Node.js Interview Notes - Complete Guide

Node.js Core Concepts

Architecture and Runtime

What is Node.js and Where to Use It

Interview One-liner: "Node.js is a JavaScript runtime built on Chrome's V8 engine that allows JavaScript to run on servers, perfect for I/O-intensive applications like APIs, real-time apps, and microservices."

```javascript
// Node.js allows server-side JavaScript
const http = require('http');

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js server!');
});

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

Best use cases: APIs, real-time applications, streaming services, microservices, build tools

V8 Engine and JavaScript Runtime Environment

Interview One-liner: "V8 is Google's JavaScript engine that compiles JavaScript to native machine code, providing the execution environment for Node.js with additional APIs for system operations."

```javascript
```

```
// V8 features in Node.js
console.log(process.versions.v8); // V8 version
console.log(process.arch); // Architecture
console.log(process.platform); // Operating system

// V8 optimizations
const start = Date.now();
// V8 optimizes hot functions
function hotFunction(x) {
  return x * x * x;
}

for (let i = 0; i < 1000000; i++) {
  hotFunction(i);
}
console.log(`Execution time: ${Date.now() - start}ms`);
```

Single-threaded Event-driven Architecture

Interview One-liner: "Node.js uses a single main thread with an event loop for non-blocking operations, while delegating I/O operations to a thread pool, making it efficient for concurrent requests."

javascript

```
// Single-threaded but non-blocking
console.log('Start');

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

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

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

console.log('End');

// Output: Start, End, Next tick callback, Immediate callback, Timer callback
```

Event Loop in Node.js vs Browser

Interview One-liner: "Node.js event loop has six phases (timer, pending callbacks, idle, poll, check, close callbacks) while browser event loop is simpler with just macrotasks and microtasks."

javascript

```javascript
// Node.js event loop phases
const fs = require('fs');

// Timer phase
setTimeout(() => console.log('Timer'), 0);

// Check phase
setImmediate(() => console.log('Immediate'));

// Poll phase
fs.readFile(__filename, () => {
  console.log('File read');

  // These will run in order within the same phase
  setTimeout(() => console.log('Inner timer'), 0);
  setImmediate(() => console.log('Inner immediate'));
});

// Next tick (highest priority)
process.nextTick(() => console.log('Next tick'));
```

Non-blocking I/O and Asynchronous Programming

Interview One-liner: "Node.js uses non-blocking I/O where operations don't wait for completion, allowing the event loop to handle other requests while I/O operations run in the background."

javascript

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

// Blocking (synchronous) - BAD
console.log('Before sync read');
const data = fs.readFileSync('large-file.txt', 'utf8');
console.log('After sync read');

// Non-blocking (asynchronous) - GOOD
console.log('Before async read');
fs.readFile('large-file.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('File read complete');
});
console.log('After async read - this runs immediately');
```

Core Modules and APIs

File System Operations (fs module)

Interview One-liner: "The fs module provides file system operations with both synchronous and asynchronous methods for reading, writing, and manipulating files and directories."

```javascript
```

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

// Read file asynchronously
fs.readFile('data.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Write file
fs.writeFile('output.txt', 'Hello Node.js', (err) => {
  if (err) throw err;
  console.log('File written');
});

// Check if file exists
fs.access('file.txt', fs.constants.F_OK, (err) => {
  console.log(err ? 'File does not exist' : 'File exists');
});

// Create directory
fs.mkdir('new-folder', { recursive: true }, (err) => {
  if (err) throw err;
  console.log('Directory created');
});

