BACKEND DEVELOPER INTERVIEW GUIDE 2026

SLAWOMIR PLAMOWSKI

Node.js Backend Developer Interview Guide 2026

Copyright © 2026 EasyInterview.me
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, recording, or otherwise—without prior written permission.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

The information in this book is distributed on an "as is" basis, without warranty.

First Edition: January 2026

https://easyinterview.me

Contents

How to Use This Guide

This guide contains carefully curated interview questions for Node.js Backend Developer positions. Whether you're preparing for your first junior role or aiming for a senior backend architect position, you'll find relevant questions organized by topic and difficulty.

Difficulty Levels

Questions are marked with difficulty badges:

- [Junior] — Entry-level fundamentals. If you're just starting out, master these first.
- [Mid] — Intermediate concepts requiring practical experience. Expected for mid-level roles.
- [Senior] — Advanced topics covering architecture, scalability, and system design. Required for senior positions.

Recommended Study Approach

1. **Assess your level** — Skim through chapters and note which questions you can answer confidently.
2. **Focus on gaps** — Spend more time on topics where you struggled.
3. **Practice out loud** — Explain answers as if in an interview. This builds confidence.
4. **Study the code** — Don't just read examples; type them out and experiment.
5. **Build projects** — Apply concepts in real applications to solidify your understanding.

> **Interview Reality**
> Backend interviews often focus on system design, scalability, and real-world problem-solving. Be ready to discuss trade-offs, explain your architectural decisions, and walk through debugging scenarios.

What's Covered

Chapter	Topics	Questions
1. Node.js Fundamentals	Event Loop, Streams, Modules, V8	29
2. TypeScript	Types, Generics, Decorators, Config	22
3. Express.js & REST APIs	Routing, Middleware, Auth, OpenAPI	26
4. SQL & PostgreSQL	Queries, Indexing, Transactions, CTEs	25
5. MongoDB	Documents, Aggregation, Indexes	20
6. Redis	Caching, Data Structures, Pub/Sub	18
7. Docker	Containers, Compose, Networks, Security	22
8. Testing	Unit, Integration, E2E, Mocking	20
9. Security	OWASP, Auth, Encryption, Headers	20
10. DevOps & Git	CI/CD, Deployment, Git Workflows	20

Good luck with your interview!

1

Node.js Fundamentals

Node.js revolutionized server-side development by bringing JavaScript to the backend. Built on Chrome's V8 engine and designed around an event-driven, non-blocking I/O model, Node.js excels at handling concurrent connections and building scalable network applications. This chapter explores the core concepts that every Node.js developer must understand.

1.1 Node.js Core Concepts

Q1.1: What is Node.js and how does it differ from traditional web servers? [Junior]

Node.js is a **JavaScript runtime** built on Chrome's V8 engine that allows you to run JavaScript code outside of a web browser. Unlike traditional web servers like Apache or Nginx that create a new thread for each incoming request, Node.js uses a **single-threaded event loop** with non-blocking I/O operations.

Traditional servers follow a thread-per-request model: each connection spawns a new thread, consuming memory and CPU resources. With thousands of concurrent connections, this becomes a bottleneck. Node.js handles all requests in a single thread, delegating I/O operations to the system kernel and processing them asynchronously when complete. This makes Node.js exceptionally efficient for I/O-heavy workloads like APIs, real-time applications, and microservices.

```
// A simple Node.js HTTP server
const http = require('http');

const server = http.createServer((req, res) => {
  // This callback runs for every request
  // But it doesn't block other requests
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js!');
});

server.listen(3000, () => {
```

```
  console.log('Server running on port 3000');
});
```

> **Pro Tip**
> Node.js is not ideal for CPU-intensive tasks like image processing or complex calculations, as these block the single thread. For such workloads, use Worker Threads or offload to separate services.

Q1.2: Describe the role of the single-threaded event loop in Node.js. [Junior]

The event loop is the **heart of Node.js**, responsible for executing code, collecting and processing events, and executing queued callbacks. Despite being single-threaded for JavaScript execution, Node.js achieves high concurrency through its event-driven architecture.

When Node.js starts, it initializes the event loop, processes the input script (which may include async API calls, timers, or `process.nextTick()`), and then begins processing the event loop. The loop continuously checks for pending callbacks, I/O operations, and timers, executing them when ready. This design allows thousands of concurrent operations without the overhead of thread management.

```
// The event loop in action
console.log('1. Script start');

setTimeout(() => {
  console.log('2. setTimeout callback');
}, 0);

Promise.resolve().then(() => {
  console.log('3. Promise callback');
});

console.log('4. Script end');

// Output:
// 1. Script start
// 4. Script end
// 3. Promise callback (microtask)
// 2. setTimeout callback (macrotask)
```

Q1.3: How does Node.js handle I/O operations differently from multi-threaded servers? [Mid]

Node.js uses **non-blocking, asynchronous I/O** operations. When you perform an I/O operation (reading a file, querying a database, making an HTTP request), Node.js doesn't wait for it to complete. Instead, it registers a callback and continues executing other code. When the I/O operation finishes, the callback is added to the event queue and executed.

This is fundamentally different from multi-threaded servers where each I/O operation blocks its thread until completion. In Node.js, a single thread can initiate thousands of I/O operations simultaneously, with the operating system's kernel handling the actual work through mechanisms like `epoll` (Linux), `kqueue` (macOS), or `IOCP` (Windows).

