
```

---

13. UPGRADE STRATEGIES

13.1 MongoDB Driver Upgrade Path

When upgrading MongoDB driver across major versions, follow these steps:

1. Create a comprehensive test suite for all database operations
2. Update your db.js wrapper to use new API patterns
3. Run tests against the new driver version
4. Deploy to staging environment
5. Monitor for connection pool issues and memory leaks
6. Deploy to production with ability to rollback

13.2 Deprecated Package Replacements

Common deprecated packages and their modern replacements:

DEPRECATED	REPLACEMENT
`request`	`axios` or native `fetch` (Node 18+)
`node-uuid`	`uuid`
`aws-sdk` v2	`@aws-sdk/client-*` v3
`node-sass`	`sass` (dart-sass)
`connect-mongo` v3	`connect-mongo` v5

---

14. CODE EXAMPLES AND TEMPLATES

14.1 Complete Task Script Template

```
// tasks/processData.js
const db = require('/globals/server_modules/db');
const logger = require('/globals/server_modules/log');

const TASK_NAME = 'processData';

async function main() {
    logger.prod().info(`${TASK_NAME} - Started`, {
        evt: { category: 'task', name: TASK_NAME, outcome: 'started' }
    });

    try {
        // Main task logic
        const items = await db.getCollection('items', [
            { $match: { processed: { $exists: false } } },
            { $limit: 100 }
        ]);

        let processed = 0;
        for (const item of items) {
            await processItem(item);
            processed++;
        }

        logger.prod().info(`${TASK_NAME} - Completed`, {
            processed,
            evt: { category: 'task', name: TASK_NAME, outcome: 'success' }
        });

    } catch (err) {
        logger.prod().error(`${TASK_NAME} - Failed`, {
            error: err,
            evt: { category: 'task', name: TASK_NAME, outcome: 'failure' }
        });
        throw err;
    }
}

async function cleanup(signal) {
    logger.prod().info(`${TASK_NAME} - Cleanup`, { signal });
    db.cleanupDB();
    process.exit(0);
}

['SIGINT', 'SIGTERM', 'uncaughtException', 'unhandledRejection'].forEach(sig =>
    process.on(sig, () => cleanup(sig));
```

```
});

main()
    .then(() => cleanup('completed'))
    .catch(() => process.exit(1));
```

14.2 Complete Watcher Template

```
// watchers/watchOrders.js
const db = require('/globals/server_modules/db');
const logger = require('/globals/server_modules/log');

const WATCHER_NAME = 'watchOrders';
let changeStream;

async function startWatcher() {
    logger.prod().info(`${WATCHER_NAME} - Starting`);

    try {
        changeStream = await db.watchCollection('orders', [], {
            fullDocument: 'updateLookup'
        });

        changeStream.on('change', handleChange);
        changeStream.on('error', handleError);

        logger.prod().info(`${WATCHER_NAME} - Watching`);

    } catch (err) {
        logger.prod().error(`${WATCHER_NAME} - Failed to start`, { error: err }
        setTimeout(startWatcher, 5000);
    }
}

async function handleChange(change) {
    const doc = change.fullDocument;
    if (!doc) return;

    try {
        // Process the change
        logger.prod().debug(`${WATCHER_NAME} - Change detected`, {
            operationType: change.operationType,
            documentId: doc._id
        });

        // Your business logic here

    } catch (err) {
        logger.prod().error(`${WATCHER_NAME} - Processing failed`, {
            error: err,
            documentId: doc._id
        });
    }
}
```

```
function handleError(err) {
    logger.prod().error(`${WATCHER_NAME} - Stream error`, { error: err });
    setTimeout(startWatcher, 5000);
}

async function cleanup(signal) {
    logger.prod().info(`${WATCHER_NAME} - Cleanup`, { signal });
    if (changeStream) await changeStream.close();
    db.cleanupDB();
    process.exit(0);
}

['SIGINT', 'SIGTERM', 'uncaughtException', 'unhandledRejection'].forEach(sig =>
    process.on(sig, () => cleanup(sig));
});

startWatcher();
```

---

15. SUMMARY CHECKLIST

Before Deploying to Production

- [ ] All processes handle SIGINT, SIGTERM, uncaughtException, unhandledRejection
- [ ] Database connections use connection pooling with proper cleanup
- [ ] All async operations use try/catch with proper error logging
- [ ] Secrets are NOT in version control
- [ ] npm audit shows no critical vulnerabilities
- [ ] package-lock.json is committed and npm ci is used for installs
- [ ] NODE_ENV=production is set
- [ ] Memory limits are configured (NODE_OPTIONS=--max-old-space-size)
- [ ] Structured logging is configured with proper log levels
- [ ] Health check endpoint exists (/api/health)
- [ ] Rate limiting is configured for all endpoints
- [ ] Input validation is implemented for all user inputs

This guide represents patterns proven over 10 years of production Node.js development. The key insight: abstractions matter. Wrap your dependencies, handle your errors, and log everything.

Document generated: January 2026