81 NodeJS Interview Questions and Answers

1. What is Node.js?

Answer: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that allows executing JavaScript code outside a web browser. It uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

Frequency: Very High (95%)
Companies: Almost all (Amazon, Google, Microsoft, Facebook, Netflix, PayPal)

2. Explain the event loop in Node.js

Answer: The event loop is a mechanism that allows Node.js to perform non-blocking I/O operations despite JavaScript being single-threaded. It works by offloading operations to the system kernel whenever possible and executing callbacks when operations complete.

Frequency: Very High (90%)
Companies: Microsoft, Amazon, Netflix, Uber, PayPal

```
// Example demonstrating the event loop
console.log('Start');

setTimeout(() => {
  console.log('Timeout callback'); // This will execute after the main tl
}, 0);

Promise.resolve().then(() => {
  console.log('Promise resolved'); // This goes to microtask queue
});

console.log('End');

// Output:
// Start
// End
// Promise resolved
// Timeout callback
```

3. What are the advantages of using Node.js?

Answer: - Single programming language (JavaScript) for both client and server - Non-blocking I/O for handling multiple requests - High performance due to V8 engine - Large ecosystem with npm - Great for real-time applications - Perfect for microservices architecture

Frequency: High (80%)
Companies: Walmart, LinkedIn, Netflix, PayPal

4. What is npm?

Answer: npm (Node Package Manager) is the default package manager for Node.js. It consists of a command-line client and an online database of packages called the npm registry that enables sharing and reusing code.

Frequency: High (80%)
Companies: All companies using Node.js

5. Explain the difference between package.json and package-lock.json

Answer: - package.json: Records project metadata and defines dependencies with version ranges - package-lock.json: Locks down exact versions of dependencies and their dependencies, ensuring consistent installs across environments

Frequency: High (75%)
Companies: Amazon, Microsoft, Netflix, Stripe

6. What is middleware in Express.js?

Answer: Middleware functions are functions that have access to the request object, response object, and the next middleware function in the application's request-response cycle. They can execute code, modify request/response objects, end the request-response cycle, or call the next middleware.

Frequency: High (70%)
Companies: Facebook, Netflix, Uber, PayPal

```
// Express middleware example
app.use((req, res, next) => {
  console.log('Time:', Date.now()); // Logs timestamp for every request
  next(); // Calls the next middleware
});
```

7. What is the difference between `process.nextTick()` and `setImmediate()`?

Answer: - `process.nextTick()` : Executes callback before the next event loop iteration, immediately after the current operation completes - `setImmediate()` : Executes callback in the next event loop iteration (after I/O events)

Frequency: Medium (60%)
Companies: Netflix, Uber, PayPal, Microsoft

```
// Example showing the difference
console.log('Start');

setImmediate(() => {
  console.log('setImmediate');
});

process.nextTick(() => {
  console.log('nextTick');
});

console.log('End');

// Output:
// Start
// End
// nextTick
// setImmediate
```

8. What are streams in Node.js?

Answer: Streams are objects that let you read or write data continuously. They solve the problem of handling large amounts of data efficiently without loading everything into memory. Node.js provides four types of streams: Readable, Writable, Duplex, and Transform.

Frequency: High (75%)
Companies: Netflix, PayPal, Uber, Airbnb

```
// Stream example - reading a large file
const fs = require('fs');
const readStream = fs.createReadStream('largefile.txt');
readStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data`);
});
readStream.on('end', () => {
  console.log('Finished reading file');
});
```

9. How does Node.js handle child threads or processes?

Answer: Node.js provides the `child_process` module to create child processes. For threading, it offers the `worker_threads` module (introduced in Node.js 10) for CPU-intensive tasks that would otherwise block the main thread.

Frequency: Medium (50%)
Companies: Netflix, Uber, Microsoft, LinkedIn

```
// Using worker threads
const { Worker, isMainThread, parentPort } = require('worker_threads');

if (isMainThread) {
  // Main thread creates a worker
  const worker = new Worker(__filename);
  worker.on('message', (msg) => {
    console.log(`Worker result: ${msg}`);
  });
  worker.postMessage('Start processing');
} else {
  // Worker thread logic
  parentPort.on('message', (msg) => {
    // CPU intensive work here
    parentPort.postMessage('Work completed!');
  });
}
```

10. Explain the concept of callback hell and how to avoid it

Answer: Callback hell (pyramid of doom) refers to deeply nested callbacks that make code hard to read and maintain. To avoid it: - Use Promises - Use async/await - Modularize code - Use named functions instead of anonymous ones - Use libraries like async.js

Frequency: High (70%)
Companies: Google, Netflix, Uber, Airbnb

```
// Callback hell example
getData(function(a) {
  getMoreData(a, function(b) {
    getMoreData(b, function(c) {
      getMoreData(c, function(d) {
        // Work with data
      });
    });
  });
});

// Solution with async/await
async function getAllData() {
  const a = await getData();
  const b = await getMoreData(a);
  const c = await getMoreData(b);
  const d = await getMoreData(c);
  // Work with data
}
```

11. What is the purpose of module.exports and require in Node.js?

Answer: `module.exports` is used to expose functions, objects, or values from a module, while `require()` is used to import functionality from other modules. They form the module system in Node.js.

Frequency: High (75%)
Companies: All companies using Node.js

```
// In math.js
module.exports.add = (a, b) => a + b;
module.exports.subtract = (a, b) => a - b;

// In app.js
const math = require('./math');
console.log(math.add(5, 3)); // 8
```

12. What are the differences between Node.js and browser JavaScript?

Answer: - Node.js has no DOM/window objects while browsers do - Node.js can access the filesystem, browsers have limited access - Node.js uses CommonJS module system, browsers use ES modules (though this is converging) - Node.js has global and process objects, browsers have window object - Node.js is server-side, browsers are client-side

Frequency: Medium (60%)
Companies: Microsoft, Netflix, PayPal, Twitter

13. Explain what is meant by "callback-first error handling"

Answer: In Node.js, callbacks typically follow the pattern where the first parameter is reserved for an error object. If an error occurred, the first parameter contains the error information; if not, it's null/undefined. This pattern is also called "error-first callbacks" or "Node.js error conventions."

Frequency: Medium (55%)
Companies: PayPal, Netflix, Uber

```
fs.readFile('file.txt', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  // Process data if no error occurred
  console.log(data);
});
```

14. What is EventEmitter in Node.js?

Answer: EventEmitter is a class in Node.js that facilitates communication between objects. It implements the observer pattern and is the foundation of many Node.js components that need to notify subscribers when events happen.

Frequency: High (70%)
Companies: Netflix, PayPal, Microsoft, Uber

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// Register listener
myEmitter.on('event', (arg) => {
  console.log('Event triggered with:', arg);
});

// Emit event
myEmitter.emit('event', 'some data');
```

15. How do you debug a Node.js application?

Answer: Node.js can be debugged using: - Built-in debugger with `node inspect` - Chrome DevTools (--inspect flag) - VS Code debugger - Libraries like debug or winston for logging - Node.js profiler

Frequency: Medium (50%)
Companies: Microsoft, Amazon, Google, Airbnb

16. What is cluster module in Node.js?

Answer: The cluster module allows creating child processes (workers) that share server ports, enabling Node.js applications to utilize multi-core systems and improve performance. It helps in load balancing requests across multiple CPU cores.

Frequency: Medium (45%)
Companies: Netflix, PayPal, LinkedIn, Uber

```
const cluster = require('cluster');
const http = require('http');
const numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers
  for (let i = 0; i < numCPUs; i++) {
    cluster.fork();
  }
  cluster.on('exit', (worker) => {
    console.log(`Worker ${worker.process.pid} died`);
    cluster.fork(); // Replace dead worker
  });
} else {
  // Workers share the TCP connection
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello from worker ' + process.pid);
  }).listen(8000);
}
```

17. What is the difference between setTimeout and setInterval?

Answer: - `setTimeout` : Executes a function once after a specified delay - `setInterval` : Executes a function repeatedly with a fixed time delay between each call

Frequency: Medium (45%)
Companies: Google, Amazon, Netflix

```
// setTimeout example
setTimeout(() => {
  console.log('This runs once after 2 seconds');
}, 2000);

// setInterval example
const intervalId = setInterval(() => {
  console.log('This runs every 2 seconds');
}, 2000);

// Stop the interval after 10 seconds
setTimeout(() => clearInterval(intervalId), 10000);
```

18. What is the difference between asynchronous

and non-blocking?

Answer: Asynchronous means operations can occur concurrently, with results processed later. Non-blocking specifically refers to I/O operations that return immediately without waiting for data to be read/written. In Node.js, non-blocking I/O is implemented through asynchronous operations.

Frequency: Medium (40%)
Companies: Microsoft, Netflix, PayPal

19. What are Buffer objects in Node.js?

Answer: Buffers are objects in Node.js used to work with binary data directly. They store raw data similar to an array of integers but correspond to raw memory allocations outside the V8 heap, primarily used when dealing with TCP streams or file system operations.

Frequency: Medium (45%)
Companies: PayPal, Netflix, Uber

```
// Creating a buffer
const buf1 = Buffer.alloc(10); // Creates a buffer of 10 bytes filled wi
const buf2 = Buffer.from('Hello, world!'); // Creates a buffer from a st

// Converting buffer to string
console.log(buf2.toString()); // 'Hello, world!'