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

// Non-blocking (asynchronous) - Node.js way
fs.readFile('large-file.txt', (err, data) => {
  if (err) throw err;
  console.log('File read complete');
});
console.log('This runs immediately, not waiting for file');

// Blocking (synchronous) - avoid in production
const data = fs.readFileSync('large-file.txt');
console.log('This waits until file is read');
```

> **Common Mistake**
>
> Never use synchronous I/O methods (like `readFileSync`) in production server code. They block the entire event loop, preventing Node.js from handling other requests until the operation completes.

Q1.4: What is the difference between the V8 engine and Node.js? [Junior]

V8 is Google's open-source JavaScript engine written in C++. It compiles JavaScript directly to native machine code for fast execution. V8 is used in Chrome browser and provides the core JavaScript execution capabilities.

Node.js is a runtime environment that wraps V8 and extends it with additional APIs for server-side programming. Node.js adds capabilities that don't exist in browser JavaScript: file system access, network operations, process management, and more. It also includes `libuv` for the event loop and asynchronous I/O, plus a standard library of modules.

```
// V8 provides core JavaScript
const arr = [1, 2, 3].map(x => x * 2);
const obj = { ...defaults, ...overrides };

// Node.js adds server-side APIs
const fs = require('fs');       // File system
const http = require('http');   // HTTP server
const path = require('path');   // Path utilities
const crypto = require('crypto'); // Cryptography

// Node.js globals not in browsers
console.log(process.env.NODE_ENV);
console.log(__dirname);
console.log(__filename);
```

Q1.5: What is libuv and what role does it play in Node.js? [Mid]

libuv is a multi-platform C library that provides Node.js with its event loop and asynchronous I/O capabilities. It abstracts operating system differences and provides a consistent API for non-blocking operations across Windows, Linux, and macOS.

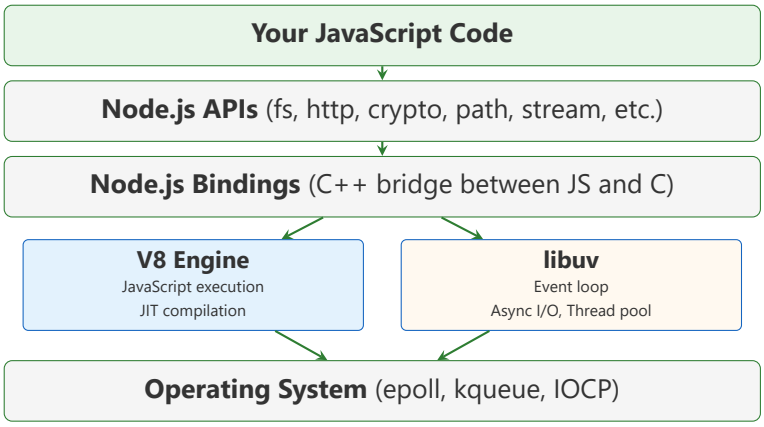
libuv handles file system operations, DNS resolution, network operations, child processes, and thread pool management. When Node.js needs to perform an async operation that the OS kernel can't handle asynchronously (like file system operations on some platforms), libuv uses a thread pool (default 4 threads) to perform the work without blocking the main thread.

```
// libuv thread pool handles these operations

// File system operations use the thread pool
const fs = require('fs');
fs.readFile('file.txt', callback); // Thread pool

// DNS lookups use the thread pool
const dns = require('dns');
dns.lookup('example.com', callback); // Thread pool

// Crypto operations use the thread pool
const crypto = require('crypto');
crypto.pbkdf2('password', 'salt', 100000, 64, 'sha512', callback);

// Network I/O uses OS kernel (epoll/kqueue/IOCP)
const http = require('http');
http.get('http://example.com', callback); // Kernel async
```

Pro Tip

You can increase the thread pool size by setting the `UV_THREADPOOL_SIZE` environment variable (max 1024). This can help if your application performs many concurrent file system or crypto operations.

Node.js Architecture



Q1.6: What is the global object in Node.js? [Junior]

In Node.js, the **global** object is the top-level object that holds global variables and functions, similar to `window` in browsers. However, variables declared with `var`, `let`, or `const` at the top level of a module are **not** added to `global`—they're scoped to that module.

Node.js also provides several global-like objects that are available in every module without requiring imports: `process`, `console`, `Buffer`, `__dirname`, `__filename`, `require()`, `module`, and `exports`. Note that `__dirname` and `__filename` are module-scoped, not truly global.

```
// Accessing the global object
console.log(global === globalThis); // true in Node.js

// These are available globally
console.log(process.version);  // Node.js version
console.log(process.platform); // 'linux', 'darwin', 'win32'

// Module-scoped "globals" (not on global object)
console.log(__dirname);  // Directory of current file
console.log(__filename); // Full path of current file

// Variables don't become global automatically
var myVar = 'test';
console.log(global.myVar); // undefined

