

# Node.js best practices along with sample code



Parmar shyamsinh · [Follow](#)

6 min read · Jan 12, 2024



156



1



Photo by [Christopher Gower](#) on [Unsplash](#)

# Mastering Node.js Performance: Essential Best Practices for Efficient and Scalable Apps

Unlock the full potential of Node.js by mastering these essential best practices. Build efficient, scalable, and performant applications with confidence.

## Introduction:

- Briefly introduce Node.js and its popularity for building fast and scalable web applications.
- Highlight the importance of following best practices to ensure optimal performance and maintainability.
- State the key benefits of adhering to best practices, such as improved speed, scalability, security, and code readability.

Here are some Node.js best practices along with sample code snippets to illustrate each practice:

## 1. Use the Latest LTS Version:

Always use the latest LTS (Long Term Support) version of Node.js to benefit from stability and security updates.

## 2. Project Structure:

Organize your project structure for scalability and maintainability.

```
/project  
/src
```

```
/controllers  
/models  
/routes  
/services  
/config  
/tests
```

### 3. Dependency Management:

Use a `package.json` file to manage dependencies. Specify versions to ensure consistent builds.

```
{  
  "dependencies": {  
    "express": "^4.17.1",  
    "mongoose": "^6.0.12"  
  },  
  "devDependencies": {  
    "jest": "^27.2.4"  
  }  
}
```

### 4. Environment Variables:

Use `dotenv` for managing environment variables.

```
// .env  
PORT=3000  
  
// server.js  
require('dotenv').config();  
  
const port = process.env.PORT || 3000;
```

## 5. Logging:

Implement logging for better debugging and monitoring.

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

winston.log('info', 'Hello, this is an info message.');
```

```
winston.error('Oops! An error occurred.');
```

## 6. Error Handling:

Implement consistent error handling.

```
try {
  // Code that might throw an error
} catch (error) {
  console.error('An error occurred:', error.message);
}
```

## 7. Async/Await:

Use `async/await` for handling asynchronous operations.

```
async function fetchData() {
  try {
    const result = await fetch('https://api.example.com/data');
    const data = await result.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
```

```
}  
}
```

## 8. Middleware:

Use middleware for common functionality like authentication.

```
// Authentication middleware  
function authenticate(req, res, next) {  
  // Check authentication logic  
  if (authenticated) {  
    return next();  
  } else {  
    return res.status(401).send('Unauthorized');  
  }  
}
```

## 9. Testing:

Write unit tests for your code using testing frameworks like Jest.

```
// Sample Jest test  
test('adds 1 + 2 to equal 3', () => {  
  expect(sum(1, 2)).toBe(3);  
});
```

## 10. Security:

Implement security best practices, such as input validation and sanitization.

```
const sanitizeInput = (input) => {  
  return input.replace(/&<>"'/]/g, (char) => {  
    switch (char) {  
      case '&': return '&amp;';  
      case '<': return '&lt;';  
      case '>': return '&gt;';  
      case '"': return '&quot;';  
      case "'": return '&#x27;';  
      case '/': return '&#x2F;';  
      default: return char;  
    }  
  });  
};
```

## 11. Code Linting:

Use a linter (e.g., ESLint) to enforce coding standards.

```
// Sample ESLint configuration in package.json  
"eslintConfig": {  
  "extends": "eslint:recommended",  
  "rules": {  
    "no-console": "off",  
    "semi": ["error", "always"]  
  }  
}
```

## 12. Documentation:

Maintain clear and concise documentation for your code.

```
/**  
 * Function to add two numbers.  
 * @param {number} a - The first number.  
 * @param {number} b - The second number.
```

```
* @returns {number} - The sum of a and b.
*/
function add(a, b) {
  return a + b;
}
```

## 13. Continuous Integration (CI) and Continuous Deployment (CD):

- Set up CI/CD pipelines using a tool like GitHub Actions.

```
# .github/workflows/main.yml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Install Dependencies
        run: npm install

      - name: Run Tests
        run: npm test

      - name: Deploy to Production
        if: success()
        run: npm run deploy
```

## 14. Performance Optimization:

Open in app ↗



```
const NodeCache = require('node-cache');
const cache = new NodeCache();

function fetchDataFromAPI() {
  // Logic to fetch data from API
}

function getCachedData() {
  const key = 'cachedData';
  const cachedData = cache.get(key);

  if (cachedData) {
    return cachedData;
  } else {
    const newData = fetchDataFromAPI();
    cache.set(key, newData, 3600); // Cache for 1 hour
    return newData;
  }
}
```

## 15. Graceful Shutdown:

- Implement graceful shutdown to handle server shutdowns gracefully.