// Working with binary data
buf1[0] = 255; // Set the first byte to 255
console.log(buf1[0]); // 255
```

20. What's the difference between operational and programmer errors in Node.js?

Answer: - Operational errors: Runtime issues that are expected to happen occasionally (e.g., network failures, invalid user input) - Programmer errors: Bugs in code that should be fixed (e.g., trying to access a property of undefined)

Node.js error handling strategies differ based on error type.

Frequency: Low (30%)
Companies: Netflix, Airbnb, Microsoft

21. Explain how to implement authentication in Node.js

Answer: Authentication in Node.js can be implemented using: - Passport.js library with various strategies - JWT (JSON Web Tokens) - Sessions with express-session - OAuth integration - Custom authentication middleware

The choice depends on application requirements.

Frequency: High (65%)
Companies: Microsoft, Netflix, PayPal, Uber

```
// Simple JWT authentication example
const jwt = require('jsonwebtoken');
const express = require('express');
const app = express();
app.use(express.json());

// Login route
app.post('/login', (req, res) => {
  // Validate credentials (simplified)
  if (req.body.username === 'admin' && req.body.password === 'password')
    const token = jwt.sign({ username: req.body.username }, 'secret_key'
    res.json({ token });
  } else {
    res.status(401).json({ message: 'Authentication failed' });
  }
});

// Middleware to verify JWT
function verifyToken(req, res, next) {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(403).json({ message: 'No token provided'

  jwt.verify(token, 'secret_key', (err, decoded) => {
    if (err) return res.status(401).json({ message: 'Invalid token' });
    req.user = decoded;
    next();
  });
}

// Protected route
app.get('/protected', verifyToken, (req, res) => {
  res.json({ message: 'Protected data', user: req.user });
});
```

22. What is middleware chaining in Express.js?

Answer: Middleware chaining in Express.js is the process of applying multiple middleware functions to a route. Each middleware can execute code, modify the request/response objects, and either terminate the chain or pass control to the next middleware using the `next()` function.

Frequency: Medium (50%)
Companies: Facebook, Netflix, Uber, PayPal

```
app.get('/api/data',
  // Middleware 1: Authentication check
  (req, res, next) => {
    if (!req.headers.authorization) {
      return res.status(401).send('Not authenticated');
    }
    next(); // Pass control to next middleware
  },
  // Middleware 2: Authorization check
  (req, res, next) => {
    if (!hasPermission(req.user)) {
      return res.status(403).send('Not authorized');
    }
    next();
  },
  // Final handler
  (req, res) => {
    res.json({ data: 'Sensitive data' });
  }
);
```

23. How does the Node.js require resolution algorithm work?

Answer: Node.js resolves requires in this order: 1. Core modules (e.g., fs, http) 2. File modules (with exact path) 3. Directory as module (looks for package.json or index.js) 4. Node_modules folders (up the directory tree)

It caches modules on first load for better performance.

Frequency: Medium (40%)
Companies: Microsoft, Netflix, PayPal

24. What are the main security concerns in a Node.js application?

Answer: Main security concerns include: - SQL/NoSQL injection attacks - Cross-Site Scripting (XSS) - Cross-Site Request Forgery (CSRF) - Broken authentication - Sensitive data exposure - Dependency vulnerabilities - DoS attacks

Frequency: High (65%)
Companies: PayPal, Uber, Netflix, Microsoft

25. What is the purpose of the 'use strict' directive in Node.js?

Answer: 'use strict' is a directive that enables strict mode, which: - Catches coding mistakes and throws exceptions - Prevents accidental globals - Eliminates this coercion - Disallows duplicate parameter names - Makes eval safer - Makes it impossible to delete variables and functions

Frequency: Medium (35%)
Companies: Microsoft, Google, Amazon

```
'use strict';

// This would throw an error in strict mode
x = 3.14; // ReferenceError: x is not defined

// This would throw an error (duplicate parameters)
function dupes(a, a, b) { // SyntaxError in strict mode
  return a + b;
}
```

26. How do you manage configuration in Node.js applications?

Answer: Configuration management typically involves: - Environment variables - Configuration files (JSON, YAML) - Libraries like dotenv, config, or convict - Cloud service configuration (AWS Parameter Store, etc.) - Different configs for development/production/testing

Frequency: Medium (45%)

Companies: Netflix, Uber, Microsoft, PayPal

```
// Using dotenv
// In .env file:
// DB_HOST=localhost
// DB_USER=admin
// DB_PASS=password

require('dotenv').config();

const dbConfig = {
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASS
};

// Connect to database using config
```

27. What are the differences between readFile and createReadStream in fs module?

Answer: - `readFile` : Reads entire file into memory at once, good for small files - `createReadStream` : Reads file in chunks without loading entire file in memory, better for large files

Frequency: Medium (50%)
Companies: Netflix, PayPal, Microsoft

```
// readFile example
const fs = require('fs');
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data); // All file content at once
});

// createReadStream example
const readStream = fs.createReadStream('largefile.txt', 'utf8');
readStream.on('data', (chunk) => {
  console.log(`Received chunk: ${chunk.length} bytes`);
});
readStream.on('end', () => {
  console.log('Finished reading file');
});
```

28. What is the purpose of process.env in Node.js?

Answer: `process.env` is an object containing the user environment variables. It's commonly used to store configuration information like database credentials, API keys, or to determine the running environment (development, test, production).

Frequency: High (75%)
Companies: All companies using Node.js

```
// Setting environment variables
// NODE_ENV=production node app.js

// Using environment variables in code
if (process.env.NODE_ENV === 'production') {
  console.log('Running in production mode');
} else {
  console.log('Running in development mode');
}

// Using for configuration
const dbUrl = process.env.DATABASE_URL || 'mongodb://localhost:27017/dev
```

29. How do you handle errors in Node.js applications?

Answer: Error handling in Node.js typically involves: - Try/catch blocks for synchronous code - Error-first callbacks for callback-based async code - .catch() for Promises - try/catch with async/await - Unhandled rejection and uncaught exception listeners - Domain module (deprecated) - Custom error classes

Frequency: High (75%)
Companies: Netflix, PayPal, Microsoft, Uber

```
// Synchronous error handling
try {
  const result = riskyOperation();
} catch (error) {
  console.error('Error:', error);
}

// Async with callbacks
fs.readFile('file.txt', (err, data) => {
  if (err) {
    return console.error('Error reading file:', err);
  }
  // Process data
});

// Promises
fetchData()
  .then(result => processData(result))
  .catch(error => console.error('Error fetching data:', error));

// Async/await
async function getData() {
  try {
    const result = await fetchData();
    return processData(result);
  } catch (error) {
    console.error('Error:', error);
    throw error; // Re-throw or handle
  }
}

// Global handlers
process.on('uncaughtException', (err) => {
  console.error('Uncaught exception:', err);
  // Graceful shutdown
  process.exit(1);
});

process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled rejection at:', promise, 'reason:', reason);
});
```

30. What are the key performance metrics to monitor in a Node.js application?

Answer: Key metrics include: - CPU usage - Memory usage (heap statistics) - Event loop lag - Request throughput and latency - Garbage collection frequency and duration - I/O operations - Error rates - Active handles and requests

Frequency: Medium (45%)
Companies: Netflix, PayPal, Uber, Airbnb

31. What is the Crypto module in Node.js?

Answer: The Crypto module provides cryptographic functionality including a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions. It's used for data encryption, hashing passwords, and creating digital signatures.

Frequency: Medium (40%)
Companies: PayPal, Stripe, Netflix

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

// Hashing a password
function hashPassword(password) {
  const salt = crypto.randomBytes(16).toString('hex');
  const hash = crypto.pbkdf2Sync(password, salt, 1000, 64, 'sha512').toS
  return { salt, hash };
}

// Verifying a password
function verifyPassword(password, salt, storedHash) {
  const hash = crypto.pbkdf2Sync(password, salt, 1000, 64, 'sha512').toS
  return storedHash === hash;
}

// Encrypting data
function encryptData(data, key) {
  const iv = crypto.randomBytes(16);
  const cipher = crypto.createCipheriv('aes-256-cbc', Buffer.from(key), 
  let encrypted = cipher.update(data);
  encrypted = Buffer.concat([encrypted, cipher.final()]);
  return { iv: iv.toString('hex'), encryptedData: encrypted.toString('he
}
```

32. How do you ensure high availability in a Node.js application?

Answer: High availability can be ensured by: - Load balancing multiple instances - Using the cluster module to utilize all CPU cores - Implementing health checks - Proper error handling and automatic recovery - Using process managers like PM2 - Containerization and orchestration (Docker, Kubernetes) - Database redundancy - Monitoring and alerts

