Questões Teóricas

1. Explique a diferença entre callbacks, promises e async/await.

Callbacks são funções passadas como argumento para serem executadas após uma operação assíncrona, mas podem levar a código confuso e difícil de manter, conhecido como "callback hell". Promises foram introduzidas como uma alternativa mais elegante, permitindo encadear operações com .then() e lidar com erros com .catch(). Já o async/await é uma sintaxe moderna que simplifica ainda mais a leitura e a escrita de código assíncrono, permitindo tratar operações como se fossem síncronas, com melhor legibilidade e tratamento de erros usando try/catch.

2. O que é o Worker Threads no Node.js e qual sua utilidade?

Worker Threads é um módulo do Node.js que permite a criação de múltiplas threads de execução para realizar tarefas computacionalmente intensivas em paralelo. Ele é útil quando se deseja tirar proveito de múltiplos núcleos de CPU, evitando o bloqueio do event loop principal, o que melhora o desempenho em operações pesadas como processamento de imagens, cálculos matemáticos ou parsing de arquivos grandes.

3. Como o Socket.io facilita a comunicação em tempo real?

O Socket.io facilita a comunicação em tempo real ao fornecer uma abstração simples sobre WebSockets, com fallback automático para outras tecnologias quando necessário. Ele gerencia conexões bidirecionais entre cliente e servidor, permitindo a emissão e escuta de eventos em tempo real, com suporte a salas, reconexões automáticas, transmissão de mensagens e sincronização eficiente entre múltiplos clientes.

4. Qual é o papel dos clusters em aplicações Node.js?

Clusters permitem que uma aplicação Node.js aproveite todos os núcleos de CPU disponíveis criando múltiplos processos que compartilham a mesma porta. Isso aumenta

a escalabilidade da aplicação ao distribuir a carga de trabalho entre os processos filhos (workers), reduzindo gargalos e aproveitando melhor os recursos do servidor.

5. Liste três vantagens do PM2 para o gerenciamento de processos.

PM2 oferece monitoramento em tempo real dos processos, reinício automático em caso de falhas, e fácil escalonamento de instâncias com suporte nativo a clusters. Além disso, ele permite salvar o estado dos processos para que sejam restaurados após reinicialização do sistema.

6. O que é um child process e como ele é utilizado?

Um child process é um subprocesso criado a partir do processo principal do Node.js, geralmente usando o módulo child_process. Ele é utilizado para executar comandos do sistema, scripts externos ou tarefas paralelas sem bloquear o event loop, sendo útil, por exemplo, para manipular arquivos, interagir com outros programas ou rodar operações demoradas.

7. Explique o conceito de logging estruturado e sua importância.

Logging estruturado consiste em registrar logs no formato de objetos (como JSON), permitindo que as informações sejam facilmente analisadas por sistemas de monitoramento e visualização. É importante porque facilita o rastreamento de erros, análise de desempenho e correlação de eventos em sistemas distribuídos, além de permitir filtros, buscas e alertas mais eficientes.

8. Qual é a vantagem de usar o Winston para logs?

Winston é uma biblioteca robusta de logging para Node.js que permite configurar múltiplos transportes (como console, arquivos, bancos de dados) e formatos de saída (como JSON). Sua vantagem está na flexibilidade, facilidade de integração com serviços de observabilidade e no suporte a níveis de log, metadata e logs estruturados de forma consistente.

9. Por que é importante escalar aplicações Node.js em ambientes de produção?

Escalar aplicações Node.js é essencial para lidar com aumento de carga e manter a performance estável. Como o Node.js roda por padrão em um único processo, escalar horizontalmente (com clusters ou múltiplas instâncias) permite distribuir as requisições entre núcleos de CPU ou servidores distintos, aumentando a disponibilidade, resiliência e capacidade de resposta da aplicação.