// Promises version (Node.js 10+)
const fsPromises = require('fs').promises;

async function fileOperations() {
  try {
    const data = await fsPromises.readFile('data.txt', 'utf8');
    await fsPromises.writeFile('copy.txt', data);
    console.log('File copied successfully');
  } catch (error) {
    console.error(error);
  }
}
```

Path Module for File Path Operations

Interview One-liner: "The path module provides utilities for working with file and directory paths in a cross-platform way, handling differences between Windows and Unix systems."

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

console.log(path.join('/users', 'john', 'documents', 'file.txt'));
// Output: /users/john/documents/file.txt

console.log(path.resolve('folder', 'file.txt'));
// Output: /current/working/directory/folder/file.txt

console.log(path.extname('file.txt')); // .txt
console.log(path.basename('/path/to/file.txt')); // file.txt
console.log(path.dirname('/path/to/file.txt')); // /path/to

// Cross-platform path separators
console.log(path.sep); // \ on Windows, / on Unix
console.log(path.delimiter); // ; on Windows, : on Unix

// Parse path into components
const parsed = path.parse('/users/john/documents/file.txt');
console.log(parsed);
// { root: '/', dir: '/users/john/documents', base: 'file.txt', ext: '.txt', name: 'file' }
```

HTTP Module for Servers and Clients

Interview One-liner: "The HTTP module allows creating HTTP servers and clients without external dependencies, providing low-level control over requests and responses."

```javascript
```

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

// Create HTTP server
const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);

  // Set CORS headers
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Content-Type', 'application/json');

  if (req.method === 'GET' && parsedUrl.pathname === '/api/users') {
    res.statusCode = 200;
    res.end(JSON.stringify({ users: ['John', 'Jane'] }));
  } else {
    res.statusCode = 404;
    res.end(JSON.stringify({ error: 'Not Found' }));
  }
});

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

// HTTP client
const clientReq = http.request({
  hostname: 'api.github.com',
  path: '/users/octocat',
  method: 'GET',
  headers: { 'User-Agent': 'Node.js' }
}, (res) => {
  let data = '';
  res.on('data', chunk => data += chunk);
  res.on('end', () => console.log(JSON.parse(data)));
});

clientReq.end();
```

URL and Query String Modules

Interview One-liner: "URL module parses and constructs URLs while querystring module handles URL query parameters encoding and decoding."

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

// Parse URL
const myUrl = 'https://api.example.com/users?name=john&age=25&active=true';
const parsed = url.parse(myUrl, true);

console.log(parsed.protocol); // https:
console.log(parsed.hostname); // api.example.com
console.log(parsed.pathname); // /users
console.log(parsed.query); // { name: 'john', age: '25', active: 'true' }

// Modern URL constructor (Node.js 10+)
const modernUrl = new URL(myUrl);
console.log(modernUrl.searchParams.get('name')); // john
modernUrl.searchParams.set('page', '1');
console.log(modernUrl.toString());

// Query string operations
const params = querystring.parse('name=john&age=25&hobbies=coding&hobbies=reading');
console.log(params); // { name: 'john', age: '25', hobbies: ['coding', 'reading'] }

const encoded = querystring.stringify({ name: 'john doe', city: 'new york' });
console.log(encoded); // name=john%20doe&city=new%20york
```

Crypto Module for Encryption and Hashing

Interview One-liner: "The crypto module provides cryptographic functionality including hashing, HMAC, encryption, decryption, and random number generation for security purposes."

```javascript
```

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

// Hash password
function hashPassword(password) {
  return crypto.createHash('sha256').update(password).digest('hex');
}

console.log(hashPassword('mypassword'));

// Generate random token
const token = crypto.randomBytes(32).toString('hex');
console.log(token);

// HMAC for message authentication
const secret = 'my-secret-key';
const message = 'important data';
const hmac = crypto.createHmac('sha256', secret).update(message).digest('hex');
console.log(hmac);

// Encryption/Decryption
function encrypt(text, password) {
  const algorithm = 'aes-256-ctr';
  const key = crypto.scryptSync(password, 'salt', 32);
  const iv = crypto.randomBytes(16);

  const cipher = crypto.createCipher(algorithm, key);
  const encrypted = Buffer.concat([iv, cipher.update(text), cipher.final()]);

  return encrypted.toString('hex');
}

function decrypt(hash, password) {
  const algorithm = 'aes-256-ctr';
  const key = crypto.scryptSync(password, 'salt', 32);

  const decipher = crypto.createDecipher(algorithm, key);
  const decrypted = Buffer.concat([decipher.update(Buffer.from(hash, 'hex')), decipher.final()]);

  return decrypted.toString();
}
```

Process and Environment

Process Object and Its Properties

Interview One-liner: "The process object is a global providing information about the current Node.js process with properties for arguments, environment, memory usage, and process control."

javascript

```javascript
// Process information
console.log(process.pid); // Process ID
console.log(process.ppid); // Parent Process ID
console.log(process.platform); // 'darwin', 'win32', 'linux'
console.log(process.arch); // 'x64', 'arm64'
console.log(process.version); // Node.js version
console.log(process.versions); // All versions (node, v8, openssl, etc.)