// Explicitly add to global (not recommended)
global.myGlobalVar = 'available everywhere';
```

> **Common Mistake**
> Avoid adding properties to the `global` object. It creates implicit dependencies, makes testing harder, and can lead to naming collisions. Use modules and explicit imports instead.

1.2 Event Loop and Asynchronous Programming

Q1.7: Explain the phases of the Node.js event loop. [Mid]

The Node.js event loop has **six main phases**, each with a FIFO queue of callbacks to execute:

1. Timers: Executes callbacks scheduled by `setTimeout()` and `setInterval()`.

2. Pending callbacks: Executes I/O callbacks deferred from the previous loop iteration (like TCP errors).

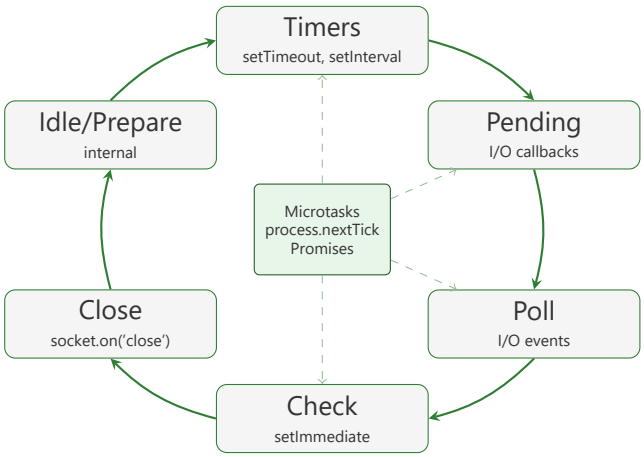
3. Idle, prepare: Internal use only by Node.js.

4. Poll: Retrieves new I/O events and executes I/O-related callbacks. This is where Node.js spends most of its time.

5. Check: Executes `setImmediate()` callbacks.

6. Close callbacks: Executes close event callbacks (like `socket.on('close')`).

Between each phase, Node.js processes the **microtask queue** (Promises, `process.nextTick()`).




```
// Demonstrating event loop phases
const fs = require('fs');

setTimeout(() => console.log('1. Timer'), 0);
setImmediate(() => console.log('2. Immediate'));

fs.readFile(__filename, () => {
  // Inside I/O callback, setImmediate always runs first
  setTimeout(() => console.log('3. Timer in I/O'), 0);
  setImmediate(() => console.log('4. Immediate in I/O'));
});

process.nextTick(() => console.log('5. nextTick'));
Promise.resolve().then(() => console.log('6. Promise'));

console.log('7. Sync');

// Output: 7, 5, 6, 1 or 2 (order varies), 2 or 1, 4, 3
```

Q1.8: What is the difference between process.nextTick() and setImmediate()? [Mid]

`process.nextTick()` and `setImmediate()` both schedule callbacks, but execute at different times in the event loop.

`process.nextTick()` fires **immediately after the current operation**, before the event loop continues. It's processed after each phase of the event loop, in the microtask queue. This makes it higher priority than any I/O or timer callbacks.

`setImmediate()` executes in the **Check phase** of the event loop, after the Poll phase completes. Inside an I/O callback, `setImmediate()` always executes before `setTimeout(..., 0)`.

```
// nextTick vs setImmediate
setImmediate(() => console.log('setImmediate'));
process.nextTick(() => console.log('nextTick'));
console.log('sync');

// Output: sync, nextTick, setImmediate

// Recursive nextTick can starve the event loop!
function badRecursion() {
  process.nextTick(badRecursion); // Never lets I/O happen
}

// setImmediate allows I/O between iterations
function goodRecursion() {
  setImmediate(goodRecursion); // I/O can happen
}
```

> **Common Mistake**
>
> Excessive use of `process.nextTick()` can starve the event loop, preventing I/O callbacks from executing. Prefer `setImmediate()` for recursive operations to allow the event loop to process other events.

Q1.9: What is callback hell and how do you avoid it? [Junior]

Callback hell (also called the "pyramid of doom") occurs when multiple asynchronous operations are nested inside each other, creating deeply indented, hard-to-read code. Each async operation requires a callback, and when operations depend on previous results, callbacks get nested deeper and deeper.

Solutions include: **1)** Named functions instead of anonymous callbacks, **2)** Promises with `.then()` chains, **3)** `async/await` syntax for linear-looking async code, **4)** Control flow libraries like `async.js`, and **5)** Breaking code into smaller, modular functions.

```
// Callback hell - hard to read and maintain
fs.readFile('file1.txt', (err, data1) => {
  if (err) return handleError(err);
  fs.readFile('file2.txt', (err, data2) => {
    if (err) return handleError(err);
    db.query(data1 + data2, (err, result) => {
      if (err) return handleError(err);
      // More nesting...
    });
  });
});

// Solution: async/await
async function processFiles() {
  try {
```

```
    const data1 = await fs.promises.readFile('file1.txt');
    const data2 = await fs.promises.readFile('file2.txt');
    const result = await db.query(data1 + data2);
    return result;
  } catch (err) {
    handleError(err);
  }
}
```

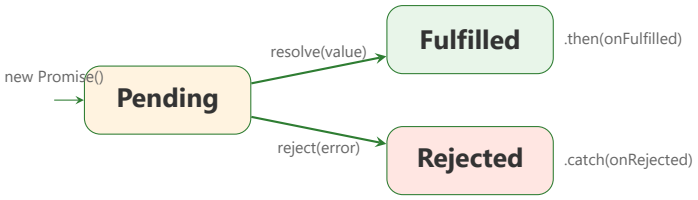
Q1.10: What are Promises and how do they improve async code? [Junior]

A **Promise** is an object representing the eventual completion or failure of an asynchronous operation. It can be in one of three states: **pending** (initial state), **fulfilled** (operation completed successfully), or **rejected** (operation failed).

Promises improve code by enabling **chaining** with `.then()` and `.catch()`, avoiding nested callbacks. They provide better error handling (errors propagate through the chain), and enable `Promise.all()` for parallel operations and `Promise.race()` for competitive execution. Promises are also the foundation for `async/await` syntax.