Frequency: Medium (40%)
Companies: Netflix, Uber, PayPal, Airbnb

33. What are the differences between Promise and async/await?

Answer: Both handle asynchronous operations, but: - Syntax: async/await is more readable and similar to synchronous code - Error handling: try/catch in async/await vs .catch() in Promises - Promise chaining vs awaiting each step - async functions always return a Promise - async/await is built on top of Promises

Frequency: High (70%)
Companies: Google, Microsoft, Netflix, Uber

```
// Promise example
function fetchDataPromise() {
  return fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => {
      return processData(data);
    })
    .catch(error => {
      console.error('Error:', error);
      throw error;
    });
}

// Async/await equivalent
async function fetchDataAsync() {
  try {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    return processData(data);
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}
```

34. What is libuv and what role does it play in Node.js?

Answer: libuv is a multi-platform C library that provides support for asynchronous I/O based on event loops. In Node.js, it handles: - File system operations - Networking (TCP, UDP, DNS) - Child processes - Thread pool for offloading work - Event loop implementation - Timers and signals

Frequency: Medium (35%)
Companies: Netflix, Microsoft, PayPal

35. How would you implement pagination in a Node.js API?

Answer: Pagination can be implemented by: - Using limit and offset/skip parameters in database queries - Using cursor-based pagination (based on a specific field value) - Returning pagination metadata (total count, next/prev page links) - Handling edge cases like insufficient data - Implementing proper validation of pagination parameters

Frequency: Medium (50%)
Companies: Netflix, Uber, Facebook, Airbnb

```javascript
// Example with MongoDB and Express
app.get('/api/users', async (req, res) => {
  try {
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    // Get paginated results
    const users = await User.find()
      .skip(skip)
      .limit(limit)
      .sort({ createdAt: -1 });

    // Get total count for pagination metadata
    const total = await User.countDocuments();

    res.json({
      users,
      pagination: {
        currentPage: page,
        totalPages: Math.ceil(total / limit),
        totalItems: total,
        hasNextPage: page * limit < total,
        hasPrevPage: page > 1
      }
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

36. What is a closure in JavaScript and how is it used in Node.js?

Answer: A closure is when a function remembers and accesses variables from its outer scope even after that scope has finished executing. In Node.js, closures are used for: - Data privacy/encapsulation - Factory functions - Callback functions that need to access outer variables - Maintaining state between function calls

Frequency: Medium (55%)
Companies: Google, Microsoft, Netflix, PayPal

```
// Simple closure example
function createCounter() {
  let count = 0; // Private variable

  return {
    increment: () => {
      count++;
      return count;
    },
    decrement: () => {
      count--;
      return count;
    },
    getCount: () => count
  };
}

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.increment()); // 2
console.log(counter.decrement()); // 1
console.log(counter.getCount());  // 1
// 'count' is not directly accessible
```

37. How does garbage collection work in Node.js?

Answer: Node.js uses V8's garbage collection mechanism: - Uses mark-and-sweep algorithm - Has different GC types: Scavenge (minor), Mark-Sweep (major), Compaction - Objects are allocated in "young" or "old" generations - Performs GC in a stop-the-world manner (pauses execution) - Can be tuned with V8 flags

Frequency: Medium (30%)
Companies: Netflix, Microsoft, Google

38. What is the purpose of the util.promisify function?

Answer: `util.promisify` converts callback-based functions to return Promises. It takes a function following the common Node.js callback pattern (with (err, value) => {}) and returns a Promise-based version.

Frequency: Medium (45%)
Companies: Microsoft, Netflix, PayPal

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

// Convert callback-based readFile to return a Promise
const readFilePromise = util.promisify(fs.readFile);

// Now we can use it with async/await
async function readFileAsync() {
  try {
    const data = await readFilePromise('file.txt', 'utf8');
    console.log(data);
  } catch (error) {
    console.error('Error reading file:', error);
  }
}

readFileAsync();
```

39. How do you handle file uploads in Node.js?

Answer: File uploads can be handled using: - Libraries like multer or formidable - Built-in http module with streams - Express middleware - Handling multipart/form-data requests - Processing and validating uploaded files - Storing to filesystem or cloud storage

Frequency: Medium (50%)
Companies: Netflix, PayPal, Uber, Airbnb

```
// Using multer with Express
const express = require('express');
const multer = require('multer');

const app = express();

// Configure storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/');
  },
  filename: (req, file, cb) => {
    cb(null, Date.now() + '-' + file.originalname);
  }
});

// File filter
const fileFilter = (req, file, cb) => {
  // Accept only images
  if (file.mimetype.startsWith('image/')) {
    cb(null, true);
  } else {
    cb(new Error('Only images are allowed!'), false);
  }
};

const upload = multer({
  storage: storage,
  limits: { fileSize: 1024 * 1024 * 5 }, // 5MB limit
  fileFilter: fileFilter
});

// Single file upload route
app.post('/upload', upload.single('file'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded');
  }
  res.send({
    message: 'File uploaded successfully',
    file: req.file
  });
});

// Error handler
app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    return res.status(400).send(err.message);
  }
  next(err);
});
```

40. What is the purpose of the NODE_ENV environment variable?

Answer: NODE_ENV is a convention used to define the application environment. It helps: - Configure application behavior for different environments - Enable/disable features based on environment - Optimize performance in production (e.g., Express caches views in production) - Control debugging and logging levels - Common values: development, production, test

Frequency: High (65%)
Companies: All companies using Node.js

```
// In app.js
if (process.env.NODE_ENV === 'production') {
  // Enable production optimizations
  app.use(compression());
  app.use(helmet());
  // Minimal logging
  app.use(morgan('tiny'));
} else {
  // Detailed logging for development
  app.use(morgan('dev'));
  // Enable development-only features
  app.use(errorHandler());
}

console.log(`Running in ${process.env.NODE_ENV || 'development'} mode`);
```

41. Explain RESTful API design in Node.js

Answer: RESTful API design in Node.js includes: - Using HTTP methods correctly (GET, POST, PUT, DELETE) - Proper resource naming and URL structure - Status codes for different responses - Versioning APIs - Authentication and authorization - Pagination, filtering, searching - HATEOAS for discoverability - Documentation

Frequency: High (70%)

Frequency: High (70%)
Companies: Microsoft, Netflix, PayPal, Uber, Airbnb

```
// RESTful API example with Express
const express = require('express');
const router = express.Router();

// GET collection
router.get('/api/users', (req, res) => {
  // Return list of users with pagination
  res.json({ users: [...], meta: { page: 1, total: 100 } });
});

// GET single resource
router.get('/api/users/:id', (req, res) => {
  // Return specific user or 404
  const user = findUser(req.params.id);
  if (!user) return res.status(404).json({ error: 'User not found' });
  res.json(user);
});

// POST - create
router.post('/api/users', (req, res) => {
  // Validate and create new user
  const newUser = createUser(req.body);
  res.status(201).json(newUser);
});

// PUT - update
router.put('/api/users/:id', (req, res) => {
  // Update existing user or 404
  const updated = updateUser(req.params.id, req.body);
  if (!updated) return res.status(404).json({ error: 'User not found' })
  res.json(updated);
});

// DELETE
router.delete('/api/users/:id', (req, res) => {
  // Delete user or 404
  const deleted = deleteUser(req.params.id);
  if (!deleted) return res.status(404).json({ error: 'User not found' })
  res.status(204).end();
});
```

42. How would you implement WebSocket communication in Node.js?

Answer: WebSocket communication can be implemented using: - The `ws` library (lowlevel) - Socket.IO (higher-level with fallbacks) - Integration with Express or other web servers - Managing connection events and message passing - Handling errors and disconnections - Scaling with Redis adapters or other solutions

Frequency: Medium (50%)
Companies: Netflix, Uber, PayPal, Gaming companies

```
// Using Socket.IO
const express = require('express');
const { createServer } = require('http');
const { Server } = require('socket.io');

const app = express();
const httpServer = createServer(app);
const io = new Server(httpServer);

// Serve static files
app.use(express.static('public'));

// WebSocket logic
io.on('connection', (socket) => {
  console.log('Client connected', socket.id);

  // Handle incoming messages
  socket.on('chat message', (msg) => {
    console.log('Message received:', msg);
    // Broadcast to all clients
    io.emit('chat message', msg);
  });

  // Join rooms
  socket.on('join room', (room) => {
    socket.join(room);
    socket.emit('room joined', room);
  });

  // Handle disconnection
  socket.on('disconnect', () => {
    console.log('Client disconnected', socket.id);
  });
});

httpServer.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

43. What is the difference between dependencies, devDependencies, and peerDependencies?

Answer: - `dependencies` : Packages required for the application to run in production - `devDependencies` : Packages only needed during development or testing - `peerDependencies` : Packages that should be provided by the consuming project (used in plugins/libraries)

Frequency: Medium (55%)
Companies: Netflix, Microsoft, PayPal

44. How do you deploy Node.js applications to production?

Answer: Production deployment typically involves: - Using process managers (PM2, Forever) - Container technologies (Docker, Kubernetes) - CI/CD pipelines - Environment configuration management - Load balancing and scaling strategies - Monitoring and logging - Security hardening - Platform-specific deployments (Heroku, AWS, Azure)

Frequency: High (60%)
Companies: Netflix, Uber, Microsoft, Amazon

45. What is Node.js streams piping?

Answer: Piping is a mechanism to connect output of one stream to the input of another stream, creating a chain of operations. It's used for efficient data processing, avoiding loading entire files into memory.

Frequency: Medium (50%)
Companies: Netflix, PayPal, Uber

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

// Create readable, transform, and writable streams
const readStream = fs.createReadStream('input.txt');
const gzip = zlib.createGzip();
const writeStream = fs.createWriteStream('output.txt.gz');

// Pipe data: read -> compress -> write
readStream
  .pipe(gzip)
  .pipe(writeStream)
  .on('finish', () => {
    console.log('File compressed successfully');
  })
  .on('error', (err) => {
    console.error('Error:', err);
  });
```