// Memory usage
console.log(process.memoryUsage());
// { rss: 123456, heapTotal: 67890, heapUsed: 45678, external: 12345, arrayBuffers: 0 }

// CPU usage
console.log(process.cpuUsage());

// Current working directory
console.log(process.cwd());

// Change directory
process.chdir('/tmp');

// Uptime
console.log(process.uptime()); // seconds since process started
```

Environment Variables and process.env

Interview One-liner: "process.env provides access to environment variables, allowing configuration without hardcoding values in source code."

javascript

```javascript
// Access environment variables
console.log(process.env.NODE_ENV); // 'development', 'production', etc.
console.log(process.env.PORT || 3000); // Default port if not set
console.log(process.env.DATABASE_URL);

// Set environment variable in code
process.env.API_KEY = 'secret-key';

// Common pattern for configuration
const config = {
  port: process.env.PORT || 3000,
  dbUrl: process.env.DATABASE_URL || 'mongodb://localhost:27017/myapp',
  jwtSecret: process.env.JWT_SECRET || 'fallback-secret',
  nodeEnv: process.env.NODE_ENV || 'development'
};

// Environment-specific behavior
if (process.env.NODE_ENV === 'production') {
  // Production-specific code
  console.log('Running in production mode');
} else {
  // Development-specific code
  console.log('Running in development mode');
}

// Load environment variables from .env file (with dotenv package)
// require('dotenv').config();
```

Command Line Arguments Handling

Interview One-liner: "Command line arguments are accessible via process.argv array, where index 0 is node executable, index 1 is script path, and remaining are user arguments."

```javascript
```

```
// process.argv contains command line arguments
console.log(process.argv);
// ['node', '/path/to/script.js', 'arg1', 'arg2', '--flag=value']

// Parse command line arguments
const args = process.argv.slice(2); // Remove 'node' and script path

function parseArgs() {
  const parsed = {};

  args.forEach(arg => {
    if (arg.startsWith('--')) {
      const [key, value] = arg.substring(2).split('=');
      parsed[key] = value || true;
    } else {
      parsed._ = parsed._ || [];
      parsed._.push(arg);
    }
  });

  return parsed;
}

const options = parseArgs();
console.log(options);

// Example usage: node script.js --port=8080 --debug input.txt
// Output: { port: '8080', debug: true, _: ['input.txt'] }

// Using commander.js library for advanced argument parsing
// const { Command } = require('commander');
// const program = new Command();
//
// program
//   .version('1.0.0')
//   .option('-p, --port <number>', 'port number', '3000')
//   .option('-d, --debug', 'enable debug mode')
//   .parse();
```

Exit Codes and Graceful Shutdowns

Interview One-liner: "Exit codes indicate process termination status (0 for success, non-zero for errors), and graceful shutdowns handle cleanup before process termination."

javascript

```
// Exit codes
process.exit(0); // Success
process.exit(1); // General error
process.exit(2); // Misuse of shell command

// Graceful shutdown handling
process.on('SIGINT', () => {
  console.log('Received SIGINT, shutting down gracefully...');

  // Cleanup operations
  server.close(() => {
    console.log('HTTP server closed');

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

process.on('SIGTERM', () => {
  console.log('Received SIGTERM, shutting down gracefully...');
  // Similar cleanup
});

// Handle uncaught exceptions
process.on('uncaughtException', (error) => {
  console.error('Uncaught Exception:', error);
  process.exit(1);
});

// Handle unhandled promise rejections
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection at:', promise, 'reason:', reason);
  process.exit(1);
});

// Graceful server shutdown example
const server = require('http').createServer();

function gracefulShutdown(signal) {
  console.log(`Received ${signal}, closing server...`);
```

```
  server.close((err) => {
    if (err) {
      console.error('Error during server shutdown:', err);
      process.exit(1);
    }
    console.log('Server closed successfully');
    process.exit(0);
  });
}

process.on('SIGINT', () => gracefulShutdown('SIGINT'));
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));
```

Advanced Node.js Topics

Event System

EventEmitter Class and Custom Events

Interview One-liner: "EventEmitter is the foundation of Node.js event-driven architecture, allowing objects to emit events and register listeners for asynchronous communication."

