A Developer's Guide to Mastering Node.js: 20 Practical Exercises

Kushan

August 26, 2025

Introduction

This guide is designed for developers who already understand software engineering principles (like you, coming from Java microservices) and want to achieve a deep, practical mastery of the Node.js ecosystem. These exercises are designed to be challenging and cover the breadth of skills expected of a developer with ˜5 years of experience in Node.js.

1 Asynchronous Flow Control & Node.js Internals

1.1 Exercise 1: Build a Promise-Based Rate Limiter

- **Objective:** Deeply understand `async/await` and Promise manipulation.
- **Problem:** Create a class or function that limits how many asynchronous tasks can run concurrently. For example, if you have 100 tasks but a concurrency limit of 10, only 10 should run at a time. When one finishes, the next one in the queue should start.
- **Requirements:** Must be implemented from scratch using `async/await`. It should accept an array of promise-returning functions and a concurrency limit.
- **Success Criteria:** The rate limiter correctly processes all tasks without ever exceeding the specified concurrency limit.

1.2 Exercise 2: Create a Custom Duplex Stream

- **Objective:** Master the Node.js Stream API.
- **Problem:** Create a custom Duplex stream that transforms data. For example, it could be a stream that takes JSON strings, parses them, adds a `processedAt` timestamp, and then stringifies them again to be piped to a destination.
- **Requirements:** Must extend the `stream.Duplex` class. It should handle back-pressure correctly.
- **Success Criteria:** You can successfully pipe a readable stream of JSON data through your custom stream to a writable stream, and the output is correctly transformed.

1.3 Exercise 3: Multi-threaded File Processor

- **Objective:** Understand when and how to use `worker`$_t hreadsforCPU-boundtasks.$`Problem:`$Writeascriptthattake$

- **Requirements:** Use the `worker`$_t hreadsmodule.Themainthreadshouldaggregatetheresultsfromallworkersandprintaf$

1.4 Exercise 4: Graceful Shutdown Server

- **Objective:** Implement production-grade error handling and process management.
- **Problem:** Create a simple Express server. Implement a function that ensures a graceful shutdown. When the server receives a `SIGTERM` or `SIGINT` signal (like from Docker or Ctrl+C), it should stop accepting new connections, finish processing any existing requests, close the database connection, and then exit.

- **Requirements:** Must handle `process.on('SIGTERM')`. The server should respond with a 503 Service Unavailable status if a request comes in during shutdown.
- **Success Criteria:** When you run the server and press Ctrl+C, it logs that it is shutting down gracefully and waits a few seconds before exiting.

---

2 API Development & Frameworks (using Express.js or Fastify)

2.1 Exercise 5: Build a RESTful API with Advanced Routing

- **Objective:** Master a web framework (Express.js is recommended).
- **Problem:** Build a REST API for a simple blog platform. It should have resources for `users`, `posts`, and `comments`. Implement full CRUD (Create, Read, Update, Delete) operations for each.
- **Requirements:** Use Express Router to modularize your routes (e.g., `routes/posts.js`). Implement a logging middleware that logs the method, URL, and response time of every request.
- **Success Criteria:** All endpoints work as expected via a tool like Postman or Insomnia.

2.2 Exercise 6: JWT-Based Authentication & Authorization

- **Objective:** Implement secure user authentication.
- **Problem:** Add JWT-based authentication to the blog API. Create `/login` and `/register` endpoints. All `POST`, `PUT`, `DELETE` endpoints should be protected and require a valid JWT. Furthermore, a user should only be able to edit or delete their own posts.
- **Requirements:** Use a library like `jsonwebtoken` and `bcrypt`. Create a middleware to verify the token and attach the user object to the request.
- **Success Criteria:** Unauthenticated requests to protected routes fail with a 401. Authenticated users can access them, but cannot modify resources owned by other users (receives a 403).

2.3 Exercise 7: Input Validation and Error Handling

- **Objective:** Create robust and predictable APIs.
- **Problem:** Add validation to your blog API. When creating a user, the email must be a valid email format and the password must be at least 8 characters long. When creating a post, the title cannot be empty.
- **Requirements:** Use a validation library like `zod` or `joi`. Create a centralized error handling middleware that catches validation errors and returns a structured 400 Bad Request response.
- **Success Criteria:** Sending invalid data to the API results in a clean, predictable JSON error message, not a server crash or HTML error page.

2.4 Exercise 8: Convert the API to TypeScript

- **Objective:** Master TypeScript in a Node.js context.
- **Problem:** Convert your entire JavaScript Express API into TypeScript.
- **Requirements:** Add types for all request bodies, route parameters, and database models. Configure `tsconfig.json` for a Node.js project. Use `ts-node-dev` or equivalent for development.
- **Success Criteria:** The project is fully typed, runs correctly, and has zero `any` types.

---

3 Databases & Caching

3.1 Exercise 9: SQL Integration with an ORM

- **Objective:** Master interacting with a SQL database.
- **Problem:** Integrate your blog API with a PostgreSQL database. Use a modern ORM like Prisma or Sequelize.
- **Requirements:** Define the schema for users, posts, and comments. All API logic should now read from and write to the PostgreSQL database instead of in-memory arrays. Implement a transaction when creating a post that also allows creating tags for the post.
- **Success Criteria:** Data is persisted between server restarts. The relationship between users, posts, and comments is correctly modeled and queried.

3.2 Exercise 10: NoSQL (MongoDB) Integration

- **Objective:** Understand NoSQL data modeling.
- **Problem:** Create a new service or a new set of endpoints for user profiles that stores flexible data (e.g., social media links, hobbies, profile picture URL). Use MongoDB with Mongoose for this.
- **Requirements:** Define a Mongoose schema for the user profile. The schema should allow for an array of social media links, which is a good use case for NoSQL.
- **Success Criteria:** You can create and retrieve complex, nested user profile documents via your API.

