Great 🚀 Let's move on to the **last technical section**.

---

22. Advanced Patterns & Best Practices in Node.js

As apps grow, **maintainability, scalability, and performance** become crucial. Advanced patterns and best practices help achieve clean, reliable, and future-proof code.

---

1. Modular Code & Layered Architecture

- Split code into **layers**:
  - **Routes** → Handle requests.
  - **Controllers** → Business logic.
  - **Services** → Reusable logic (DB, external APIs).
  - **Models** → Database schemas.
- Promotes **separation of concerns**.

📌 Example Structure:

```
/routes
/controllers
/services
/models
/utils
```

---

2. Dependency Injection

- Avoid hardcoding dependencies, make them swappable.
- Example:

```
class UserService {
  constructor(userRepo) {
    this.userRepo = userRepo;
  }
}
```

---

3. Middleware Patterns

- **Global Middleware** → e.g., logging, authentication.
- **Route-Specific Middleware** → applied only where needed.

- Avoid "middleware hell" by composing smaller middlewares.

---

4. Error Handling Patterns

- Centralized error handler in Express:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ message: "Something went wrong!" });
});
```

- Use **custom error classes** for clarity.

---

5. Async & Promise Patterns

- Always handle **promise rejections**:

```
process.on("unhandledRejection", err => {
  console.error("Unhandled rejection:", err);
});
```

- Use `util.promisify` to convert callbacks into Promises.

---

6. Configuration Management

- Use environment-specific configs:
  - `.env.development`, `.env.production`.
- Use libraries like **dotenv**, **config**, or **convict**.

---

7. Design Patterns in Node.js

- **Singleton** → DB connection pool.
- **Factory** → Creating objects like services.
- **Observer** → EventEmitter.
- **Proxy** → Middleware for caching or logging.
- **Strategy** → Swappable algorithms (e.g., different payment providers).

---

8. Security Best Practices

- Validate inputs (Joi, Yup, Zod).

- Sanitize against XSS & SQL Injection.
- Rate limiting & brute-force protection.
- Use **Helmet.js** for HTTP headers security.
- Avoid storing secrets in code → use Vaults or env vars.

---

9. Performance Best Practices

- Use **cluster mode** or PM2.
- Cache frequently accessed data (Redis).
- Use **streams** for large files.
- Avoid blocking operations (e.g., heavy computations).
- Monitor with APM tools.

---

10. Code Quality & Maintainability

- **Linting & Formatting**: ESLint + Prettier.
- **Testing Pyramid**:
    - Unit Tests → Most.
    - Integration Tests.
    - E2E Tests → Least.
- Document APIs (Swagger, Postman collections).
- Use TypeScript for better maintainability.

---

11. CI/CD & Deployment Patterns

- Automate tests before deployment.
- Use blue-green or rolling deployments.
- Ensure zero-downtime restarts (PM2 reload, K8s rolling updates).

---

12. Observability

- Centralized logging → Winston + ELK/Graylog.
- Metrics → Prometheus + Grafana.
- Error tracking → Sentry, New Relic.
- Health check endpoints → `/health`, `/ready`.

---

13. Monorepo & Microservices Best Practices

- Use **monorepo tools** (Nx, Turborepo) for large codebases.
- For microservices:

    - Use message queues (RabbitMQ, Kafka).
    - Service discovery & API gateway.
    - Isolate failures with circuit breakers (Hystrix pattern).

---

14. When to Use Node.js

- **Best suited for**:
    - Real-time apps (chat, gaming).
    - APIs & microservices.
    - Streaming apps.
- **Not suited for**:
    - CPU-heavy tasks (unless offloaded to workers/child processes).

✅ This completes **Advanced Patterns & Best Practices in Node.js**.

Would you like me to now prepare a **final polished index (v2)** that includes **all 22 topics + subtopics** so you can use it as your master ToC?