```
process.on('SIGTERM', () => {
  server.close(() => {
    console.log('Server gracefully shut down');
    process.exit(0);
  });
});

process.on('SIGINT', () => {
  server.close(() => {
    console.log('Server interrupted. Shutting down');
    process.exit(1);
  });
});
```



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

## 16. Health Check Endpoint:

- Include a health check endpoint for monitoring purposes.

```
// routes/health.js  
const express = require('express');  
const router = express.Router();  
  
router.get('/', (req, res) => {  
  res.status(200).json({ status: 'OK' });  
});  
  
module.exports = router;
```

```
// server.js  
const healthRouter = require('./routes/health');  
app.use('/health', healthRouter);
```

## 17. Secure Headers:

- Use the `helmet` middleware to secure your application headers.

```
const express = require('express');  
const helmet = require('helmet');  
const app = express();  
  
app.use(helmet());
```

## 18. Database Connection Pooling:

- Use connection pooling for database connections.

```
const { Pool } = require('pg');

const pool = new Pool({
  user: 'your_user',
  host: 'your_host',
  database: 'your_database',
  password: 'your_password',
  port: 5432,
  max: 20, // Set the maximum number of clients in the pool
  idleTimeoutMillis: 30000, // Close idle clients after 30 seconds
  connectionTimeoutMillis: 2000, // Close new clients after 2 seconds of inactiv
});

pool.query('SELECT * FROM your_table', (err, res) => {
  // Handle query results
});
```

## 19. Request Validation:

- Use a request validation library, such as `express-validator`, to validate incoming requests.

```
const { body, validationResult } = require('express-validator');

app.post('/user', [
  // Validate and sanitize fields
  body('username').isAlphanumeric().trim(),
  body('email').isEmail().normalizeEmail(),
  body('password').isLength({ min: 5 }).trim(),
], (req, res) => {
  // Handle the request
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
});
```

```
}  
  
// Continue with the request handling  
});
```

## 20. JWT Authentication:

- Implement JWT (JSON Web Token) authentication for securing APIs.

```
const jwt = require('jsonwebtoken');  
  
// Middleware to verify JWT token  
function verifyToken(req, res, next) {  
  const token = req.header('Authorization');  
  
  if (!token) {  
    return res.status(401).json({ message: 'Unauthorized' });  
  }  
  
  jwt.verify(token, 'your-secret-key', (err, decoded) => {  
    if (err) {  
      return res.status(401).json({ message: 'Invalid token' });  
    }  
  
    req.user = decoded.user;  
    next();  
  });  
}  
  
// Example usage in a route  
app.get('/protected-route', verifyToken, (req, res) => {  
  // Access granted for authenticated user  
});
```

## 21. Docker Integration:

- Use Docker for containerization to simplify deployment.

```
# Dockerfile
FROM node:14

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

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

## 22. Internationalization (i18n):

- Implement internationalization for multilingual support

```
const i18n = require('i18n');

i18n.configure({
  locales: ['en', 'fr', 'es'],
  directory: __dirname + '/locales',
  defaultLocale: 'en',
});

app.use(i18n.init);

// Example usage in a route
app.get('/greet', (req, res) => {
  res.send(res.__( 'Hello!' ));
});
```

## 23. Rate Limiting:

- Use a rate-limiting middleware to prevent abuse or DoS attacks.

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

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
});

app.use(limiter);
```

## 24. GraphQL Integration:

- Integrate GraphQL for flexible API queries.

```
const { ApolloServer, gql } = require('apollo-server-express');

const typeDefs = gql`
  type Query {
    hello: String
  }
`;

const resolvers = {
  Query: {
    hello: () => 'Hello, World!',
  },
};

const server = new ApolloServer({ typeDefs, resolvers });
server.applyMiddleware({ app });
```

## 25. Background Jobs:

- Use a job queue or background processing system for handling tasks asynchronously.

```
const Queue = require('bull');

const myQueue = new Queue('my queue');

myQueue.process(async (job) => {
  // Process the job data asynchronously
  console.log(`Processing job with data: ${job.data}`);
});

// Enqueue a job
myQueue.add({ data: 'example job data' });
```

## 26. File Uploads:

- Implement file uploads securely using packages like `multer`.

```
const express = require('express');
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });

const app = express();

app.post('/upload', upload.single('file'), (req, res) => {
  // Handle the uploaded file
  const file = req.file;
  res.send(`File uploaded: ${file.originalname}`);
});
```

## 27. CORS Handling:

- Use the `cors` middleware to handle Cross-Origin Resource Sharing.

```
const express = require('express');
const cors = require('cors');
```

```
const app = express();  
  
// Enable CORS for all routes  
app.use(cors());
```

## 28. API Versioning:

- Implement API versioning to manage changes over time.