```
// Creating a Promise
function fetchUser(id) {
  return new Promise((resolve, reject) => {
    db.query(`SELECT * FROM users WHERE id = ${id}`, (err, user) => {
      if (err) reject(err);
      else resolve(user);
    });
  });
}

// Chaining promises
fetchUser(1)
  .then(user => fetchOrders(user.id))
  .then(orders => processOrders(orders))
  .catch(err => console.error('Error:', err));

// Parallel execution
Promise.all([
  fetchUser(1),
  fetchUser(2),
  fetchUser(3)
]).then(users => console.log('All users:', users));

// First to complete wins
Promise.race([
  fetchFromCache(key),
  fetchFromDatabase(key)
]).then(result => console.log('Fastest result:', result));
```

Promise States



Q1.11: How does async/await simplify asynchronous code? [Junior]

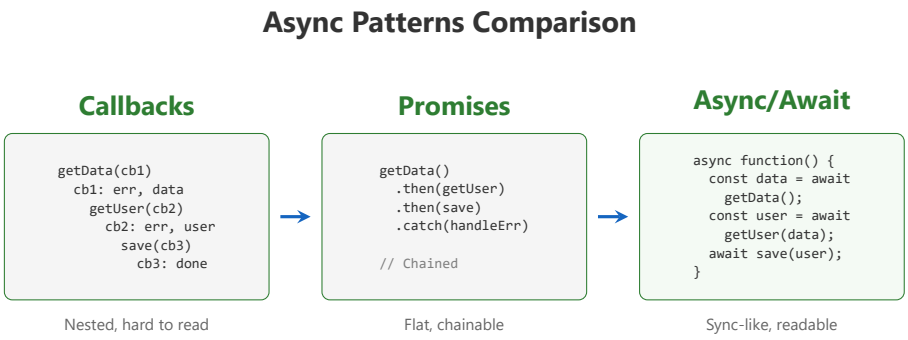
`async/await` is syntactic sugar over Promises that makes asynchronous code look and behave like synchronous code. An `async` function always returns a Promise, and the `await` keyword pauses execution until the Promise resolves, returning its value.

This syntax eliminates `.then()` chains and allows using standard control flow statements (`if`, `for`, `try/catch`) with async operations. Error handling becomes intuitive with `try/catch` blocks instead of `.catch()` callbacks. The code reads top-to-bottom, making it easier to understand and debug.

```
// async function declaration
async function getUserWithOrders(userId) {
  try {
    const user = await fetchUser(userId);
    const orders = await fetchOrders(user.id);

    // Can use regular loops with await
    for (const order of orders) {
      order.details = await fetchOrderDetails(order.id);
    }

    return { user, orders };
  } catch (error) {
    console.error('Failed to fetch data:', error);
    throw error;
  }
}

// Parallel await with Promise.all
async function fetchAllUsers(ids) {
  const promises = ids.map(id => fetchUser(id));
  const users = await Promise.all(promises);
  return users;
}

// Arrow function with async
const getUser = async (id) => {
  return await fetchUser(id);
};
```

> **Pro Tip**
> Avoid sequential `await` when operations can run in parallel. Instead of awaiting each op-

eration one by one, use `Promise.all()` to run them concurrently and await the combined result.



Q1.12: What is the difference between parallel and sequential execution of async operations? [Mid]

Sequential execution runs async operations one after another, waiting for each to complete before starting the next. Total time equals the sum of all operation times. Use this when operations depend on previous results.

Parallel execution starts all operations simultaneously and waits for all to complete. Total time equals the longest operation. Use this when operations are independent. In Node.js, use `Promise.all()` for parallel execution with `async/await`.

```
// Sequential - takes 3 seconds total
async function sequential() {
  const a = await fetchA(); // 1 second
  const b = await fetchB(); // 1 second
  const c = await fetchC(); // 1 second
  return [a, b, c];
}

// Parallel - takes 1 second total
async function parallel() {
  const [a, b, c] = await Promise.all([
    fetchA(), // 1 second
    fetchB(), // 1 second
    fetchC()  // 1 second
  ]);
  return [a, b, c];
}

// Sequential needed when operations depend on each other
async function dependent() {
  const user = await fetchUser(1);
  const orders = await fetchOrders(user.id); // Needs user.id
  return { user, orders };
}
```

Common Mistake

With `Promise.all()`, if any promise rejects, the entire operation fails immediately. Use `Promise.allSettled()` if you want to wait for all promises regardless of success or failure.

1.3 Modules and Package Management

Q1.13: What are the key differences between CommonJS and ES Modules? [Mid]

CommonJS (CJS) is Node.js's original module system using `require()` and `module.exports`. It loads modules **synchronously** and at **runtime**, meaning you can use dynamic paths and conditional imports.

ES Modules (ESM) use `import` and `export` syntax. They are loaded **asynchronously** and **statically analyzed** at parse time, enabling tree-shaking and better optimization. ESM is the JavaScript standard and works in both browsers and Node.js. ESM imports are hoisted and must be at the top level (no dynamic imports in regular syntax, though `import()` function exists for dynamic loading).