46. How do you optimize a Node.js application for performance?

Answer: Performance optimization strategies include: - Caching (Redis, in-memory, CDN) - Load balancing with cluster module - Asynchronous operations and non-blocking code - Proper error handling - Memory leak identification and fixing - Database query optimization - HTTP compression - Optimizing static assets - Profiling and benchmarking

Frequency: High (65%)
Companies: Netflix, Uber, PayPal, Microsoft

47. What is the difference between spawn, exec, execFile, and fork methods in Node.js?

Answer: - `spawn` : Launches a command in a new process, streaming data - `exec` : Runs a command in a shell, buffering output - `execFile` : Similar to exec but doesn't use shell, more secure - `fork` : Special case of spawn for Node.js processes with IPC channel

Frequency: Medium (40%)
Companies: Netflix, Microsoft, Uber

```
const { spawn, exec, execFile, fork } = require('child_process');

// spawn example
const ls = spawn('ls', ['-la']);
ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});
ls.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});
ls.on('close', (code) => {
  console.log(`child process exited with code ${code}`);
});

// exec example
exec('ls -la', (error, stdout, stderr) => {
  if (error) {
    console.error(`exec error: ${error}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
  console.error(`stderr: ${stderr}`);
});

// execFile example
execFile('node', ['--version'], (error, stdout, stderr) => {
  if (error) {
    console.error(`execFile error: ${error}`);
    return;
  }
  console.log(`Node.js version: ${stdout}`);
});

// fork example
const child = fork('child-script.js');
child.on('message', (message) => {
  console.log('Message from child:', message);
});
child.send({ hello: 'from parent' });
```

48. What is the Event-Driven Architecture in Node.js?

Answer: Event-Driven Architecture is a programming paradigm where the flow of the program is determined by events. In Node.js, it's implemented through: - EventEmitter class for custom events - Callbacks for asynchronous operations - Event loop for handling I/O operations - Observers and listeners for event handling

Frequency: High (70%)
Companies: Netflix, PayPal, Microsoft, Uber

49. How would you implement rate limiting in a Node.js API?

Answer: Rate limiting can be implemented using: - Libraries like express-rate-limit or rate-limiter-flexible - Redis for distributed rate limiting - Token bucket or leaky bucket algorithms - Custom middleware for tracking requests - Headers for communicating limits to clients

Frequency: Medium (50%)
Companies: Netflix, PayPal, Stripe, Uber

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

const app = express();

// Configure rate limiter
const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` h
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
  message: 'Too many requests, please try again after 15 minutes'
});

// Apply rate limiter to all API endpoints
app.use('/api/', apiLimiter);

// Create a more restrictive limiter for login attempts
const loginLimiter = rateLimit({
  windowMs: 60 * 60 * 1000, // 1 hour
  max: 5, // 5 login attempts per hour
  message: 'Too many login attempts, please try again after an hour'
});

app.use('/api/login', loginLimiter);

// Routes
app.get('/api/data', (req, res) => {
  res.json({ message: 'API response' });
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

50. What is ESM and how does it differ from CommonJS in Node.js?

Answer: ESM (ECMAScript Modules) is the official standard module system for JavaScript, while CommonJS is Node.js's original module system.

Differences: - Syntax: `import/export` in ESM vs `require/module.exports` in CommonJS - ESM is static (analyzable at compile time), CommonJS is dynamic - ESM imports are asynchronous, CommonJS imports are synchronous - ESM has top-level await, CommonJS doesn't - File extensions: .mjs for ESM, .cjs for CommonJS

Frequency: Medium (55%)
Companies: Microsoft, Netflix, PayPal

```
// CommonJS (traditional Node.js)
// math.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
module.exports = { add, subtract };

// app.js
const { add } = require('./math');
console.log(add(5, 3)); // 8

// ESM (newer approach)
// math.mjs
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// app.mjs
import { add } from './math.mjs';
console.log(add(5, 3)); // 8
```

51. What is the purpose of the 'path' module in Node.js?

Answer: The path module provides utilities for working with file and directory paths in a cross-platform way. It helps with: - Normalizing paths - Joining path segments - Resolving absolute paths - Extracting directories and filenames - Handling file extensions

Frequency: High (70%)
Companies: Microsoft, Netflix, PayPal, Amazon

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

// Joining paths (cross-platform)
const fullPath = path.join(__dirname, 'subfolder', 'file.txt');
console.log(fullPath);

// Resolving to absolute path
const absolutePath = path.resolve('relative/path/file.txt');
console.log(absolutePath);

// Path components
console.log(path.dirname('/users/john/file.txt')); // /users/john
console.log(path.basename('/users/john/file.txt')); // file.txt
console.log(path.extname('/users/john/file.txt')); // .txt

// Parsing path
const pathInfo = path.parse('/users/john/file.txt');
console.log(pathInfo);
// {
//   root: '/',
//   dir: '/users/john',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file'
// }

// Normalize path (resolves '..' and '.')
console.log(path.normalize('/users/./john/../john/file.txt')); // /users
```

52. How do you implement server-side caching in Node.js?

Answer: Server-side caching can be implemented using: - In-memory caching (Map, Object) - Node-cache or memory-cache libraries - Redis or Memcached for distributed caching - File system caching - Database query caching - HTTP caching headers

Frequency: Medium (55%)
Companies: Netflix, Uber, PayPal, Microsoft

```
// Simple in-memory cache with Express
const express = require('express');
const app = express();

// Simple cache implementation
const cache = new Map();
const CACHE_TTL = 60 * 1000; // 1 minute

// Middleware for API caching
function cacheMiddleware(req, res, next) {
  const key = req.originalUrl;
  const cachedResponse = cache.get(key);

  if (cachedResponse) {
    const { data, timestamp } = cachedResponse;
    const now = Date.now();

    // Check if cache is still valid
    if (now - timestamp < CACHE_TTL) {
      console.log('Cache hit for', key);
      return res.json(data);
    } else {
      // Cache expired
      cache.delete(key);
    }
  }

  // Cache miss, modify res.json to cache the response
  const originalJson = res.json;
  res.json = function(data) {
    cache.set(key, {
      data,
      timestamp: Date.now()
    });
    return originalJson.call(this, data);
  };

  next();
}

// Apply cache middleware to specific routes
app.get('/api/products', cacheMiddleware, async (req, res) => {
  // Simulate expensive operation
  await new Promise(resolve => setTimeout(resolve, 500));
  const products = [{ id: 1, name: 'Product 1' }, { id: 2, name: 'Produc
  res.json(products);
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

53. What is the purpose of the .npmrc file?

Answer: The .npmrc (npm configuration) file is used to: - Configure npm settings per project or globally - Set custom registry URLs - Configure authentication for private registries - Set proxy configurations - Define default values for npm commands - Configure package scopes

Frequency: Low (25%)
Companies: Netflix, Microsoft, PayPal

54. How do you handle database migrations in Node.js applications?

Answer: Database migrations can be handled using: - Libraries like Knex.js, Sequelize, or TypeORM - Migration scripts with versioning - Up and down functions for migrations - CI/CD pipeline integration - Version control for migration files - Testing migrations in development/staging

Frequency: Medium (40%)
Companies: Netflix, Uber, PayPal, Airbnb

```
// Example with Knex.js
// Migration file: 20230101000000_create_users_table.js

exports.up = function(knex) {
  return knex.schema.createTable('users', table => {
    table.increments('id').primary();
    table.string('email').notNullable().unique();
    table.string('name').notNullable();
    table.string('password_hash').notNullable();
    table.boolean('is_active').defaultTo(true);
    table.timestamps(true, true);
  });
};

exports.down = function(knex) {
  return knex.schema.dropTable('users');
};

// Running migrations with CLI
// $ knex migrate:latest
// $ knex migrate:rollback
```

55. What are the differences between PUT and PATCH HTTP methods?

Answer: - PUT: Replaces the entire resource with the provided representation - PATCH: Updates only specific fields of the resource

In Node.js APIs, PUT typically requires all fields while PATCH allows partial updates.

