Node.js Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Create a debounce utility with cancel capability

Solution:

```
function debounce(fn, wait) {
  let timeout;
  const debounced = function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => fn.apply(this, args), wait);
  };
  debounced.cancel = () => clearTimeout(timeout);
  return debounced;
}
```

2. Implement a rate limiter middleware using the token bucket algorithm

Solution:

Here's an implementation of a rate limiter middleware:

```
const rateLimit = (tokensPerInterval, interval) => {
  const tokens = new Map();
  return (req, res, next) => {
    const now = Date.now();
    const key = req.ip;
    const bucket = tokens.get(key) || { tokens: tokensPerInterval, last: now };
    bucket.tokens += (now - bucket.last) * (tokensPerInterval / interval);
    bucket.tokens = Math.min(bucket.tokens, tokensPerInterval);
    bucket.last = now;
    if (bucket.tokens >= 1) {
      bucket.tokens -= 1;
      tokens.set(key, bucket);
      next();
    } else {
      res.status(429).send('Rate limit exceeded');
    }
  };
};
```

3. Create a custom EventEmitter implementation with once() method

Solution:

```
class MyEventEmitter {
  constructor() {
    this.events = new Map();
  }
  on(event, callback) {
    if (!this.events.has(event)) this.events.set(event, []);
    this.events.get(event).push(callback);
  }
  once(event, callback) {
    const wrapper = (...args) => {
      callback(...args);
      this.off(event, wrapper);
    };
```

```
    this.on(event, wrapper);
  }
}
```

4. Implement a promise-based delay function with timeout capability

Solution:

```
const delayWithTimeout = (ms, timeoutMs) => {
  const delay = new Promise(resolve => setTimeout(resolve, ms));
  const timeout = new Promise((_, reject) =>
    setTimeout(() => reject('Timeout'), timeoutMs));
  return Promise.race([delay, timeout]);
};
```

Usage:

```
await delayWithTimeout(1000, 2000); // Resolves after 1s
await delayWithTimeout(3000, 2000); // Rejects after 2s
```

5. Create a worker pool implementation for CPU-intensive tasks

Solution:

```
const { Worker } = require('worker_threads');
class WorkerPool {
  constructor(size, workerScript) {
    this.workers = Array(size).fill().map(() => new Worker(workerScript));
    this.queue = [];
    this.workers.forEach((worker, i) => {
      worker.on('message', result => this.handleComplete(i, result));
    });
  }
}
```

6. Implement a custom middleware chain executor

Solution:

```
const compose = middlewares => {
  return (context, next) => {
    let index = -1;
    const dispatch = i => {
      if (i <= index) return Promise.reject('next() called multiple times');
      index = i;
      const fn = i === middlewares.length ? next : middlewares[i];
      if (!fn) return Promise.resolve();
      return Promise.resolve(fn(context, () => dispatch(i + 1)));
    };
    return dispatch(0);
  };
};
```

7. Create a circular buffer implementation with streaming capability

Solution:

```
class CircularBuffer {
  constructor(size) {
    this.buffer = new Array(size);
    this.size = size;
    this.head = 0;
    this.tail = 0;
    this.length = 0;
  }
  write(data) {
    this.buffer[this.head] = data;
    this.head = (this.head + 1) % this.size;
```

```
    this.length = Math.min(this.length + 1, this.size);
  }
}
```

8. Implement a custom promisify function for callback-based APIs

Solution:

```
const customPromisify = fn => {
  return function (...args) {
    return new Promise((resolve, reject) => {
      fn.call(this, ...args, (err, result) => {
        if (err) reject(err);
        else resolve(result);
      });
    });
  };
};
```

9. Implement a custom async iterator for batch processing

Solution:

```
class BatchProcessor {
  constructor(items, batchSize) {
    this.items = items;
    this.batchSize = batchSize;
  }
  async *[Symbol.asyncIterator]() {
    for (let i = 0; i < this.items.length; i += this.batchSize) {
      yield this.items.slice(i, i + this.batchSize);
      await new Promise(resolve => setTimeout(resolve, 0));
    }
  }
}
```

10. Create a memory-efficient stream transformer for large files

Solution:

```
const { Transform } = require('stream');
class ChunkTransformer extends Transform {
  constructor(options = {}) {
    super({ ...options, objectMode: true });
    this.buffer = '';
  }
  _transform(chunk, encoding, callback) {
    this.buffer += chunk.toString();
    const lines = this.buffer.split('\n');
    this.buffer = lines.pop();
    lines.forEach(line => this.push(JSON.parse(line)));
    callback();
  }
}
```

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) Cache in Node.js with a fixed capacity. What's the time complexity?