```
// CommonJS (file.cjs or default in Node.js)
const fs = require('fs');
const { readFile } = require('fs');

// Dynamic require (works)
const moduleName = condition ? 'moduleA' : 'moduleB';
const mod = require(moduleName);

module.exports = { myFunction };
module.exports.namedExport = value;

// ES Modules (file.mjs or "type": "module" in package.json)
import fs from 'fs';
import { readFile } from 'fs';
import * as fsAll from 'fs';

// Dynamic import (async)
const mod = await import('./module.js');

export const myFunction = () => {};
export default myClass;
```

Pro Tip

To use ES Modules in Node.js, either name your files with `.mjs` extension or add `"type": "module"` to your `package.json`. Modern Node.js projects increasingly prefer ESM for better compatibility with frontend code and tooling.

Q1.14: What is npm and why is it vital for Node.js development? [Junior]

npm (Node Package Manager) is the default package manager for Node.js and the world's largest software registry. It serves three purposes: **1)** A command-line tool for installing, updating, and managing project dependencies, **2)** An online registry hosting over 2 million open-source packages, and **3)** A website for discovering and documenting packages.

npm handles dependency resolution, version management, and script running. It installs packages locally in `node_modules` or globally for CLI tools. The `package.json` file defines your project's dependencies, scripts, and metadata, making projects reproducible across different machines.

```
# Initialize a new project
npm init -y

# Install dependencies
npm install express           # Production dependency
npm install jest --save-dev   # Development dependency
npm install -g typescript     # Global installation

# Other common commands
npm update                    # Update packages
npm outdated                  # Check for outdated packages
npm audit                     # Security vulnerability scan
npm run test                  # Run scripts from package.json
npm ci                        # Clean install from lock file
```

Q1.15: What is the purpose of the package.json file? [Junior]

`package.json` is the **manifest file** for Node.js projects. It contains project metadata (name, version, description), lists all dependencies with their version constraints, defines npm scripts for common tasks, and specifies the project's entry point.

This file makes projects reproducible—anyone can clone the repository, run `npm install`, and get the same dependencies. It also enables npm to resolve dependency trees, run scripts, and publish packages to the registry.

```
{
  "name": "my-api",
  "version": "1.0.0",
  "description": "A REST API built with Express",
  "main": "src/index.js",
  "type": "module",
  "scripts": {
    "start": "node src/index.js",
    "dev": "nodemon src/index.js",
    "test": "jest",
    "lint": "eslint src/"
  },
  "dependencies": {
    "express": "^4.18.2",
```

```
    "pg": "^8.11.0"
  },
  "devDependencies": {
    "jest": "^29.5.0",
    "nodemon": "^3.0.1"
  },
  "engines": {
    "node": ">=18.0.0"
  }
}
```

Q1.16: What is the difference between dependencies and devDependencies? [Junior]

dependencies are packages required for your application to run in production. They include frameworks (Express), database drivers (pg, mongoose), and utility libraries your code directly uses.

devDependencies are packages only needed during development and testing. They include testing frameworks (Jest), linters (ESLint), build tools (TypeScript, webpack), and development servers (nodemon). When deploying to production with `npm install --production` or `npm ci --omit=dev`, devDependencies are not installed, reducing the production bundle size.

```
# Install as production dependency
npm install express mongoose

# Install as dev dependency
npm install --save-dev jest eslint typescript

# Production install (omits devDependencies)
npm install --production
npm ci --omit=dev

// package.json
{
  "dependencies": {
    "express": "^4.18.2",   // Needed in production
    "mongoose": "^7.0.0"
  },
  "devDependencies": {
    "jest": "^29.5.0",      // Only for testing
    "eslint": "^8.40.0",    // Only for development
    "typescript": "^5.0.0"  // Only for building
  }
}
```

Q1.17: What is package-lock.json and why is it important? [Junior]

`package-lock.json` is an automatically generated file that locks the **exact versions** of all installed packages and their dependencies. While `package.json` specifies version ranges (like `^4.18.0`), the lock file records the precise version installed (like `4.18.2`).

This ensures **reproducible builds**—every developer and CI/CD pipeline gets identical dependencies. Without it, two `npm install` commands at different times could install different versions, potentially causing "works on my machine" bugs. Always commit `package-lock.json` to version control.

```
// package.json - version range
"express": "^4.18.0"  // Could install 4.18.0, 4.18.2, 4.19.0...

// package-lock.json - exact version
"express": {
  "version": "4.18.2",
  "resolved": "https://registry.npmjs.org/express/-/...",
  "integrity": "sha512-...",
  "dependencies": {
    "accepts": "~1.3.8",
    ...
  }
}

# Use npm ci for clean installs from lock file
npm ci  # Deletes node_modules, installs exact versions

# npm install updates lock file if needed
npm install  # May update lock file with newer versions
```

> **Pro Tip**
> Use `npm ci` instead of `npm install` in CI/CD pipelines. It's faster, more reliable, and ensures the exact versions from `package-lock.json` are installed without any modifications.

Q1.18: What are peer dependencies and when would you use them? [Mid]

Peer dependencies specify packages that your package is compatible with but doesn't directly depend on. They're used primarily by libraries and plugins that extend another package. Instead of bundling the host package, a peer dependency says "I work with version X of this package, and the consuming project must provide it."

For example, a React component library declares `react` as a peer dependency because it needs React to work but shouldn't bundle its own copy. This prevents version conflicts and duplicate packages. If the consuming project has an incompatible version, npm warns during installation.