```javascript
```

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

// Create custom event emitter
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// Register event listeners
myEmitter.on('message', (data) => {
  console.log('Received message:', data);
});

myEmitter.once('start', () => {
  console.log('Started - this runs only once');
});

// Emit events
myEmitter.emit('message', 'Hello World');
myEmitter.emit('start');
myEmitter.emit('start'); // Won't trigger the 'once' listener again

// Real-world example: File processor
class FileProcessor extends EventEmitter {
  processFile(filename) {
    this.emit('start', filename);

    // Simulate async file processing
    setTimeout(() => {
      if (Math.random() > 0.5) {
        this.emit('success', filename);
      } else {
        this.emit('error', new Error('Processing failed'));
      }
    }, 1000);
  }
}

const processor = new FileProcessor();
processor.on('start', (file) => console.log(`Processing ${file}...`));
processor.on('success', (file) => console.log(`Successfully processed ${file}`));
processor.on('error', (err) => console.error('Error:', err.message));

processor.processFile('document.pdf');
```

Event-driven Programming Patterns

Interview One-liner: "Event-driven programming uses events to trigger actions, promoting loose coupling and reactive programming patterns in Node.js applications."

javascript

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

// Publisher-Subscriber pattern
class OrderService extends EventEmitter {
  createOrder(orderData) {
    const order = { id: Date.now(), ...orderData };

    // Emit event after order creation
    this.emit('order:created', order);
    return order;
  }
}

// Subscribers
const orderService = new OrderService();

// Email service subscriber
orderService.on('order:created', (order) => {
  console.log(`Sending confirmation email for order ${order.id}`);
});

// Inventory service subscriber
orderService.on('order:created', (order) => {
  console.log(`Updating inventory for order ${order.id}`);
});

// Analytics service subscriber
orderService.on('order:created', (order) => {
  console.log(`Recording analytics for order ${order.id}`);
});

// Create order - all subscribers are notified
orderService.createOrder({ product: 'Laptop', quantity: 1 });

// Error handling in event-driven systems
orderService.on('error', (error) => {
  console.error('Order service error:', error);
});

// Maximum listeners warning
orderService.setMaxListeners(20); // Default is 10
```

process.nextTick() vs setImmediate()

Interview One-liner: "process.nextTick() executes before any other asynchronous operations in the same phase, while setImmediate() executes in the check phase of the event loop."

```javascript
```

```
// Execution order demonstration
console.log('Start');

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

process.nextTick(() => console.log('Next Tick 1'));

setImmediate(() => console.log('Immediate 1'));

process.nextTick(() => console.log('Next Tick 2'));

setImmediate(() => console.log('Immediate 2'));

console.log('End');

// Output order:
// Start
// End
// Next Tick 1
// Next Tick 2
// Immediate 1
// Immediate 2
// Timer

// Recursive nextTick (can starve event loop - be careful!)
function recursiveNextTick(count) {
  if (count > 0) {
    process.nextTick(() => recursiveNextTick(count - 1));
  }
}

// Better approach with setImmediate for recursive calls
function recursiveImmediate(count) {
  if (count > 0) {
    setImmediate(() => recursiveImmediate(count - 1));
  }
}

// Practical use case: ensuring callback execution order
function asyncFunction(callback) {
  // Ensure callback is always asynchronous
  process.nextTick(callback);
}
```

```
asyncFunction(() => console.log('Callback executed'));
console.log('After function call');
```

Streams and Buffers

Four Types of Streams

Interview One-liner: "Node.js has four stream types: Readable (data source), Writable (data destination), Duplex (both), and Transform (modify data while reading/writing)."

javascript