Solution

We can implement an LRU Cache using a Map and maintain capacity:

```
class LRUCache {
  constructor(capacity) {
    this.cache = new Map();
    this.capacity = capacity;
  }
  get(key) {
    if (!this.cache.has(key)) return -1;
    const value = this.cache.get(key);
    this.cache.delete(key);
    this.cache.set(key, value);
    return value;
  }
  put(key, value) {
    if (this.cache.has(key)) this.cache.delete(key);
    else if (this.cache.size >= this.capacity) {
      this.cache.delete(this.cache.keys().next().value);
    }
    this.cache.set(key, value);
  }
}
```

Time Complexity: O(1) for both get and put operations using Map

2. Implement a function that finds all pairs in an array that sum to a target value. What's the most efficient approach?

Solution

Using a Set for O(n) time complexity:

```
function findPairs(arr, target) {
  const seen = new Set();
  const pairs = [];
  for (const num of arr) {
    if (seen.has(target - num)) {
      pairs.push([num, target - num]);
    }
    seen.add(num);
  }
  return pairs;
}
```

Time Complexity: O(n) where n is array length
Space Complexity: O(n) for the Set

3. Create a rate limiter using the sliding window algorithm. How would you implement it?

Solution

```
class RateLimiter {
  constructor(windowMs, maxRequests) {
    this.windowMs = windowMs;
    this.maxRequests = maxRequests;
    this.requests = new Map();
  }
  isAllowed(clientId) {
    const now = Date.now();
    const windowStart = now - this.windowMs;
    const recentRequests = this.requests.get(clientId) || [];
    const validRequests = recentRequests.filter(time => time > windowStart);
    const isAllowed = validRequests.length < this.maxRequests;
    if (isAllowed) validRequests.push(now);
    this.requests.set(clientId, validRequests);
    return isAllowed;
  }
}
```

Time Complexity: O(k) where k is requests in window

4. Implement a circular buffer (ring buffer) in Node.js. What are the key considerations?

Solution

```
class CircularBuffer {
  constructor(size) {
    this.buffer = new Array(size);
    this.size = size;
    this.writePtr = 0;
    this.readPtr = 0;
    this.isFull = false;
  }
  write(data) {
    this.buffer[this.writePtr] = data;
    this.writePtr = (this.writePtr + 1) % this.size;
    this.isFull = this.writePtr === this.readPtr;
  }
  read() {
    if (this.readPtr === this.writePtr && !this.isFull) return null;
    const data = this.buffer[this.readPtr];
    this.readPtr = (this.readPtr + 1) % this.size;
    this.isFull = false;
    return data;
  }
}
```

Time Complexity: O(1) for both read and write operations

5. Implement a function to detect a cycle in a linked list using constant space.

Solution

Using Floyd's Tortoise and Hare algorithm:

```
function hasCycle(head) {
  let slow = head, fast = head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow === fast) return true;
  }
  return false;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

This uses the **two-pointer technique** where the fast pointer moves twice as fast as the slow pointer.

6. Implement a thread-safe producer-consumer queue with a maximum size limit.

Solution

```
class BoundedQueue {
  constructor(capacity) {
    this.queue = [];
    this.capacity = capacity;
    this.mutex = new Mutex();
    this.notFull = new ConditionVariable();
    this.notEmpty = new ConditionVariable();
  }
  async enqueue(item) {
    await this.mutex.acquire();
    while (this.queue.length === this.capacity) {
      await this.notFull.wait();
    }
    this.queue.push(item);
    this.notEmpty.signal();
    this.mutex.release();
  }
}
```

Note: This is a simplified version. Real implementation would need proper synchronization primitives.

7. Implement a trie (prefix tree) for autocomplete functionality. What's the space-time trade-off?

Solution