3.3 Exercise 11: Implement a Redis Cache

- **Objective:** Improve API performance with caching.
- **Problem:** Add a caching layer to your blog API using Redis. When a user requests a specific post, first check if itś in the Redis cache. If so, return it from there. If not, fetch it from the database, store it in the cache for future requests, and then return it.
- **Requirements:** Use a Redis client library like `ioredis`. When a post is updated or deleted, the cache for that post must be invalidated (cleared).
- **Success Criteria:** Hitting the same `GET /posts/:id` endpoint multiple times results in only one database query, with subsequent requests being served much faster from Redis.

---

4 Microservices & Distributed Systems

4.1 Exercise 12: Build a GraphQL Gateway

- **Objective:** Understand GraphQL as an alternative to REST.
- **Problem:** Create a new GraphQL endpoint that acts as a gateway. It should be able to fetch a user and, within the same query, fetch all the posts written by that user.
- **Requirements:** Use `apollo-server-express`. Define your GraphQL schema (types, queries, mutations). Write resolver functions that fetch data from your existing services or database.
- **Success Criteria:** A single GraphQL query can fetch a user and their posts, avoiding the multiple round trips required in a typical REST API.

4.2 Exercise 13: Inter-Service Communication (REST & gRPC)

- **Objective:** Build a simple microservices system.
- **Problem:** Split your application into two services: `users-service` and `posts-service`. The `posts-service` needs to get author information. Implement this communication first using a simple REST call (e.g., with `axios`). Then, refactor it to use gRPC for more efficient, strongly-typed communication.
- **Requirements:** Each service runs on a different port. For the gRPC part, you will need to define a `.proto` file.
- **Success Criteria:** The system works as a whole. You can clearly see the network request between the two services for both REST and gRPC implementations.

4.3 Exercise 14: Asynchronous Communication with a Message Queue

- **Objective:** Decouple services using a message broker.
- **Problem:** Add a `notifications-service`. When a new comment is posted on the `posts-service`, instead of handling it directly, the `posts-service` should publish a `comment.created` event to a RabbitMQ (or Kafka) queue. The `notifications-service` will be the consumer of this queue, and upon receiving a message, it will simply log "Sending notification for comment...".
- **Requirements:** Use a library like `amqplib`. The `posts-service` should not know or care about the `notifications-service`.
- **Success Criteria:** When you post a comment, the `posts-service` responds instantly, and the `notifications-service` logs its message shortly after, demonstrating decoupling.

---

5 Testing & DevOps

5.1 Exercise 15: Comprehensive API Testing

- **Objective:** Master modern testing strategies.
- **Problem:** Write a full test suite for your blog API.
- **Requirements:**
  1. **Unit Tests (Jest):** Test individual functions, like a utility that formats dates.
  2. **Integration Tests (Supertest):** Test your API endpoints. Use an in-memory database (like SQLite) or a dedicated test database to test the full request-response cycle without mocking the database.
  3. **Mocking:** Mock the JWT library so you dont́ need real tokens in tests. Mock any external API calls.
- **Success Criteria:** Your tests achieve a high level of coverage (e.g., ¿80%) and run automatically.

5.2 Exercise 16: Containerize the Application with Docker

- **Objective:** Prepare an application for modern deployment.
- **Problem:** Write a `Dockerfile` for your main API service.

- **Requirements:** Use a multi-stage build. The first stage should install dependencies (including `devDependencies`) and build your TypeScript code. The final stage should be a slim Node.js image, copying only the compiled code and `node`$_m$*odulesneededforproduction.Payattentiontocachinglayers.*`Success Criteria:`*Youcansuccessfull*

5.3 Exercise 17: Orchestrate with Docker Compose

- **Objective:** Manage a multi-service local environment.
- **Problem:** Create a `docker-compose.yml` file that starts your entire application stack: your Node.js API container, a PostgreSQL container, and a Redis container.
- **Requirements:** The services should be able to communicate with each other using their service names. Use environment variables to pass configuration like database credentials.
- **Success Criteria:** Running `docker-compose up` starts the entire application, and everything is fully functional.

5.4 Exercise 18: Implement Production-Grade Logging

- **Objective:** Create observable applications.
- **Problem:** Replace all `console.log` statements with a structured logger like `pino` or `winston`.
- **Requirements:** Logs should be in JSON format. Log levels (`info`, `warn`, `error`) should be used appropriately. All logs should include a timestamp and a request ID to trace a single request through the system.
- **Success Criteria:** Your applicationś output is structured, machine-readable JSON, not plain text.

5.5 Exercise 19: Add Security Headers and Rate Limiting

- **Objective:** Secure a public-facing API.
- **Problem:** Add basic security measures to your API.
- **Requirements:** Use `helmet` to set secure HTTP headers. Implement a rate limiter that prevents a single IP address from making more than 100 requests per minute to the API.
- **Success Criteria:** You can see the security headers in the response, and making rapid requests eventually results in a 429 Too Many Requests error.

5.6 Exercise 20: Create a CI/CD Pipeline

- **Objective:** Automate the testing and build process.
- **Problem:** Create a simple Continuous Integration pipeline using GitHub Actions (or GitLab CI).
- **Requirements:** The pipeline should be triggered on every push to the `main` branch. It should perform the following steps: install dependencies, run the linter, run all tests, and (as a bonus) build the Docker image.
- **Success Criteria:** A push to GitHub automatically triggers the pipeline, and you can see it pass or fail based on your code quality and test results.