```
const fs = require('fs');
const { Readable, Writable, Duplex, Transform } = require('stream');

// 1. Readable Stream
class NumberStream extends Readable {
  constructor(max) {
    super();
    this.current = 0;
    this.max = max;
  }

  _read() {
    if (this.current < this.max) {
      this.push(this.current.toString());
      this.current++;
    } else {
      this.push(null); // End of stream
    }
  }
}

const numberStream = new NumberStream(3);
numberStream.on('data', chunk => console.log('Read:', chunk.toString()));
numberStream.on('end', () => console.log('Reading finished'));

// 2. Writable Stream
class ConsoleStream extends Writable {
  _write(chunk, encoding, callback) {
    console.log('Writing:', chunk.toString());
    callback();
  }
}

const consoleStream = new ConsoleStream();
consoleStream.write('Hello');
consoleStream.write('World');
consoleStream.end();

// 3. Duplex Stream (both readable and writable)
class EchoStream extends Duplex {
  _read() {
    // Reading logic
  }
```

```
  _write(chunk, encoding, callback) {
    this.push(chunk); // Echo the data back
    callback();
  }
}

// 4. Transform Stream (modify data)
class UpperCaseTransform extends Transform {
  _transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}

const upperCaseStream = new UpperCaseTransform();
upperCaseStream.write('hello world');
upperCaseStream.on('data', data => console.log(data.toString())); // HELLO WORLD
```

Buffer Class for Binary Data Handling

Interview One-liner: "Buffer is a global class for handling binary data directly in memory, providing methods to work with raw bytes before Node.js had native binary support."

```
javascript
```

```
// Creating buffers
const buf1 = Buffer.alloc(10); // Creates 10-byte buffer filled with zeros
const buf2 = Buffer.from('Hello World', 'utf8');
const buf3 = Buffer.from([1, 2, 3, 4, 5]);

console.log(buf1); // <Buffer 00 00 00 00 00 00 00 00 00 00>
console.log(buf2); // <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64>
console.log(buf3); // <Buffer 01 02 03 04 05>

// Buffer operations
const buffer = Buffer.from('Node.js');
console.log(buffer.length); // 7
console.log(buffer.toString()); // Node.js
console.log(buffer.toString('hex')); // 4e6f64652e6a73
console.log(buffer.toString('base64')); // Tm9kZS5qcw==

// Buffer manipulation
buffer.write('Hello', 0); // Write at position 0
console.log(buffer.toString()); // Hello.js (partially overwritten)

// Copying buffers
const source = Buffer.from('Source');
const target = Buffer.alloc(10);
source.copy(target, 2); // Copy to target starting at position 2
console.log(target.toString()); // Source

// Comparing buffers
const buf4 = Buffer.from('ABC');
const buf5 = Buffer.from('ABC');
console.log(buf4.equals(buf5)); // true
console.log(Buffer.compare(buf4, buf5)); // 0 (equal)

// JSON representation
const jsonBuffer = Buffer.from('Hello');
console.log(JSON.stringify(jsonBuffer)); // {"type":"Buffer","data":[72,101,108,108,111]}
```

Stream Processing for Large Files

Interview One-liner: "Stream processing handles large files efficiently by processing data in chunks rather than loading entire files into memory, preventing memory exhaustion."