```
class TrieNode {
  constructor() {
    this.children = new Map();
    this.isEndOfWord = false;
  }
}
class Trie {
  constructor() {
    this.root = new TrieNode();
  }
  insert(word) {
    let node = this.root;
    for (const char of word) {
      if (!node.children.has(char)) {
        node.children.set(char, new TrieNode());
      }
      node = node.children.get(char);
    }
    node.isEndOfWord = true;
  }
}
```

Time Complexity: O(m) for insertion where m is word length
Space Complexity: O(ALPHABET_SIZE * m * n) where n is number of words

8. Implement a consistent hashing algorithm for distributed caching. What are the key benefits?

Solution

```
class ConsistentHash {
  constructor(nodes, replicas) {
    this.ring = new Map();
    this.points = [];
    nodes.forEach(node => {
      for (let i = 0; i < replicas; i++) {
        const hash = this.hash(`${node}-${i}`);
```

```
      this.ring.set(hash, node);
      this.points.push(hash);
    }
  });
  this.points.sort((a, b) => a - b);
 }
```

Benefits:

- Minimizes data redistribution when nodes change
- Ensures even distribution
- Supports dynamic node addition/removal

9. Design a memory-efficient bloom filter implementation. How do you handle false positives?

Solution

```
class BloomFilter {
 constructor(size, hashFunctions) {
  this.size = size;
  this.filter = new Array(size).fill(0);
  this.hashFunctions = hashFunctions;
 }
 add(item) {
  this.hashFunctions.forEach(hashFn => {
   const index = hashFn(item) % this.size;
   this.filter[index] = 1;
  });
 }
 mightContain(item) {
  return this.hashFunctions.every(hashFn =>
   this.filter[hashFn(item) % this.size] === 1);
 }
}
```

False Positive Rate: (1 - e^(-kn/m))^k where k=hash functions, n=items, m=filter size

10. Implement a priority queue with O(log n) insertion and extraction. What data structure would you use?

Solution

```
class PriorityQueue {
 constructor() {
  this.heap = [];
 }
 push(val) {
  this.heap.push(val);
  this.bubbleUp(this.heap.length - 1);
 }
 pop() {
  if (this.heap.length === 0) return null;
  const result = this.heap[0];
  const last = this.heap.pop();
  if (this.heap.length > 0) {
   this.heap[0] = last;
   this.bubbleDown(0);
  }
  return result;
 }
}
```

Time Complexity: O(log n) for push and pop operations
Implementation: Uses binary heap data structure

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service like bit.ly using Node.js

Key Components:

- **API Layer:** Express.js REST endpoints for URL creation/retrieval
- **Database:** MongoDB for storing URL mappings
- **Cache Layer:** Redis for frequently accessed URLs
- **ID Generation:** Base62 encoding for short URLs

Sample Code:

```
const shortId = require('shortid');
const urlMap = new Map();

app.post('/shorten', async (req, res) => {
  const longUrl = req.body.url;
  const urlId = shortId.generate();
  await redis.set(urlId, longUrl, 'EX', 3600);
  await URL.create({ short: urlId, original: longUrl });
  return res.json({ shortUrl: `domain/${urlId}` });
});
```

2. Design a real-time chat system that can handle millions of concurrent connections

Architecture Components:

- **WebSocket Server:** Socket.io for real-time bidirectional communication
- **Message Queue:** Redis Pub/Sub for scaling across nodes
- **Load Balancer:** Nginx with sticky sessions
- **Database:** MongoDB for message persistence

```
const io = require('socket.io')(server);
const sub = redis.createClient();

io.on('connection', socket => {
  socket.on('message', async msg => {
    await redis.publish('chat', JSON.stringify(msg));
    await Message.create({ room: msg.room, text: msg.text });
  });
});
```

3. Design a social media feed system that can handle high read/write throughput

System Components:

- **Cache:** Redis for hot posts and user feeds
- **Database:** Cassandra for posts (write-heavy)
- **Queue:** RabbitMQ for async processing
- **CDN:** CloudFront for media delivery

```
async function getFeed(userId) {
  const cachedFeed = await redis.get(`feed:${userId}`);
  if (cachedFeed) return JSON.parse(cachedFeed);
  const posts = await Post.find({ followers: userId })
    .sort('-createdAt').limit(50);
  await redis.setex(`feed:${userId}`, 3600, JSON.stringify(posts));
```