```
const express = require('express');  
const app = express();  
  
// Version 1  
app.get('/api/v1/users', (req, res) => {  
  // Handle version 1 of the API  
});  
  
// Version 2  
app.get('/api/v2/users', (req, res) => {  
  // Handle version 2 of the API  
});
```

## 29. Monitoring and Logging Tools:

- Integrate monitoring and logging tools, such as Prometheus, Grafana, or ELK Stack.

```
const prometheus = require('prom-client');  
const express = require('express');  
const app = express();  
  
// Prometheus metrics endpoint  
app.get('/metrics', (req, res) => {  
  res.set('Content-Type', prometheus.register.contentType);
```

```
res.end(prometheus.register.metrics());  
});
```

## 30. WebSockets with Socket.IO:

- Use Socket.IO for easier WebSocket implementation.

```
const express = require('express');  
const http = require('http');  
const socketIO = require('socket.io');  
  
const app = express();  
const server = http.createServer(app);  
const io = socketIO(server);  
  
io.on('connection', (socket) => {  
  console.log('A user connected');  
  
  // Handle socket events  
  socket.on('chat message', (msg) => {  
    io.emit('chat message', msg);  
  });  
  
  socket.on('disconnect', () => {  
    console.log('User disconnected');  
  });  
});  
  
server.listen(3000, () => {  
  console.log('Server is listening on port 3000');  
});
```

## Conclusion:

- Summarize the key takeaways and benefits of following Node.js best practices.



- Encourage readers to continuously learn and adopt new best practices as Node.js evolves.
- Provide a call to action for further exploration and experimentation.

More: <https://woycetech.com/nodejs/node-js-best-practices-2024/>

Nodejs

Node Js Development

Best Practices

Node Js Tutorial

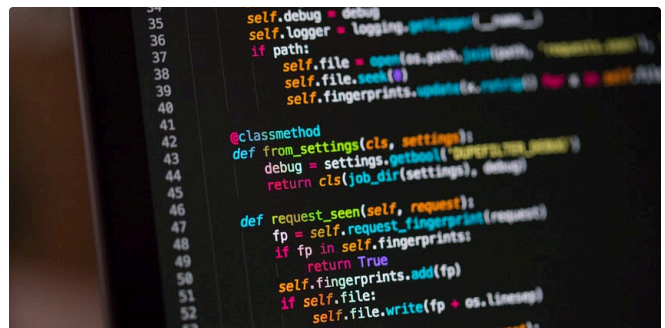
**Written by Parmar shyamsinh**

29 Followers · 3 Following

Follow

ChatGPT | AWS | Backend Developer | BotPress | Chatwoot | Vertex AI  
[parmarshyamsingh8@gmail.com](mailto:parmarshyamsingh8@gmail.com)

## More from Parmar shyamsinh





Parmar shyamsinh

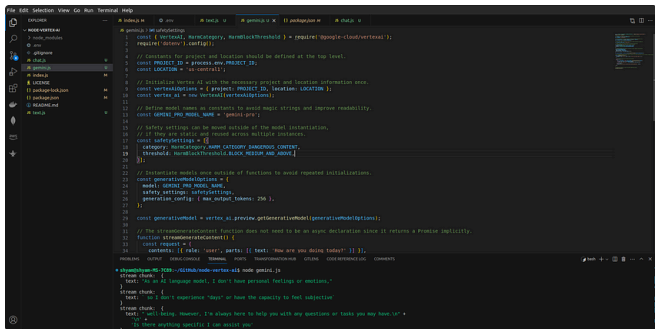