Frequency: Medium (45%)
Companies: Microsoft, Netflix, PayPal, Uber

```
// Express handlers for PUT vs PATCH

// PUT - Full update
app.put('/api/users/:id', async (req, res) => {
  try {
    // Validate that all required fields are present
    const requiredFields = ['name', 'email', 'role', 'status'];
    for (const field of requiredFields) {
      if (!req.body[field]) {
        return res.status(400).json({ error: `${field} is required` });
      }
    }

    // Replace entire user object
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true, runValidators: true }
    );

    if (!updatedUser) {
      return res.status(404).json({ error: 'User not found' });
    }

    res.json(updatedUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

// PATCH - Partial update
app.patch('/api/users/:id', async (req, res) => {
  try {
    // Only update fields that are provided
    const updatedUser = await User.findByIdAndUpdate(
      req.params.id,
      { $set: req.body }, // Only update provided fields
      { new: true, runValidators: true }
    );
    if (!updatedUser) {
      return res.status(404).json({ error: 'User not found' });
    }

    res.json(updatedUser);
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});
```

56. What are the differences between setTimeout, setImmediate, and process.nextTick()?

Answer: - `process.nextTick()` : Executes on the current iteration of the event loop, after the current operation completes and before I/O events - `setImmediate()` : Executes in the next iteration of the event loop, after I/O events - `setTimeout(fn, 0)` : Schedules execution after a minimum delay (not guaranteed to be immediate)

Frequency: Medium (45%)
Companies: Netflix, Microsoft, PayPal

```
console.log('Start');

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

setImmediate(() => {
  console.log('setImmediate');
});

process.nextTick(() => {
  console.log('nextTick');
});

console.log('End');

// Output:
// Start
// End
// nextTick
// setTimeout or setImmediate (order can vary)
// setImmediate or setTimeout (order can vary)
```

57. How would you implement a Task Queue in Node.js?

Answer: Task queues can be implemented using: - Queue data structures in memory - Message brokers like RabbitMQ, Kafka, or Redis - Libraries like Bull, Bee-Queue, or Agenda - Worker processes with job processing logic - Retry mechanisms and error handling - Job priority and scheduling

Frequency: Medium (40%)
Companies: Netflix, Uber, PayPal, Airbnb

```
// Example using Bull queue with Redis
const Queue = require('bull');

// Create queue
const videoQueue = new Queue('video transcoding', 'redis://localhost:637

// Add jobs to queue
async function addJob(videoId) {
  await videoQueue.add(
    { videoId },
    {
      priority: 1,
      attempts: 3,
      backoff: {
        type: 'exponential',
        delay: 1000
      }
    }
  );
  console.log(`Job added for video ${videoId}`);
}

// Process jobs (in worker)
videoQueue.process(async (job) => {
  const { videoId } = job.data;
  console.log(`Processing video ${videoId}`);

  // Update progress
  job.progress(25);

  // Simulate processing
  await transcodeVideo(videoId);

  job.progress(100);
  return { status: 'complete', videoId };
});

// Handle events
videoQueue.on('completed', (job, result) => {
  console.log(`Job ${job.id} completed with result:`, result);
});

videoQueue.on('failed', (job, error) => {
  console.error(`Job ${job.id} failed with error:`, error);
});

// Example usage
addJob('video123');

// Simulated transcoding function
async function transcodeVideo(videoId) {
  return new Promise(resolve => setTimeout(() => resolve(), 5000));
}
```

58. How would you handle graceful shutdown in a Node.js application?

Answer: Graceful shutdown involves: - Listening for termination signals (SIGTERM, SIGINT) - Stopping accepting new requests - Finishing processing in-flight requests - Closing database connections - Releasing resources - Logging shutdown completion - Exiting with appropriate code

Frequency: Medium (45%)
Companies: Netflix, Uber, Microsoft, PayPal

```
const express = require('express');
const http = require('http');
const mongoose = require('mongoose');

const app = express();
const server = http.createServer(app);

// Track active connections
let connections = [];
server.on('connection', connection => {
  connections.push(connection);
  connection.on('close', () => {
    connections = connections.filter(curr => curr !== connection);
  });
});
```

```
// Start server
server.listen(3000, () => {
  console.log('Server running on port 3000');
});

// Graceful shutdown function
function gracefulShutdown(signal) {
  console.log(`${signal} received. Starting graceful shutdown`);

  // Stop accepting new requests
  server.close(() => {
    console.log('HTTP server closed');

    // Close database connection
    mongoose.connection.close(false, () => {
      console.log('Database connection closed');
      process.exit(0);
    });
  });

  // If server hasn't finished in 10s, force shutdown
  setTimeout(() => {
    console.error('Could not close connections in time, forcefully shutt
    process.exit(1);
  }, 10000);

  // Close all open connections
  connections.forEach(connection => connection.end());
  setTimeout(() => {
    connections.forEach(connection => connection.destroy());
  }, 5000);
}

// Listen for termination signals
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));
process.on('SIGINT', () => gracefulShutdown('SIGINT'));

// Handle uncaught exceptions and unhandled rejections
process.on('uncaughtException', (err) => {
  console.error('Uncaught exception:', err);
  gracefulShutdown('uncaughtException');
});

process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled rejection at:', promise, 'reason:', reason);
  gracefulShutdown('unhandledRejection');
});
```

59. What is the difference between Horizontal and Vertical scaling, and how does Node.js support each?

Answer: - Horizontal scaling: Adding more machines/instances (scaling out) - Vertical scaling: Adding more resources to existing machines (scaling up)

Node.js supports horizontal scaling through: - Cluster module - Load balancers - Stateless design - Message brokers for communication - Shared databases/caches

Frequency: Medium (35%)
Companies: Netflix, Uber, Microsoft, Amazon

60. How would you implement internationalization (i18n) in a Node.js application?

Answer: Internationalization can be implemented using: - Libraries like i18n, i18next, or node-polyglot - Locale detection from HTTP headers - Translation files in JSON/YAML - Template string interpolation - Number and date formatting - Right-to-left (RTL) support - Content negotiation

Frequency: Low (30%)
Companies: Microsoft, PayPal, Airbnb

```
// Example with i18n package
const express = require('express');
const i18n = require('i18n');
const app = express();

// Configure i18n
i18n.configure({
  locales: ['en', 'fr', 'es', 'de'],
  directory: __dirname + '/locales',
  defaultLocale: 'en',
  cookie: 'locale',
  queryParameter: 'lang'
});

// Use i18n middleware
app.use(i18n.init);

// Routes
app.get('/', (req, res) => {
  res.send(res.__('Welcome to our website'));
});

app.get('/change-locale/:locale', (req, res) => {
  res.cookie('locale', req.params.locale);
  res.redirect('back');
});

// Start server
app.listen(3000, () => {
  console.log('Server running on port 3000');
});

// Example translation files:
// locales/en.json
// {
//   "Welcome to our website": "Welcome to our website"
// }
//
// locales/fr.json
// {
//   "Welcome to our website": "Bienvenue sur notre site web"
// }
```

61. What are N-API and Node-API in Node.js?

Answer: N-API (now called Node-API) is an API for building native addons that is independent from the underlying JavaScript runtime. Benefits include: - ABI stability across Node.js versions - Backwards compatibility - Reduced need to update native modules - Better isolation and error handling - Simplifies maintaining native modules

Frequency: Low (20%)
Companies: Microsoft, Google, Netflix

62. What are the core modules in Node.js?

Answer: Core modules are built-in modules that come with Node.js installation. Key ones include: - fs (file system) - http/https (HTTP servers/clients) - path (path manipulation) - os (operating system info) - events (event handling) - util (utility functions) - stream (streaming data) - crypto (cryptography) - zlib (compression) - child_process (spawning processes)

Frequency: High (75%)
Companies: All companies using Node.js

63. How do you implement CORS in a Node.js application?

Answer: CORS (Cross-Origin Resource Sharing) can be implemented using: - The cors package in Express - Custom middleware setting CORS headers - Specific headers for preflight requests - Configuration for allowed origins, methods, headers - Credentials settings

Frequency: High (70%)
Companies: Netflix, PayPal, Microsoft, Uber

```
// Using cors package with Express
const express = require('express');
const cors = require('cors');
const app = express();

// Simple usage (allow all origins)
app.use(cors());

// Configured usage
const corsOptions = {
  origin: ['https://example.com', 'https://subdomain.example.com'],
  methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
  credentials: true,
  preflightContinue: false,
  optionsSuccessStatus: 204,
  allowedHeaders: ['Content-Type', 'Authorization']
};

app.use(cors(corsOptions));

// Or implement on specific routes
app.get('/api/public-data', cors(), (req, res) => {
  res.json({ data: 'This is public' });
});

// Different config for specific route
app.get('/api/restricted-data', cors({
  origin: 'https://trusted-site.com'
}), (req, res) => {
  res.json({ data: 'This is restricted' });
});

// Manual implementation
app.use((req, res, next) => {
  res.header('Access-Control-Allow-Origin', 'https://example.com');
  res.header('Access-Control-Allow-Headers', 'Origin, X-Requested-With, 
  res.header('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, OP

  // Handle preflight requests
  if (req.method === 'OPTIONS') {
    return res.sendStatus(204);
  }
  next();
});
```