```
// A React component library's package.json
{
```

```
  "name": "my-react-components",
  "peerDependencies": {
    "react": "^17.0.0 || ^18.0.0",
    "react-dom": "^17.0.0 || ^18.0.0"
  },
  "devDependencies": {
    // For development, install peer deps as dev deps
    "react": "^18.2.0",
    "react-dom": "^18.2.0"
  }
}

// A Jest plugin's package.json
{
  "name": "jest-custom-reporter",
  "peerDependencies": {
    "jest": ">=28.0.0"
  }
}
```

Q1.19: What is npm audit and how do you fix security vulnerabilities? [Junior]

`npm audit` scans your project's dependency tree for known security vulnerabilities by checking against the npm security advisory database. It reports vulnerabilities with severity levels (low, moderate, high, critical) and provides remediation advice.

To fix vulnerabilities: **1)** Run `npm audit fix` for automatic fixes that don't break semver, **2)** Use `npm audit fix --force` for breaking changes (review carefully), **3)** Manually update specific packages, or **4)** Use overrides in `package.json` to force specific versions of transitive dependencies.

```
# Run security audit
npm audit

# Output shows vulnerabilities
# High: prototype-pollution in lodash < 4.17.21
# Moderate: regex-dos in some-package

# Automatic fix (safe)
npm audit fix

# Force fix (may break things)
npm audit fix --force

# Override transitive dependency in package.json
{
  "overrides": {
    "lodash": "^4.17.21"
  }
}
```

```
# Generate JSON report for CI
npm audit --json > audit-report.json
```

Common Mistake

`npm audit fix --force` can install major version updates that break your application. Always review the changes and run your test suite after fixing vulnerabilities.

1.4 Built-in Modules

Q1.20: What are the most important built-in modules in Node.js? [Junior]

Node.js includes many built-in modules that don't require installation:

fs - File system operations (read, write, watch files)

path - File path manipulation (join, resolve, parse paths)

http/https - HTTP server and client functionality

crypto - Cryptographic operations (hashing, encryption)

stream - Abstract interface for streaming data

os - Operating system information

events - Event emitter pattern implementation

util - Utility functions (promisify, format, inspect)

buffer - Binary data handling

child_process - Spawn child processes

```
const fs = require('fs');
const path = require('path');
const crypto = require('crypto');
const os = require('os');

// Path manipulation
const fullPath = path.join(__dirname, 'data', 'file.json');

// File operations
const data = fs.readFileSync(fullPath, 'utf8');

// Hashing
const hash = crypto.createHash('sha256')
  .update(data)
  .digest('hex');
```

```
// System info
console.log(os.platform());  // 'linux', 'darwin', 'win32'
console.log(os.cpus().length); // Number of CPU cores
console.log(os.freemem());    // Free memory in bytes
```

Q1.21: How do you read and write files synchronously vs asynchronously? [Junior]

Node.js `fs` module provides three APIs: **synchronous** (blocking), **callback-based** (async), and **promise-based** (async). Synchronous methods have `Sync` suffix and block the event loop until complete. Callback methods take an error-first callback. Promise methods are in `fs.promises` namespace and work with `async/await`.

Use synchronous methods only at startup or in CLI tools. For servers, always use asynchronous methods to avoid blocking other requests.

```
const fs = require('fs');
const fsPromises = require('fs/promises');

// Synchronous - blocks event loop
const data = fs.readFileSync('file.txt', 'utf8');
fs.writeFileSync('output.txt', data);

// Callback-based async
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  fs.writeFile('output.txt', data, (err) => {
    if (err) throw err;
    console.log('Done!');
  });
});

// Promise-based async (recommended)
async function copyFile() {
  const data = await fsPromises.readFile('file.txt', 'utf8');
  await fsPromises.writeFile('output.txt', data);
  console.log('Done!');
}
```

Q1.22: What are streams in Node.js and what types exist? [Mid]

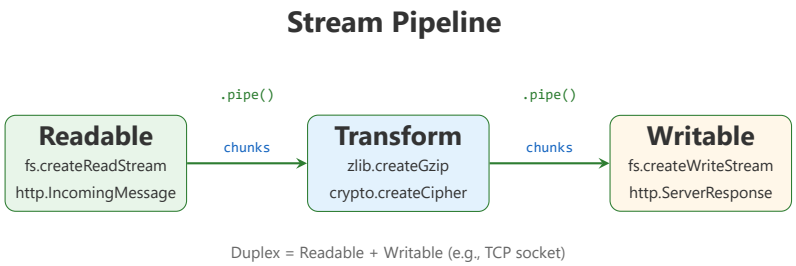
Streams are **abstract interfaces** for handling data that comes in chunks over time, rather than loading everything into memory at once. They're essential for processing large files, network data, or any data too big to fit in memory.

Four types of streams exist: **Readable** (source of data, e.g., `fs.createReadStream`), **Writable** (destination for data, e.g., `fs.createWriteStream`), **Duplex** (both readable and writable, e.g., TCP sockets), and **Transform** (duplex that modifies data passing through, e.g., `zlib.createGzip`).