```
  return posts;
}
```

4. Design a distributed job scheduling system using Node.js

Core Components:

- **Queue:** Bull for job management
- **State Store:** Redis for job status
- **Workers:** Cluster module for parallel processing
- **Monitoring:** Bull Board for dashboard

```
const Queue = require('bull');
const jobQueue = new Queue('processing');

jobQueue.process(async job => {
  const result = await processTask(job.data);
  await redis.hset('job:results', job.id, JSON.stringify(result));
  return result;
});
```

5. Design a rate limiting system for a REST API

Implementation Approach:

- **Algorithm:** Token Bucket implementation
- **Storage:** Redis for distributed rate limiting
- **Middleware:** Express middleware for integration

```
const rateLimit = async (req, res, next) => {
  const key = `ratelimit:${req.ip}`;
  const tokens = await redis.get(key) || 100;
  if (tokens > 0) {
    await redis.decrby(key, 1);
    next();
  } else {
    res.status(429).send('Too Many Requests');
  }
};
```

6. Design a caching system for a high-traffic e-commerce platform

Caching Strategy:

- **Multi-level:** Application + Redis + CDN
- **Invalidation:** TTL + Event-based
- **Cache Aside Pattern:** For product data
- **Write-Through:** For inventory updates

```
async function getProduct(id) {
  const cached = await redis.get(`product:${id}`);
  if (cached) return JSON.parse(cached);
  const product = await Product.findById(id);
  await redis.setex(`product:${id}`, 3600, JSON.stringify(product));
  return product;
}
```

7. Design a notification system that can handle multiple channels (email, push, SMS)

Architecture:

- **Queue:** Kafka for message streaming
- **Workers:** Separate services per channel
- **Templates:** Handlebars for message formatting
- **Tracking:** MongoDB for delivery status

```
const notify = async (userId, message, channels) => {
  const user = await User.findById(userId);
```

```
  channels.forEach(channel => {
    kafka.produce('notifications', {
      channel, userId, message,
      template: 'default',
      metadata: { timestamp: Date.now() }
    });
  });
};
```

8. Design a real-time analytics pipeline for user events

Components:

- **Collection:** Kafka for event streaming
- **Processing:** Node.js streams for ETL
- **Storage:** ClickHouse for analytics
- **Visualization:** Socket.io for real-time updates

```
const eventStream = kafka.consumer('user-events');
eventStream.pipe(through2.obj((event, enc, cb) => {
  const processed = transform(event);
  clickhouse.insert('events', processed);
  io.emit('analytics-update', processed);
  cb();
}));
```

9. Design a distributed session management system

Key Features:

- **Storage:** Redis Cluster for session data
- **Security:** JWT for authentication
- **Scaling:** Horizontal scaling with sticky sessions
- **Cleanup:** TTL-based session expiry

```
const session = {
  async create(userId) {
    const sessionId = uuid();
    await redis.hset(`session:${sessionId}`, { userId, created: Date.now() });
    await redis.expire(`session:${sessionId}`, 86400);
    return jwt.sign({ sessionId }, process.env.JWT_SECRET);
  }
};
```

10. Design a service discovery system for microservices

Components:

- **Registry:** etcd for service registration
- **Health Check:** HTTP/TCP probes
- **Load Balancing:** Round-robin client-side
- **Caching:** Local cache with TTL

```
class ServiceRegistry {
  async register(service) {
    await etcd.put(`/services/${service.name}`, JSON.stringify({
      endpoints: service.endpoints,
      health: '/health',
      metadata: service.metadata
    }), { lease: await etcd.lease(30) });
  }
}
```

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a function to flatten a nested array in Node.js without using built-in flat() method.

Solution:

Here's an efficient recursive solution:

```
const flatten = (arr) => {
  return arr.reduce((flat, item) => {
    return flat.concat(Array.isArray(item) ? flatten(item) : item);
  }, []);
};
```

Usage:

```
console.log(flatten([1, [2, [3, 4], 5], 6])); // [1,2,3,4,5,6]
```

2. Implement a function to detect and break circular references in an object for JSON serialization.

Solution:

```
function decircular(obj, seen = new WeakSet()) {
  if (typeof obj === 'object' && obj !== null) {
    if (seen.has(obj)) return '[Circular]';
    seen.add(obj);
    return Object.fromEntries(
      Object.entries(obj).map(([k, v]) => [k, decircular(v, seen)])
    );
  }
  return obj;
}
```

Key points:

- Uses WeakSet to track seen objects
- Preserves object structure while handling circular refs
- Memory efficient due to WeakSet usage

3. Create a memory leak detector function that monitors heap usage in Node.js.

Solution:

```
const detectMemoryLeak = (iterations = 5) => {
  const usage = [];
  return setInterval(() => {
    const mem = process.memoryUsage().heapUsed;
    usage.push(mem);
    if (usage.length > iterations) {
      const growth = usage[usage.length-1] - usage[0];
      console.log(`Memory growth: ${growth} bytes`);
    }
  }, 1000);
};
```

Important aspects:

- Monitors heap usage over time

- Calculates memory growth pattern
- Helps identify memory leaks in long-running processes

4. Implement a rate limiter middleware for Express using the token bucket algorithm.

Solution:

```
const rateLimit = (tokensPerInterval, interval) => {
  let tokens = tokensPerInterval;
  setInterval(() => tokens = tokensPerInterval, interval);
  return (req, res, next) => {
    if (tokens > 0) {
      tokens--;
      next();
    } else {
      res.status(429).send('Rate limit exceeded');
    }
  };
};
```

Usage:

```
app.use(rateLimit(100, 60000)); // 100 requests per minute
```

5. Create a custom error handling middleware that logs errors to both console and file.

Solution:

```
const errorHandler = (err, req, res, next) => {
  const errLog = {
    timestamp: new Date().toISOString(),
    error: err.message,
    stack: err.stack,
    path: req.path
  };
  fs.appendFileSync('error.log', JSON.stringify(errLog) + '\n');
  res.status(500).json({ error: err.message });
};
```

- Captures error details and request context
- Provides structured logging
- Maintains error tracking history

6. Implement a function to safely monkey patch a module method while preserving the original functionality.

Solution:

```
function monkeyPatch(object, methodName, patchFn) {
  const original = object[methodName];
  object[methodName] = function (...args) {
    patchFn.call(this, args);
    return original.apply(this, args);
  };
  return () => object[methodName] = original;
}
```

Usage:

```
const unpatch = monkeyPatch(console, 'log',
  args => console.trace(`Log called with: ${args}`));
```

7. Write a function to implement debouncing with support for both leading and trailing edge execution.

Solution:

```
function debounce(fn, wait, leading = false) {
```

```
  let timeout;
  return function (...args) {
    const callNow = leading && !timeout;
    clearTimeout(timeout);
    timeout = setTimeout(() => {
      timeout = null;
      if (!leading) fn.apply(this, args);
    }, wait);
    if (callNow) fn.apply(this, args);
  };
}
```

Key features:

- Configurable leading/trailing execution
- Maintains proper this context
- Handles multiple arguments

8. Create a function to profile async function execution time with detailed metrics.

Solution:

```
async function profileAsync(fn, context = {}) {
  const start = process.hrtime.bigint();
  try {
    const result = await fn();
    const end = process.hrtime.bigint();
    return {
      result,
      duration: Number(end - start) / 1e6,
      ...context
    };
  } catch (error) {
    throw error;
  }
}
```

Usage:

```
const stats = await profileAsync(async () => await db.query());
```

9. Implement a custom EventEmitter with support for wildcard event listeners.

Solution:

```
class WildcardEmitter {
  constructor() {
    this.events = new Map();
  }
  on(event, listener) {
    if (!this.events.has(event)) {
      this.events.set(event, []);
    }
    this.events.get(event).push(listener);
  }
  emit(event, ...args) {
    this.events.forEach((listeners, key) => {
      if (key === '*' || key === event) {
        listeners.forEach(l => l(...args));
      }
    });
  }
}
```

10. Write a function to implement a concurrent task queue with configurable concurrency limits.

Solution:

```
class TaskQueue {
  constructor(concurrency = 2) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }
  async add(task) {
    if (this.running >= this.concurrency) {
      await new Promise(resolve => this.queue.push(resolve));
    }
    this.running++;
    try {
      return await task();
    } finally {
      this.running--;
      if (this.queue.length) this.queue.shift()();
    }
  }
}
```

Usage:

```
const queue = new TaskQueue(3);
await queue.add(async () => fetch(url));
```