64. How do you test Node.js applications?

Answer: Testing approaches include: - Unit testing with Jest, Mocha, or Jasmine - Integration testing with Supertest - End-to-end testing with Cypress or Playwright - Mock/stub libraries like Sinon.js - Code coverage with Istanbul/nyc - Test pyramid approach - CI/CD integration

Frequency: High (65%)
Companies: Netflix, Microsoft, PayPal, Uber

```
// Unit test example with Jest
// userService.js
const userService = {
  getUser: async (id) => {
    // DB call to get user
    return { id, name: 'John Doe' };
  }
};

module.exports = userService;

// userService.test.js
const userService = require('./userService');

jest.mock('./userService', () => ({
  getUser: jest.fn()
}));

describe('User Service', () => {
  afterEach(() => {
    jest.clearAllMocks();
  });

  test('getUser returns user data', async () => {
    const mockUser = { id: 1, name: 'John Doe' };
    userService.getUser.mockResolvedValue(mockUser);

    const result = await userService.getUser(1);

    expect(userService.getUser).toHaveBeenCalledWith(1);
    expect(result).toEqual(mockUser);
  });
});

// API integration test with Supertest
const request = require('supertest');
const app = require('../app');

describe('User API', () => {
  test('GET /api/users returns user list', async () => {
    const response = await request(app)
      .get('/api/users')
      .set('Authorization', 'Bearer token123');

    expect(response.status).toBe(200);
    expect(response.body).toHaveProperty('users');
    expect(Array.isArray(response.body.users)).toBe(true);
  });
});
```

65. What is the purpose of package-lock.json vs shrinkwrap.json?

Answer: - `package-lock.json` : Automatically generated, locks exact versions of dependencies, ignored by npm when published to registry - `npm-shrinkwrap.json` : Similar to package-lock.json but gets published with the package, controlling downstream installations

Both ensure consistent installations but are used in different scenarios.

Frequency: Low (30%)
Companies: Netflix, Microsoft, PayPal

66. What is pnpm and how does it differ from npm?

Answer: pnpm is an alternative package manager for Node.js that: - Uses a symlink-based approach for efficient disk space usage - Creates a non-flat node_modules structure - Offers faster installation times - Provides stricter dependency management - Has built-in monorepo support - Uses a central content-addressable store

Frequency: Low (25%)
Companies: Microsoft, Airbnb, some startups

67. How do you implement server-side rendering (SSR) with Node.js?

Answer: Server-side rendering can be implemented using: - Frameworks like Next.js or Nuxt.js - Template engines like EJS, Pug, or Handlebars - React's renderToString or Vue's renderToString - Routing logic on the server - Data fetching before rendering - Hydration on the client side

Frequency: Medium (45%)
Companies: Netflix, Airbnb, Facebook, PayPal

~~~
// Basic SSR with Express and React
const express = require('express');
const React = require('react');
const ReactDOMServer = require('react-dom/server');
const App = require('./App'); // React component

const app = express();

```javascript
// Basic SSR with Express and React
const express = require('express');
const React = require('react');
const ReactDOMServer = require('react-dom/server');
const App = require('./App'); // React component

const app = express();

app.get('*', (req, res) => {
  // Initial data fetching could happen here
  const initialData = { user: { name: 'John' } };

  // Render React app to string
  const appHtml = ReactDOMServer.renderToString(
    React.createElement(App, { data: initialData })
  );

  // Send complete HTML with rendered app
  res.send(`
    <!DOCTYPE html>
    <html>
      <head>
        <title>My SSR App</title>
      </head>
      <body>
        <div id="root">${appHtml}</div>
        <script>
          window.__INITIAL_DATA__ = ${JSON.stringify(initialData)}
        </script>
        <script src="/client.js"></script>
      </body>
    </html>
  `);
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
~~~

68. What is middleware in Node.js streaming?

Answer: In Node.js streaming, middleware refers to Transform streams that sit between Readable and Writable streams. They can modify, filter, or aggregate data as it passes through. Examples include compression, encryption, parsing, or data transformation.

Frequency: Low (25%)
Companies: Netflix, PayPal

```
const { Transform } = require('stream');
const fs = require('fs');

// Create a transform stream middleware
class UppercaseTransform extends Transform {
  _transform(chunk, encoding, callback) {
    // Transform each chunk to uppercase
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}

// Create streams
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');
const uppercaseTransform = new UppercaseTransform();

// Pipeline with middleware: read -> transform -> write
readStream
  .pipe(uppercaseTransform) // Middleware
  .pipe(writeStream)
  .on('finish', () => {
    console.log('Pipeline complete');
  });
```

69. How does Node.js handle URL parsing?

Answer: Node.js provides the URL module for parsing and formatting URLs. It supports both the WHATWG URL standard and the legacy Node.js URL API. It can parse components like protocol, hostname, path, query parameters, and hash fragments.

Frequency: Medium (40%)
Companies: PayPal, Microsoft, Netflix

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

// WHATWG URL API (newer)
const myURL = new URL('https://user:pass@example.com:8080/path/name?query

console.log(myURL.protocol); // https:
console.log(myURL.hostname); // example.com
console.log(myURL.pathname); // /path/name
console.log(myURL.searchParams.get('query')); // string
console.log(myURL.hash); // #hash

// Legacy Node.js URL API
const parsedUrl = url.parse('https://example.com/path?query=string', tru
console.log(parsedUrl.protocol); // https:
console.log(parsedUrl.pathname); // /path
console.log(parsedUrl.query.query); // string

// Formatting URL
const formattedUrl = url.format({
  protocol: 'https',
  hostname: 'example.com',
  pathname: '/products',
  query: { id: 123 }
});
console.log(formattedUrl); // https://example.com/products?id=123
```

70. What is the role of async_hooks in Node.js?

Answer: async_hooks is a core module that provides an API to track the lifetime of asynchronous resources in Node.js. It enables: - Tracking async operations across the event loop - Maintaining context across async operations - Debugging complex async patterns - Implementing context tracking systems - Performance monitoring and diagnostics

Frequency: Low (20%)
Companies: Netflix, Microsoft

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

// Store contexts by async ID
const contexts = new Map();

// Create hooks
const hooks = async_hooks.createHook({
  init(asyncId, type, triggerAsyncId) {
    // Copy context from trigger to new resource
    if (contexts.has(triggerAsyncId)) {
      contexts.set(asyncId, contexts.get(triggerAsyncId));
    }
    console.log(`Init: ${type} with ID ${asyncId}, trigger: ${triggerAsyn
  },
  before(asyncId) {
    console.log(`Before: ${asyncId}`);
  },
  after(asyncId) {
    console.log(`After: ${asyncId}`);
  },
  destroy(asyncId) {
    contexts.delete(asyncId);
    console.log(`Destroy: ${asyncId}`);
  }
});

// Enable hooks
hooks.enable();

// Setup a context
const eid = async_hooks.executionAsyncId();
contexts.set(eid, { userId: 'user-123' });

// Async operation
setTimeout(() => {
  // This callback runs in a new async context
  const asyncId = async_hooks.executionAsyncId();
  console.log(`Timer callback with ID: ${asyncId}`);
  console.log(`Current context:`, contexts.get(asyncId));

  fs.readFile(__filename, () => {
    // Another async context
    console.log(`ReadFile context:`, contexts.get(async_hooks.executionA
  });
}, 100);
```

71. What is the purpose of the querystring module in Node.js?

Answer: The querystring module provides utilities for parsing and formatting URL query strings. It helps with: - Converting between query strings and objects - Handling URL encoding/decoding - Working with form submissions - Customizing separator characters

Note: While still available, URLSearchParams is now preferred in newer code.

Frequency: Low (30%)
Companies: PayPal, Microsoft

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

// Parse query string
const parsed = querystring.parse('foo=bar&abc=xyz&abc=123');
console.log(parsed);
// { foo: 'bar', abc: ['xyz', '123'] }

// Stringify object to query string
const stringified = querystring.stringify({
  foo: 'bar',
  baz: ['qux', 'quux'],
  corge: ''
});
console.log(stringified);
// foo=bar&baz=qux&baz=quux&corge=

// With different separators
const custom = querystring.stringify({ foo: 'bar', baz: 'qux' }, ';', ':
console.log(custom);
// foo:bar;baz:qux

// Modern approach with URLSearchParams
const params = new URLSearchParams({ foo: 'bar', baz: 'qux' });
params.append('baz', 'quux');
console.log(params.toString());
// foo=bar&baz=qux&baz=quux
```

72. How do you handle file uploads with streams in Node.js?

Answer: File uploads with streams typically involve: - Parsing multipart form data -

Processing file chunks as they arrive - Writing directly to storage without buffering entire file - Handling backpressure - Progress tracking - Error handling