```
const fs = require('fs');
const zlib = require('zlib');

// Readable stream
const readable = fs.createReadStream('large-file.txt');

// Writable stream
const writable = fs.createWriteStream('output.txt');

// Transform stream (gzip compression)
const gzip = zlib.createGzip();

// Pipe: read -> compress -> write
readable
  .pipe(gzip)
  .pipe(writable)
  .on('finish', () => console.log('Compression complete'));

// Reading stream data manually
readable.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes`);
});
readable.on('end', () => console.log('Done reading'));
readable.on('error', (err) => console.error(err));
```

Pro Tip

Use the `pipeline()` function from `stream` module instead of chaining `.pipe()` calls. It properly handles errors and cleanup, preventing memory leaks when streams fail.



Q1.23: What is the Buffer class and when would you use it? [Mid]

Buffer is a class for handling raw binary data directly in memory, outside the V8 heap. Since JavaScript strings are Unicode and can't represent arbitrary binary data efficiently, Buffers provide a way to work with binary data from files, network packets, or encryption operations.

Use Buffers when working with: file I/O (especially binary files), network protocols, cryptographic operations, image/audio processing, or converting between encodings. Buffers have a fixed size determined at creation and can be sliced, copied, and concatenated.

```
// Creating buffers
const buf1 = Buffer.alloc(10);           // 10 zero bytes
const buf2 = Buffer.from('Hello');       // From string
const buf3 = Buffer.from([72, 105]);     // From array

// Reading and writing
buf1.write('Hi');
console.log(buf1.toString());  // 'Hi'

// Encoding conversions
const base64 = Buffer.from('Hello').toString('base64');
console.log(base64);  // 'SGVsbG8='

const original = Buffer.from(base64, 'base64').toString('utf8');
console.log(original);  // 'Hello'

// Working with binary files
const fs = require('fs');
const imageBuffer = fs.readFileSync('image.png');
console.log(imageBuffer.length);  // File size in bytes
```

Q1.24: How do you handle environment variables in Node.js? [Junior]

Environment variables are accessed through the `process.env` object, which contains all environment variables as string key-value pairs. They're commonly used for configuration that varies between environments (development, staging, production) like database URLs, API keys, and feature flags.

For local development, use a `.env` file with the `dotenv` package to load variables. Never commit `.env` files with secrets to version control—use `.env.example` to document required variables without actual values.

```
// Access environment variables
const port = process.env.PORT || 3000;
const dbUrl = process.env.DATABASE_URL;
const nodeEnv = process.env.NODE_ENV;

// Using dotenv package
require('dotenv').config();

// .env file
// PORT=3000
// DATABASE_URL=postgres://localhost/mydb
// API_KEY=secret123

// Type-safe config pattern
const config = {
  port: parseInt(process.env.PORT, 10) || 3000,
  db: {
    url: process.env.DATABASE_URL,
```

```
    pool: parseInt(process.env.DB_POOL_SIZE, 10) || 10
  },
  isDev: process.env.NODE_ENV === 'development'
};

// Validate required variables
if (!process.env.DATABASE_URL) {
  throw new Error('DATABASE_URL is required');
}
```

> **Common Mistake**
> All environment variables are strings. Always parse numeric values with `parseInt()` or `parseFloat()`, and compare boolean values as strings (`process.env.DEBUG === 'true'`).

1.5 Error Handling

Q1.25: How do you handle errors in synchronous vs asynchronous code? [Junior]

Synchronous errors are caught with standard `try/catch` blocks. The error propagates up the call stack until caught or crashes the application.

Asynchronous errors require different approaches: callbacks use the error-first pattern (error as first argument), Promises use `.catch()` or `try/catch` with `async/await`, and EventEmitters emit `'error'` events. Unhandled async errors can crash the process or be silently ignored, so proper error handling is critical.

```
// Synchronous error handling
try {
  const data = JSON.parse(invalidJson);
} catch (error) {
  console.error('Parse failed:', error.message);
}

// Callback error handling (error-first pattern)
fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Read failed:', err);
    return;
  }
  console.log(data);
});

// Promise error handling
fetchData()
  .then(data => process(data))
  .catch(err => console.error('Failed:', err));
```

```
// async/await error handling
async function main() {
  try {
    const data = await fetchData();
    return process(data);
  } catch (error) {
    console.error('Failed:', error);
  }
}
```

Q1.26: What is the difference between operational errors and programmer errors? [Mid]

Operational errors are runtime problems that occur in correctly written programs: network failures, file not found, invalid user input, database timeouts. These are expected failure modes that your code should handle gracefully—retry, return an error response, or fail the specific operation.

Programmer errors are bugs in your code: typos, undefined variables, wrong function arguments, logic errors. These indicate your code is broken and usually shouldn't be caught—let the process crash and fix the bug. Trying to "handle" programmer errors often masks the real problem.

```
// Operational errors - handle gracefully
async function getUser(id) {
  try {
    const user = await db.findUser(id);
    if (!user) {
      // User not found - operational, expected
      throw new NotFoundError(`User ${id} not found`);
    }
    return user;
  } catch (error) {
    if (error.code === 'ECONNREFUSED') {
      // Database down - operational, retry or fail gracefully
      throw new ServiceUnavailableError('Database unavailable');
    }
    throw error;
  }
}

// Programmer errors - let them crash
function calculateTotal(items) {
  // If items is undefined, that's a bug - don't catch it
  return items.reduce((sum, item) => sum + item.price, 0);
}
```

> **Pro Tip**
> Use different error classes for operational vs programmer errors. Operational errors might extend a base `AppError` class with status codes, while programmer errors should crash the process so you can identify and fix them.

Q1.27: How do you handle uncaught exceptions and unhandled promise rejections? [Mid]

Node.js provides global handlers for catching errors that escape your code: `process.on('uncaughtException')` for synchronous errors and `process.on('unhandledRejection')` for Promise rejections without `.catch()`.

These should be used for logging and graceful shutdown, not for continuing execution. After an uncaught exception, the application state may be corrupted, so the safest approach is to log the error, close connections gracefully, and exit the process. Let a process manager (PM2, systemd, Kubernetes) restart it.