```
javascript
```

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

// Reading large files with streams
function processLargeFile(filename) {
  const readStream = fs.createReadStream(filename, { highWaterMark: 1024 });
  let totalSize = 0;
  let chunkCount = 0;

  readStream.on('data', (chunk) => {
    totalSize += chunk.length;
    chunkCount++;
    console.log(`Processed chunk ${chunkCount}, size: ${chunk.length}`);
  });

  readStream.on('end', () => {
    console.log(`File processed: ${totalSize} bytes in ${chunkCount} chunks`);
  });

  readStream.on('error', (error) => {
    console.error('Error reading file:', error);
  });
}

// Line-by-line processing for huge text files
function processLogFile(filename) {
  const fileStream = fs.createReadStream(filename);
  const rl = readline.createInterface({
    input: fileStream,
    crlfDelay: Infinity // Handle Windows line endings
  });

  let errorCount = 0;
  let totalLines = 0;

  rl.on('line', (line) => {
    totalLines++;

    if (line.includes('ERROR')) {
      errorCount++;
      console.log(`Error found at line ${totalLines}: ${line}`);
    }
```

```
    // Process line without loading entire file
  });

  rl.on('close', () => {
    console.log(`Processed ${totalLines} lines, found ${errorCount} errors`);
  });
}

// Writing large data with streams
function generateLargeFile(filename, records) {
  const writeStream = fs.createWriteStream(filename);

  for (let i = 0; i < records; i++) {
    const record = `Record ${i}: ${Date.now()}\n`;

    if (!writeStream.write(record)) {
      // Buffer is full, wait for drain event
      await new Promise(resolve => writeStream.once('drain', resolve));
    }
  }

  writeStream.end();
  return new Promise((resolve, reject) => {
    writeStream.on('finish', resolve);
    writeStream.on('error', reject);
  });
}
```

Pipe Operations and Stream Chaining

Interview One-liner: "Pipe operations connect readable streams to writable streams, allowing data to flow from source to destination while handling backpressure automatically."

javascript

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

// Basic pipe operation
const readStream = fs.createReadStream('input.txt');
const writeStream = fs.createWriteStream('output.txt');

readStream.pipe(writeStream);

// Chaining multiple transforms
const upperCaseTransform = new Transform({
  transform(chunk, encoding, callback) {
    callback(null, chunk.toString().toUpperCase());
  }
});

const addTimestampTransform = new Transform({
  transform(chunk, encoding, callback) {
    const timestamped = `[${new Date().toISOString()}] ${chunk}`;
    callback(null, timestamped);
  }
});

// Stream pipeline
fs.createReadStream('input.txt')
  .pipe(upperCaseTransform)
  .pipe(addTimestampTransform)
  .pipe(fs.createWriteStream('processed.txt'));

// Error handling with pipeline (preferred method)
pipeline(
  fs.createReadStream('input.txt'),
  zlib.createGzip(), // Compress
  upperCaseTransform,
  fs.createWriteStream('output.txt.gz'),
  (error) => {
    if (error) {
      console.error('Pipeline failed:', error);
    } else {
      console.log('Pipeline succeeded');
    }
  }
```

```
);

// Complex example: CSV processing pipeline
const csvTransform = new Transform({
  objectMode: true,
  transform(chunk, encoding, callback) {
    const lines = chunk.toString().split('\n');

    lines.forEach(line => {
      if (line.trim()) {
        const columns = line.split(',');
        this.push({ name: columns[0], age: columns[1], city: columns[2] });
      }
    });

    callback();
  }
});

const jsonTransform = new Transform({
  objectMode: true,
  transform(record, encoding, callback) {
    callback(null, JSON.stringify(record) + '\n');
  }
});

pipeline(
  fs.createReadStream('users.csv'),
  csvTransform,
  jsonTransform,
  fs.createWriteStream('users.json'),
  (error) => {
    console.log(error ? 'Failed' : 'CSV to JSON conversion complete');
  }
);
```

Child Processes and Clustering

fork() vs spawn() Methods

Interview One-liner: "spawn() launches any system command and streams I/O, while fork() specifically creates new Node.js processes with IPC communication channel."

javascript

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

// spawn() - for any system command with streaming I/O
const ls = spawn('ls', ['-la', '/tmp']);

ls.stdout.on('data', (data) => {
  console.log(`stdout: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.error(`stderr: ${data}`);
});

ls.on('close', (code) => {
  console.log(`Process exited with code ${code}`);
});

// fork() - specifically for Node.js scripts with IPC
// child.js
if (process.send) {
  process.send({ message: 'Hello from child' });

  process.on('message', (msg) => {
    console.log('Child received:', msg);
    process.send({ result: msg.data * 2 });
  });
}

// parent.js
const child = fork('./child.js');

child.on('message', (msg) => {
  console.log('Parent received:', msg);
});

child.send({ data: 42 });

// exec() -
```