Frequency: Medium (40%)
Companies: Netflix, PayPal, Uber

```
const http = require('http');
const fs = require('fs');
const path = require('path');
const { Readable } = require('stream');
const { pipeline } = require('stream/promises');
const busboy = require('busboy');

const server = http.createServer(async (req, res) => {
  if (req.method === 'POST' && req.headers['content-type']?.includes('mu
    const bb = busboy({ headers: req.headers });

    // Handle file upload
    bb.on('file', (name, file, info) => {
      const { filename, encoding, mimeType } = info;
      console.log(`File upload started: ${filename}, type: ${mimeType}`)

      // Create write stream
      const savePath = path.join(__dirname, 'uploads', filename);
      const writeStream = fs.createWriteStream(savePath);

      // Track progress
      let fileSize = 0;
      file.on('data', (data) => {
        fileSize += data.length;
        console.log(`Progress: ${fileSize} bytes received`);
      });

      // Use pipeline for proper error handling and cleanup
      pipeline(file, writeStream)
        .then(() => {
          console.log(`File ${filename} uploaded successfully`);
        })
        .catch(err => {
          console.error(`Error uploading file ${filename}:`, err);
          // Cleanup partial file
          fs.unlink(savePath, () => {});
        });
    });

    bb.on('field', (name, val) => {
      console.log(`Field ${name}: ${val}`);
    });
    bb.on('close', () => {
      res.writeHead(200, { 'Content-Type': 'application/json' });
      res.end(JSON.stringify({ message: 'Upload complete' }));
    });

    req.pipe(bb);
  } else {
    res.writeHead(405);
    res.end('Method not allowed');
  }
});

server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

73. What are worker threads in Node.js and how do they work?

Answer: Worker threads are a way to run JavaScript in parallel using threads. They allow: - CPU-intensive tasks without blocking the main event loop - True parallelism with separate V8 instances - Shared memory using SharedArrayBuffer - Message passing for communication - Isolation of environments

Frequency: Medium (45%)
Companies: Netflix, Microsoft, Uber

```
// main.js
const { Worker } = require('worker_threads');

function runWorker(data) {
  return new Promise((resolve, reject) => {
    // Create new worker
    const worker = new Worker('./worker.js', {
      workerData: data
    });

    // Handle messages from worker
    worker.on('message', resolve);

    // Handle errors
    worker.on('error', reject);

    // Handle worker exit
    worker.on('exit', (code) => {
      if (code !== 0) {
        reject(new Error(`Worker stopped with exit code ${code}`));
      }
    });
  });
}

// Execute multiple CPU-intensive tasks in parallel
async function main() {
  try {
    const results = await Promise.all([
      runWorker({ value: 10, iterations: 1000000 }),
      runWorker({ value: 20, iterations: 1000000 }),
      runWorker({ value: 30, iterations: 1000000 }),
      runWorker({ value: 40, iterations: 1000000 })
    ]);

    console.log('Results:', results);
  } catch (err) {
    console.error(err);
  }
}

main();

// worker.js
const { parentPort, workerData } = require('worker_threads');

// CPU-intensive calculation
function fibonacci(n) {
  return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);
}

// Perform work based on data from main thread
function doWork() {
  const { value, iterations } = workerData;
  let result = 0;

  // Simulate CPU-intensive work
  for (let i = 0; i < iterations; i++) {
    result += fibonacci(value % 20); // Limit to avoid stack overflow
  }

  return { input: value, result };
}

// Send result back to main thread
```

```
parentPort.postMessage(doWork());
```

74. What is the purpose of the util.promisify function in Node.js?

Answer: `util.promisify` converts callback-style functions to return Promises. It: - Transforms callback-based APIs to Promise-based - Makes code more readable with async/await - Handles error-first callbacks properly - Works with most Node.js core APIs - Can handle custom callback formats with util.promisify.custom

Frequency: Medium (50%)
Companies: Netflix, Microsoft, PayPal

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

// Convert callback-based function to Promise-based
const readFile = util.promisify(fs.readFile);
const writeFile = util.promisify(fs.writeFile);

async function processFile() {
  try {
    // Much cleaner than nested callbacks
    const data = await readFile('input.txt', 'utf8');
    const processed = data.toUpperCase();
    await writeFile('output.txt', processed);
    console.log('File processing complete');
  } catch (err) {
    console.error('Error:', err);
  }
}

processFile();

// Custom promisification
function customCallback(value, callback) {
  setTimeout(() => {
    callback(null, value * 2, 'extra');
  }, 100);
}

// Define custom promisify behavior
customCallback[util.promisify.custom] = (value) => {
  return new Promise((resolve) => {
    customCallback(value, (err, result, extra) => {
      resolve({ result, extra });
    });
  });
};

const promisified = util.promisify(customCallback);

promisified(21).then(console.log); // { result: 42, extra: 'extra' }
```

75. What is the "same-origin policy" and how does it affect Node.js applications?

Answer: Same-Origin Policy is a browser security mechanism that restricts how documents/scripts from one origin can interact with resources from another origin. In Node.js applications: - It doesn't directly apply to server-side code - Node.js servers can make requests to any origin - Servers must implement CORS to allow browser clients - API clients made with Node.js aren't restricted by it

Frequency: Low (25%)
Companies: Netflix, PayPal, Microsoft

76. How would you implement real-time collaborative editing in Node.js?

Answer: Real-time collaborative editing can be implemented using: - WebSockets (Socket.IO) - Operational Transformation algorithms - Conflict resolution strategies - Document versioning - Presence indicators - Cursor position sharing - Differential synchronization

Frequency: Low (20%)
Companies: Slack, Google, Microsoft, Notion

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const { json } = require('body-parser');

const app = express();
app.use(json());
const server = http.createServer(app);
const io = new Server(server);

// Store document state
const documents = new Map();

// Basic OT function for text insertion
function transform(content, operation) {
  const { type, position, value, author } = operation;

  if (type === 'insert') {
    return content.slice(0, position) + value + content.slice(position);
  } else if (type === 'delete') {
    return content.slice(0, position) + content.slice(position + value);
  }
  return content;
}

// Socket.IO handling
```

```
io.on('connection', (socket) => {
  console.log('Client connected', socket.id);

  // Join document room
  socket.on('join-document', (documentId) => {
    socket.join(documentId);

    // Initialize document if needed
    if (!documents.has(documentId)) {
      documents.set(documentId, { content: '', version: 0 });
    }

    // Send current document state
    socket.emit('document', documents.get(documentId));

    // Broadcast user presence
    socket.to(documentId).emit('user-connected', socket.id);
  });

  // Handle operations
  socket.on('operation', (data) => {
    const { documentId, operation } = data;
    const document = documents.get(documentId);

    if (document) {
      // Apply operation
      const newContent = transform(document.content, operation);
      document.content = newContent;
      document.version++;

      // Broadcast to all clients in room except sender
      socket.to(documentId).emit('operation', {
        operation,
        version: document.version
      });
    }
  });

  // Handle cursor position
  socket.on('cursor', (data) => {
    const { documentId, position } = data;
    socket.to(documentId).emit('cursor', {
      user: socket.id,
      position
    });
  });

  // Handle disconnection
  socket.on('disconnect', () => {
    // Broadcast user disconnection to all rooms
    socket.rooms.forEach(room => {
      socket.to(room).emit('user-disconnected', socket.id);
    });
  });
});

// REST API for document management
app.get('/documents/:id', (req, res) => {
  const doc = documents.get(req.params.id);
  if (doc) {
    res.json(doc);
  } else {
    res.status(404).json({ error: 'Document not found' });
  }
});

server.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

77. What is the purpose of the assert module in Node.js?

Answer: The assert module provides a set of assertion functions for testing invariants. It's used for: - Validating assumptions in code - Writing tests without additional testing frameworks - Throwing errors when conditions aren't met - Checking value equality, types, and errors - Verifying program behavior

Frequency: Low (25%)
Companies: Microsoft, Google, Netflix

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

function add(a, b) {
  return a + b;
}

// Basic assertions
assert.strictEqual(add(2, 3), 5, 'Addition function not working correctl
assert.notStrictEqual(add(2, 3), 6, 'Addition should not equal 6');

// Type checking
assert.ok(typeof add(2, 3) === 'number', 'Return value should be a numbe

// Error assertions
assert.throws(
  () => {
    throw new Error('Invalid operation');
  },
  /Invalid operation/,
  'Expected function to throw specific error'
);

// Deep equality
const obj1 = { a: { b: 1 } };
const obj2 = { a: { b: 1 } };
assert.deepStrictEqual(obj1, obj2, 'Objects should be deeply equal');

// Custom error messages
assert.strictEqual(
  add(2, 2),
  4,
  'Basic math is broken!'
);

console.log('All assertions passed!');
```

78. How would you implement database sharding in a Node.js application?

Answer: Database sharding in Node.js involves: - Determining shard key (partition key) - Creating connection pools for each shard - Implementing routing logic to correct shard - Handling cross-shard queries - Managing distributed transactions - Data migration strategies - Consistent hashing algorithms

Frequency: Low (15%)

Companies: Uber, PayPal, Netflix

```
// Simplified example of database sharding with MongoDB

const { MongoClient } = require('mongodb');

class ShardedDatabase {
  constructor(shardConfig) {
    this.shards = new Map();
    this.shardCount = shardConfig.length;
    this.initialized = false;
    this.shardConfig = shardConfig;
  }

  async initialize() {
    // Connect to each shard
    for (const config of this.shardConfig) {
      const client = new MongoClient(config.uri);
      await client.connect();
      this.shards.set(config.id, {
        client,
        db: client.db(config.dbName)
      });
    }
    this.initialized = true;
    console.log(`Connected to ${this.shardCount} database shards`);
  }