10. Como a programação assíncrona melhora a performance de aplicações?

A programação assíncrona permite que o Node.js continue processando outras requisições enquanto aguarda o término de operações demoradas, como leitura de arquivos ou chamadas a banco de dados. Isso evita o bloqueio do event loop e melhora o aproveitamento dos recursos do servidor, resultando em maior escalabilidade e melhor desempenho sob cargas elevadas.

Questões Práticas

1. Implemente uma função que utilize async/await para buscar dados de uma API.

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

async function buscarDados(url) {
  try {
    const resposta = await axios.get(url);
    console.log('Dados recebidos:', resposta.data);
    return resposta.data;
  } catch (erro) {
    console.error('Erro ao buscar dados da API:', erro.message);
    return null;
```

```
  }
}
```

2. Configure um Worker Thread que execute uma tarefa independente.

worker.js

```
const { parentPort } = require('worker_threads');

// Exemplo: calcular a soma de 1 até N
function calcularSoma(limite) {
  let soma = 0;
  for (let i = 1; i <= limite; i++) {
    soma += i;
  }
  return soma;
}

// Ouvir mensagem do thread principal
parentPort.on('message', (limite) => {
  const resultado = calcularSoma(limite);
  parentPort.postMessage(resultado); // devolve o resultado ao thread principal
});
```

index.js

```
const { Worker } = require('worker_threads');
```

```
function executarWorker(limite) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./worker.js');

    worker.postMessage(limite); // Envia o valor para o worker

    worker.on('message', (resultado) => {
      console.log(`Resultado da soma: ${resultado}`);
      resolve(resultado);
    });

    worker.on('error', reject);
    worker.on('exit', (codigo) => {
      if (codigo !== 0) reject(new Error(`Worker parou com código ${codigo}`));
    });
  });
}
// Exemplo de uso
executarWorker(100000000)
  .then(() => console.log('Tarefa concluída sem travar o loop principal.'))
  .catch((err) => console.error('Erro no worker:', err));
```

3. Crie um servidor Socket.io que receba mensagens de um cliente.

index.js

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

```
const http = require('http');
const { Server } = require('socket.io');

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

// Servir um HTML simples
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Evento de conexão do cliente
io.on('connection', (socket) => {
  console.log('Um cliente se conectou:', socket.id);

  // Ouvindo mensagem enviada pelo cliente
  socket.on('mensagem', (dados) => {
    console.log(`Mensagem recebida do cliente ${socket.id}:`, dados);

    // (Opcional) responder ou broadcast
    socket.emit('resposta', 'Mensagem recebida com sucesso!');
  });

  socket.on('disconnect', () => {
    console.log('Cliente desconectado:', socket.id);
  });
});

server.listen(3000, () => {
  console.log('Servidor Socket.io rodando em http://localhost:3000');
});
```

index.html

```
<!DOCTYPE html>
```

```
<html>
  <head>
    <title>Cliente Socket.io</title>
  </head>
  <body>
    <h1>Enviar mensagem ao servidor</h1>
    <input id="mensagem" placeholder="Digite uma mensagem" />
    <button onclick="enviar()">Enviar</button>
    <p id="resposta"></p>

    <script src="/socket.io/socket.io.js"></script>
    <script>
      const socket = io();

      function enviar() {
        const msg = document.getElementById('mensagem').value;
        socket.emit('mensagem', msg);
      }

      socket.on('resposta', (texto) => {
        document.getElementById('resposta').innerText = texto;
      });
    </script>
  </body>
</html>
```

4. Implemente um cluster que utilize todos os núcleos do processador.