```
// Handle uncaught synchronous exceptions
process.on('uncaughtException', (error) => {
  console.error('Uncaught Exception:', error);
  // Log to external service
  logger.fatal(error);
  // Graceful shutdown
  server.close(() => {
    process.exit(1);
  });
});

// Handle unhandled promise rejections
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection:', reason);
  // In Node.js 15+, this crashes by default
  // Log and exit to be safe
  logger.fatal({ reason, promise });
  process.exit(1);
});

// Graceful shutdown on signals
process.on('SIGTERM', () => {
  console.log('SIGTERM received, shutting down...');
  server.close(() => {
    db.close().then(() => process.exit(0));
  });
});
```

> **Common Mistake**
> Never use `uncaughtException` to resume normal operation. The Node.js documentation explicitly warns against this—the application may be in an undefined state.

Q1.28: What is the error-first callback pattern? [Junior]

The **error-first callback** (also called "Node-style callback") is a convention where callback functions receive an error object as the first argument. If the operation succeeded, the error is `null` or `undefined`, and subsequent arguments contain the results. If it failed, the error argument contains the Error object.

This pattern ensures consistent error handling across all async operations in Node.js. Most built-in modules and npm packages follow this convention, making it predictable how to handle success and failure cases.

```
// Error-first callback pattern
fs.readFile('file.txt', (err, data) => {
  if (err) {
    // Handle error - first argument
    console.error('Failed to read:', err.message);
    return;
  }
  // Success - use data (second argument)
  console.log(data);
});

// Creating functions that follow the pattern
function fetchUser(id, callback) {
  db.query(`SELECT * FROM users WHERE id = ${id}`, (err, rows) => {
    if (err) {
      return callback(err, null);  // Error first
    }
    if (rows.length === 0) {
      return callback(new Error('User not found'), null);
    }
    callback(null, rows[0]);  // null error = success
  });
}

// Convert callback to Promise with util.promisify
const util = require('util');
const readFile = util.promisify(fs.readFile);
const data = await readFile('file.txt');
```

Q1.29: How do you create custom error classes in Node.js? [Mid]

Custom error classes extend the built-in `Error` class and add application-specific properties like HTTP status codes, error codes, or additional context. They help distinguish different error types, enable specific error handling, and provide better debugging information.

Create a base `AppError` class for operational errors, then extend it for specific cases like `NotFoundError`, `ValidationError`, or `UnauthorizedError`. This pattern integrates well with Express error handling middleware.

```
// Base application error
class AppError extends Error {
  constructor(message, statusCode, code) {
    super(message);
    this.statusCode = statusCode;
    this.code = code;
    this.isOperational = true;
    Error.captureStackTrace(this, this.constructor);
  }
}

// Specific error types
class NotFoundError extends AppError {
  constructor(resource) {
    super(`${resource} not found`, 404, 'NOT_FOUND');
  }
}

class ValidationError extends AppError {
  constructor(message, errors = []) {
    super(message, 400, 'VALIDATION_ERROR');
    this.errors = errors;
  }
}

// Usage
if (!user) {
  throw new NotFoundError('User');
}

// Express error handler can check error type
app.use((err, req, res, next) => {
  if (err.isOperational) {
    res.status(err.statusCode).json({ error: err.message });
  } else {
    res.status(500).json({ error: 'Internal server error' });
  }
});
```

Pro Tip

Call `Error.captureStackTrace(this, this.constructor)` in custom errors to exclude the constructor call from the stack trace, making debugging easier by pointing directly to where the error was thrown.

2

TypeScript for Backend

TypeScript has become the standard for professional Node.js development. It adds static typing to JavaScript, catching errors at compile time rather than runtime, improving code quality, and enabling better tooling. This chapter covers TypeScript fundamentals and advanced features essential for backend development.

2.1 Type System Basics

Q2.1: What is TypeScript and why use it for backend development? [Junior]

TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. It adds optional type annotations, interfaces, generics, and other features that help catch errors during development rather than at runtime.

For backend development, TypeScript provides: **1)** Early error detection through static type checking, **2)** Better IDE support with autocompletion and refactoring, **3)** Self-documenting code through type annotations, **4)** Easier maintenance of large codebases, and **5)** Safer refactoring with compile-time validation.

```
// JavaScript - errors only found at runtime
function getUser(id) {
  return db.query(`SELECT * FROM users WHERE id = ${id}`);
}
getUser('abc'); // No error until runtime

// TypeScript - errors caught at compile time
function getUser(id: number): Promise<User> {
  return db.query(`SELECT * FROM users WHERE id = ${id}`);
}
getUser('abc'); // Error: Argument of type 'string'
                // is not assignable to parameter of type 'number'
```

TypeScript Compilation Flow

FREE SAMPLE

Thank you for reading this free sample!

This sample includes:

- Complete Table of Contents
- Full chapter structure overview
- Entire first chapter with all questions & answers

The full book contains 200+ interview questions with detailed answers and code examples.

Get the complete book at:

easyinterview.me

© 2026 EasyInterview.me - All rights reserved