  // Calculate shard based on user ID
  getUserShard(userId) {
    // Simple hash function
    const hash = this._hashString(userId);
    const shardId = hash % this.shardCount;
    return this.shardConfig[shardId].id;
  }

  _hashString(str) {
    let hash = 0;
    for (let i = 0; i < str.length; i++) {
      hash = (hash << 5) - hash + str.charCodeAt(i);
      hash |= 0; // Convert to 32bit integer
    }
    return Math.abs(hash);
  }

  async findUser(userId) {
    if (!this.initialized) await this.initialize();
    // Get appropriate shard
    const shardId = this.getUserShard(userId);
    const shard = this.shards.get(shardId);

    if (!shard) {
      throw new Error(`Shard ${shardId} not found`);
    }

    // Query specific shard
    return shard.db.collection('users').findOne({ userId });
  }

  async createUser(userData) {
    if (!this.initialized) await this.initialize();

    // Get appropriate shard based on user ID
    const shardId = this.getUserShard(userData.userId);
    const shard = this.shards.get(shardId);

    if (!shard) {
      throw new Error(`Shard ${shardId} not found`);
    }

    // Insert into specific shard
    return shard.db.collection('users').insertOne(userData);
  }

  // Handle cross-shard operations
  async findAllUsers() {
    if (!this.initialized) await this.initialize();

    // Query all shards and combine results
    const promises = [];
    for (const [_, shard] of this.shards) {
      promises.push(shard.db.collection('users').find({}).toArray());
    }

    const results = await Promise.all(promises);
    // Combine results from all shards
    return results.flat();
  }

  async close() {
    for (const [_, shard] of this.shards) {
      await shard.client.close();
    }
    this.shards.clear();
    this.initialized = false;
  }
}

// Usage
async function main() {
  const shardedDB = new ShardedDatabase([
    { id: 'shard0', uri: 'mongodb://localhost:27017', dbName: 'shard0' }
    { id: 'shard1', uri: 'mongodb://localhost:27018', dbName: 'shard1' }
    { id: 'shard2', uri: 'mongodb://localhost:27019', dbName: 'shard2' }
  ]);

  await shardedDB.initialize();

  // Create users - will be distributed across shards
  await shardedDB.createUser({ userId: 'user1', name: 'Alice' });
  await shardedDB.createUser({ userId: 'user2', name: 'Bob' });

  // Find user - directed to specific shard
  const user = await shardedDB.findUser('user1');
  console.log('Found user:', user);

  // Find all users - queries all shards
  const allUsers = await shardedDB.findAllUsers();
  console.log('All users:', allUsers);

  await shardedDB.close();
}
```

79. What are HTTP/2 and HTTP/3, and how can you use them in Node.js?

Answer: HTTP/2 and HTTP/3 are newer HTTP protocols offering improved performance over HTTP/1.1:

HTTP/2: - Multiplexed connections - Header compression - Server push - Binary protocol - Available in Node.js http2 module

HTTP/3: - Built on QUIC instead of TCP - Better connection migration - Improved loss recovery - Requires external libraries in Node.js

Frequency: Low (25%)
Companies: Netflix, Google, Facebook

```
// HTTP/2 server example
const http2 = require('http2');
const fs = require('fs');

// Create secure server with SSL certificates
const server = http2.createSecureServer({
  key: fs.readFileSync('server.key'),
  cert: fs.readFileSync('server.cert')
});

server.on('error', (err) => console.error(err));

server.on('stream', (stream, headers) => {
  // Get path from headers
  const path = headers[':path'];

  // Handle different routes
  if (path === '/') {
    // Respond to request
    stream.respond({
      'content-type': 'text/html',
      ':status': 200
    });

    // Send main HTML
    stream.end('<html><body><h1>Hello HTTP/2</h1></body></html>');

    // Push additional resources
    const pushStream = stream.pushStream({ ':path': '/style.css' }, (err
      if (err) throw err;
      pushStream.respond({
        'content-type': 'text/css',
        ':status': 200
      });
      pushStream.end('body { color: red; }');
    });

  } else if (path === '/api') {
    stream.respond({
      'content-type': 'application/json',
      ':status': 200
    });
    stream.end(JSON.stringify({ message: 'Hello from HTTP/2 API' }));
  } else {
    stream.respond({
      ':status': 404
    });
    stream.end('Not found');
  }
});
```

```
server.listen(8443, () => {
  console.log('HTTP/2 server listening on port 8443');
});

// HTTP/3 typically requires third-party libraries like quic or quiche
```

80. How do you implement data validation in Node.js applications?

Answer: Data validation can be implemented using: - Libraries like Joi, Yup, AJV, or validator.js - Schema-based validation - Express middleware for request validation - Custom validation functions - MongoDB schema validation - Input sanitization techniques - Error message formatting

Frequency: Medium (55%)
Companies: PayPal, Netflix, Microsoft, Uber

```
// Example using Joi with Express
const express = require('express');
const Joi = require('joi');

const app = express();
app.use(express.json());

// Define validation schema
const userSchema = Joi.object({
  username: Joi.string().alphanum().min(3).max(30).required(),
  email: Joi.string().email().required(),
  password: Joi.string().pattern(new RegExp('^[a-zA-Z0-9]{8,30}$')).requ
  age: Joi.number().integer().min(18).max(120),
  role: Joi.string().valid('user', 'admin', 'editor')
});

// Validation middleware
function validateUser(req, res, next) {
  const { error } = userSchema.validate(req.body);

  if (error) {
    return res.status(400).json({
      status: 'error',
      message: 'Invalid request data',
      details: error.details.map(x => x.message)
    });
  }
  next();
}

// Use middleware with route
app.post('/api/users', validateUser, (req, res) => {
  // At this point, req.body data is valid
  // Process the request
  res.status(201).json({
    status: 'success',
    message: 'User created',
    data: {
      id: '123',
      ...req.body
    }
  });
});

// Custom validation example
function validateQueryParams(req, res, next) {
  const { page, limit, sortBy } = req.query;

  const errors = [];

  if (page && (isNaN(page) || page < 1)) {
    errors.push('Page must be a positive number');
  }

  if (limit && (isNaN(limit) || limit < 1 || limit > 100)) {
    errors.push('Limit must be between 1 and 100');
  }

  if (sortBy && !['name', 'date', 'price'].includes(sortBy)) {
    errors.push('Sort field must be one of: name, date, price');
  }

  if (errors.length > 0) {
    return res.status(400).json({ errors });
  }

  // Convert to proper types
  if (page) req.query.page = parseInt(page);
  if (limit) req.query.limit = parseInt(limit);

  next();
}

app.get('/api/products', validateQueryParams, (req, res) => {
  // Process request with validated query params
  res.json({ message: 'Products list', params: req.query });
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

81. How do you implement a circuit breaker pattern in Node.js?

Answer: The circuit breaker pattern prevents cascading failures when a service is down. Implementation involves: - Monitoring calls to external services - Opening the circuit when failure threshold is reached - Returning fallback responses during open state - Half-open state after timeout to test recovery - State transitions with appropriate actions

Frequency: Low (25%)
Companies: Netflix, Microsoft, Uber

```
// Circuit breaker implementation
class CircuitBreaker {
  constructor(request, options = {}) {
    this.request = request;
    this.state = 'CLOSED';
    this.failureThreshold = options.failureThreshold || 5;
    this.resetTimeout = options.resetTimeout || 30000; // 30 seconds
    this.fallback = options.fallback || null;

    this.failures = 0;
    this.lastFailureTime = 0;
    this.status = {
      success: 0,
      failed: 0,
      rejected: 0,
      timeout: 0
    };
  }

  async exec(...args) {
    if (this.state === 'OPEN') {
      // Check if timeout has elapsed
      if (Date.now() - this.lastFailureTime > this.resetTimeout) {
        // Move to half-open state
        this.state = 'HALF-OPEN';
        console.log('Circuit breaker state: HALF-OPEN');
      } else {
        // Circuit is open - fail fast
        this.status.rejected++;
        if (this.fallback) {
          return this.fallback(...args);
        }
        throw new Error('Circuit is OPEN - request rejected');
      }
    }

    try {
      // Execute the request
      const response = await this.request(...args);

      // Success - reset if in half-open state
      if (this.state === 'HALF-OPEN') {
        this.reset();
      }

      this.status.success++;
      return response;
    } catch (err) {
      // Request failed
      this.failures++;
      this.status.failed++;
      this.lastFailureTime = Date.now();

      // Check if failure threshold exceeded
      if ((this.state === 'CLOSED' && this.failures >= this.failureThresh
          this.state === 'HALF-OPEN') {
        this.state = 'OPEN';
        console.log('Circuit breaker state: OPEN');
      }

      // Return fallback or re-throw error
      if (this.fallback) {
        return this.fallback(...args);
      }
      throw err;
```