# Getting Started with PostgreSQL: A Beginner's Guide

PostgreSQL is a versatile and feature-rich database system that caters to both...

Jan 19

👏 114

💬 1



Parmar shyamsinh

# Integrating Vertex AI Gemini API with Node.js

Integrating Vertex AI Gemini API with Node.js: A Step-by-Step Guide

Jan 4

👏 116

💬 1



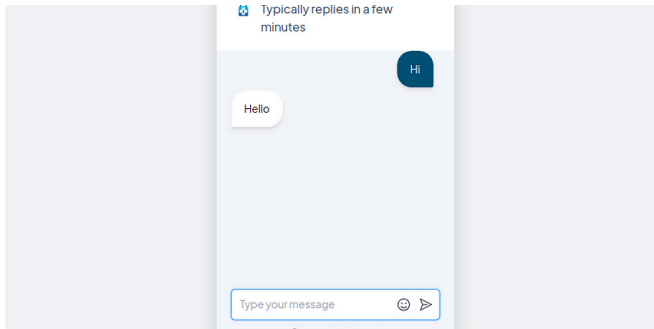
Parmar shyamsinh

# Mistral 7B LLM: Run Locally with Ollama

Mistral 7B LLM: Run Locally with OllamaMistral 7B LLM

Jan 17

👏 128



Parmar shyamsinh

# How to Create a Self-Hosted Chat Widget with Chatwoot

How to Create a Self-Hosted Chat Widget with Chatwoot

Jan 5

👏 100



See all from Parmar shyamsinh

Recommended from Medium

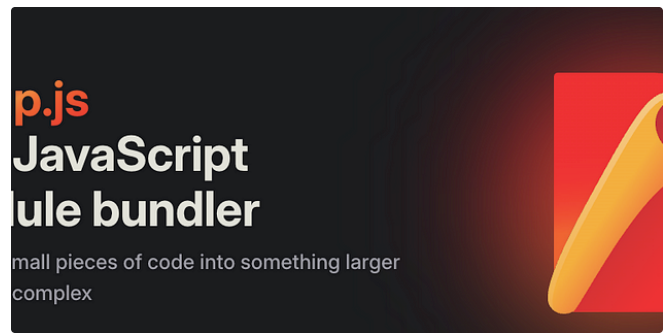



 In Coding Beauty by Tari Ibaba

## This new JavaScript operator is an absolute game changer

Say goodbye to try-catch

★ Sep 18 🖱 4.1K 💬 53



 Abdul Wahid

## How to Create an NPM Package Using Rollup.js

In this article, we are going to understand the need to use Rollup.js to create an NPM...

Jun 22 🖱 1



### Lists



#### Stories to Help You Grow as a Software Developer

19 stories · 1471 saves



#### data science and AI

40 stories · 282 saves

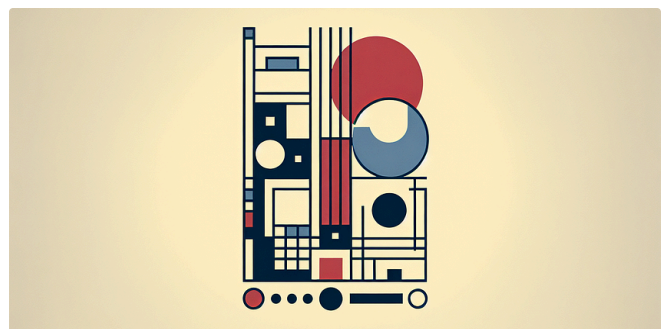



#### Natural Language Processing

1812 stories · 1424 saves



 habtesoft

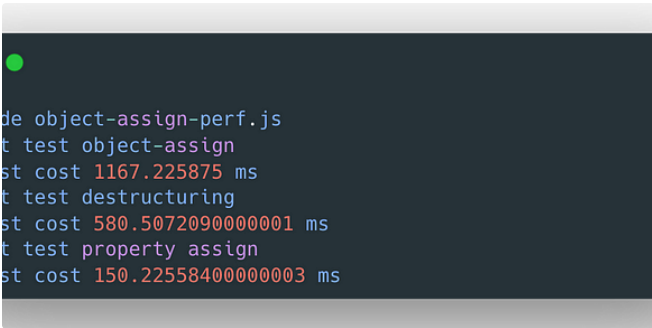



 In JavaScript in Plain English by Peter Kracik

# Helmet: Enhancing Security in Node.js

Security is a critical concern in web development, especially when building APIs...

★ Sep 16 🖱 16 📌<sup>+</sup>



 ohdarling

# 5x Faster Object.assign — A JavaScript Performance Trick

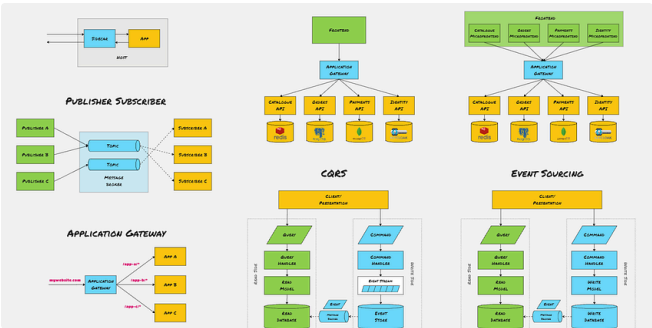
JavaScript performance optimization is a crucial aspect of modern web development....


★ 4d ago 🖱 2 📌<sup>+</sup>

# Clean Architecture Demystified: Refactoring Your Nest.js App

Transform your Nest.js app with Clean Architecture for better structure and...

Oct 28 🖱 166 💬 4 📌<sup>+</sup>



 In Level Up Coding by Matt Bentley

# My Favourite Software Architecture Patterns

Exploring my most loved Software Architecture patterns and their practical...

★ 3d ago 🖱 2.6K 💬 38 📌<sup>+</sup>

See more recommendations