```
const cluster = require('cluster');
const os = require('os');
const express = require('express');

const totalCPUs = os.cpus().length;

if (cluster.isPrimary) {
```

```
    console.log(`Processo primário PID ${process.pid} está em execução`);
    console.log(`Iniciando ${totalCPUs} workers...`);

    // Cria um worker para cada núcleo de CPU
    for (let i = 0; i < totalCPUs; i++) {
      cluster.fork();
    }

    // Reinicia o worker caso ele falhe
    cluster.on('exit', (worker, code, signal) => {
      console.log(`Worker ${worker.process.pid} morreu. Criando um novo...`);
      cluster.fork();
    });
  } else {
    // Código que os workers executam
    const app = express();

    app.get('/', (req, res) => {
      res.send(`Resposta do worker ${process.pid}`);
    });

    const PORT = 3000;
    app.listen(PORT, () => {
      console.log(`Worker ${process.pid} escutando na porta ${PORT}`);
    });
}
```

5. Configure um processo filho que execute um comando do sistema operacional.

child.js

```
const { exec } = require('child_process');
```

```
// Executa o comando 'dir' como processo filho
exec('dir', (erro, stdout, stderr) => {
  if (erro) {
    console.error(`Erro ao executar o comando: ${erro.message}`);
    return;
  }

  if (stderr) {
    console.error(`Erro no processo filho: ${stderr}`);
    return;
  }

  console.log(`Resultado do comando "dir":\n${stdout}`);
});
```

6. Instale o PM2 e monitore uma aplicação Node.js.

A instalação do gerenciador de processos PM2, amplamente utilizado em ambientes Node.js para fins de monitoramento e estabilidade operacional, inicia-se com a execução do comando npm install -g pm2, o qual realiza a instalação global da ferramenta. Posteriormente, é desenvolvida uma aplicação mínima utilizando o framework Express, cujo conteúdo consiste em uma instância de servidor HTTP escutando em porta específica, com uma rota de resposta simples. A aplicação é então executada sob gerenciamento do PM2 por meio do comando pm2 start app.js, permitindo seu monitoramento contínuo e desacoplado do terminal.

Após o início do processo, é possível acessar a interface de monitoramento em tempo real por meio do comando pm2 monit, que exibe estatísticas de uso de CPU, memória, status e logs da aplicação. Comandos adicionais, como pm2 list, pm2 logs e pm2 restart, permitem a administração eficiente dos processos em execução. Para garantir persistência

e reinicialização automática dos serviços em caso de falha ou reinício do sistema, utilizam-se os comandos pm2 save e pm2 startup, assegurando maior robustez e confiabilidade à aplicação em ambiente produtivo.

app.js

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Aplicação rodando com PM2!');
});

app.listen(PORT, () => {
  console.log(`Servidor escutando na porta ${PORT}`);
});
```

7. Configure um logger com Winston para salvar logs em arquivo.

```
const { createLogger, format, transports } = require('winston');

const logger = createLogger({
  level: 'info', // nível mínimo para gravar
  format: format.combine(
    format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    format.printf(({ timestamp, level, message }) => `${timestamp}
[${level.toUpperCase()}]: ${message}`)
  ),
  transports: [
    new transports.File({ filename: 'logs/app.log' }) // salvar logs em arquivo
  ]
});
```

8. **Crie uma aplicação que registre logs de erros e informações.**

app.js

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

const { createLogger, format, transports } = require('winston');

const app = express();

const PORT = 3000;


// Configuração do logger Winston

const logger = createLogger({

  level: 'info',

  format: format.combine(

    format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),

    format.printf(({ timestamp, level, message }) => `${timestamp}
[${level.toUpperCase()}]: ${message}`)

  ),

  transports: [

    new transports.File({ filename: 'logs/error.log', level: 'error' }),  // logs de erro

    new transports.File({ filename: 'logs/combined.log' })              // logs info e acima

  ]

});


// Middleware para registrar cada requisição recebida

app.use((req, res, next) => {

  logger.info(`Requisição recebida: ${req.method} ${req.url}`);

  next();
```

```
});

// Rota principal
app.get('/', (req, res) => {
  res.send('Aplicação com logging funcionando!');
});

// Rota que gera um erro para teste
app.get('/erro', (req, res) => {
  try {
    throw new Error('Erro intencional para teste');
  } catch (err) {
    logger.error(`Erro capturado: ${err.message}`);
    res.status(500).send('Erro interno do servidor');
  }
});

app.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
  logger.info('Servidor iniciado');
});
```

9. Use o Socket.io para criar um chat em tempo real.

index.js

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

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

const PORT = 3000;

// Serve arquivo HTML simples para o chat
app.get('/', (req, res) => {
  res.sendFile(__dirname + '/index.html');
});

// Evento de conexão do cliente
io.on('connection', (socket) => {
  console.log('Usuário conectado:', socket.id);

  // Recebe mensagem do cliente e retransmite para todos
  socket.on('chat message', (msg) => {
    io.emit('chat message', msg);
  });

  socket.on('disconnect', () => {
    console.log('Usuário desconectado:', socket.id);
  });
});

server.listen(PORT, () => {
  console.log(`Servidor rodando em http://localhost:${PORT}`);
});
```

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Chat em tempo real</title>
```

```
    <style>
      body { font-family: Arial, sans-serif; }
      #messages { list-style-type: none; padding: 0; max-height: 300px; overflow-
y: auto; }
      #messages li { padding: 5px 10px; }
      #form { display: flex; }
      #input { flex-grow: 1; padding: 10px; font-size: 1rem; }
      #send { padding: 10px; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form id="form" action="">
      <input id="input" autocomplete="off" placeholder="Digite sua mensagem"
/><button id="send">Enviar</button>
    </form>

    <script src="/socket.io/socket.io.js"></script>
    <script>
      const socket = io();
      const form = document.getElementById('form');
      const input = document.getElementById('input');
      const messages = document.getElementById('messages');

      form.addEventListener('submit', (e) => {
        e.preventDefault();
        if (input.value.trim()) {
          socket.emit('chat message', input.value);
          input.value = '';
        }
      });

      socket.on('chat message', (msg) => {
        const item = document.createElement('li');
```

```
            item.textContent = msg;
            messages.appendChild(item);
            messages.scrollTop = messages.scrollHeight;
          });
        </script>
      </body>
</html>
```

10. Combine Worker Threads e logging para monitorar tarefas em segundo plano.

worker.js

```
const { parentPort } = require('worker_threads');

function tarefaPesada() {
  // Simula tarefa demorada
  let count = 0;
  for (let i = 0; i < 1e8; i++) {
    count += i;
  }
  return count;
}

parentPort.on('message', (msg) => {
  if (msg === 'start') {
    const resultado = tarefaPesada();
    parentPort.postMessage({ status: 'done', resultado });
  }
});
```

index.js

```
const { Worker } = require('worker_threads');
const { createLogger, format, transports } = require('winston');
const path = require('path');

// Configuração do logger Winston
```

```
const logger = createLogger({
  level: 'info',
  format: format.combine(
    format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    format.printf(({ timestamp, level, message }) => `${timestamp}
[${level.toUpperCase()}]: ${message}`)
  ),
  transports: [
    new transports.Console(),
    new transports.File({ filename: 'logs/task.log' })
  ]
});

function executarWorker() {
  return new Promise((resolve, reject) => {
    const worker = new Worker(path.resolve(__dirname, 'worker.js'));

    worker.on('message', (msg) => {
      if (msg.status === 'done') {
        logger.info(`Worker finalizou a tarefa com resultado: ${msg.resultado}`);
        resolve(msg.resultado);
      }
    });

    worker.on('error', (err) => {
      logger.error(`Erro no worker: ${err.message}`);
      reject(err);
    });

    worker.on('exit', (code) => {
      if (code !== 0) {
        const errorMsg = `Worker saiu com código ${code}`;
        logger.error(errorMsg);
        reject(new Error(errorMsg));
```

```
      }
    });

    logger.info('Iniciando worker para executar tarefa pesada...');
    worker.postMessage('start');
  });
}

// Executa o worker e aguarda o resultado
executarWorker()
  .then((resultado) => {
    logger.info(`Tarefa concluída com sucesso. Resultado: ${resultado}`);
  })
  .catch((err) => {
    logger.error(`Falha ao executar tarefa: ${err.message}`);
});
```