### **1. O que é Node.js e por que é popular no desenvolvimento web?**

O Node.js é um ambiente de execução de código JavaScript do lado do servidor (backend), construído sobre o motor V8 do Google Chrome. Diferente do JavaScript que roda no navegador, o Node.js permite que os desenvolvedores utilizem a mesma linguagem para criar aplicações de rede escaláveis e de alta performance.

**Sua popularidade se deve a vários fatores:**

* **Modelo de E/S (Entrada e Saída) não bloqueante e assíncrono:** Esta é a principal característica do Node.js. Ele opera em uma única thread, utilizando um "loop de eventos" para lidar com múltiplas conexões simultaneamente. Ao invés de esperar por uma operação demorada (como a leitura de um banco de dados) ser concluída, o Node.js delega essa tarefa e continua a processar outras requisições. Isso o torna extremamente eficiente para aplicações em tempo real, como chats, jogos online e streaming de dados.
* **JavaScript em todo lugar:** A possibilidade de usar JavaScript tanto no cliente (navegador) quanto no servidor simplifica o processo de desenvolvimento, permitindo que as equipes compartilhem código e conhecimento.
* **Vasto ecossistema de pacotes (npm):** O Node Package Manager (npm) é o maior registro de pacotes de software do mundo. Com milhões de bibliotecas e ferramentas disponíveis gratuitamente, os desenvolvedores podem acelerar o desenvolvimento de suas aplicações sem precisar "reinventar a roda".
* **Ideal para Microserviços e APIs:** A natureza leve e rápida do Node.js o torna uma escolha excelente para a construção de microserviços e APIs (Interfaces de Programação de Aplicações) que precisam lidar com um grande volume de requisições.

### **2. Qual é a diferença entre o Node.js e outras tecnologias de servidor, como o Apache?**

A principal diferença reside na arquitetura e no modelo de tratamento de requisições.

|  |  |  |
| --- | --- | --- |
| Característica | Node.js | Servidor Apache |
| **Arquitetura** | Orientado a eventos, single-threaded (uma única thread principal) com loop de eventos. | Baseado em processos ou threads. |
| **Tratamento de Requisições** | Não bloqueante (assíncrono). Uma única thread lida com múltiplas conexões. | Bloqueante (síncrono). Geralmente, cria uma nova thread ou processo para cada requisição. |
| **Desempenho** | Altamente performático para operações de E/S intensivas e aplicações em tempo real. | Robusto e confiável para servir conteúdo estático, mas pode consumir mais memória sob alta concorrência. |
| **Uso de Memória** | Geralmente consome menos memória, pois não cria threads para cada conexão. | O consumo de memória pode aumentar significativamente com o aumento do número de conexões simultâneas. |

Em suma, o Node.js se destaca em cenários com muitas conexões simultâneas e operações que não exigem uso intensivo da CPU, enquanto o Apache é uma solução tradicional e comprovada, especialmente eficaz para servir arquivos estáticos.

### **3. Como você inicia um projeto Node.js usando o npm?**

Iniciar um projeto Node.js é um processo simples utilizando o npm. Siga estes passos:

1. **Crie um diretório para o seu projeto:**

Bash

mkdir meu-projeto-node  
cd meu-projeto-node

1. **Inicie o projeto com o npm:**

Bash

npm init

Este comando iniciará um assistente que fará algumas perguntas sobre o seu projeto, como nome, versão, descrição, ponto de entrada (geralmente index.js), etc. Você pode pressionar Enter para aceitar os padrões.

1. **Para aceitar todas as configurações padrão automaticamente, use:**

Bash

npm init -y

Ao final do processo, um arquivo chamado package.json será criado no diretório do seu projeto. Este arquivo é fundamental, pois contém metadados sobre o projeto e gerencia suas dependências e scripts.

### **4. O que é o Express.js e qual é o seu papel no desenvolvimento web com Node.js?**

O Express.js é um framework para aplicações web Node.js, minimalista e flexível. Ele não impõe uma estrutura rígida, mas fornece um conjunto robusto de recursos para o desenvolvimento de aplicações web e APIs.

**Seu papel é simplificar tarefas complexas do desenvolvimento web, como:**

* **Roteamento:** Facilita a definição de como a aplicação responde a requisições HTTP em diferentes URLs (endpoints).
* **Middleware:** Permite a execução de funções intermediárias para tratar requisições e respostas.
* **Renderização de Views:** Integra-se com motores de template (como Pug, EJS) para gerar HTML dinamicamente.
* **Tratamento de Requisições e Respostas:** Oferece uma API simples para manipular os objetos de requisição (req) e resposta (res).

Essencialmente, o Express.js fornece a estrutura básica para construir aplicações web sobre o Node.js, permitindo que os desenvolvedores se concentrem na lógica de negócio da aplicação.

### **5. Explique o conceito de middleware no contexto do Express.js.**

No Express.js, o **middleware** é uma função que tem acesso aos objetos de requisição (req), de resposta (res) e à próxima função de middleware no ciclo de requisição-resposta da aplicação, comumente denotada por uma variável chamada next.

**As funções de middleware podem executar as seguintes tarefas:**

* Executar qualquer código.
* Fazer alterações nos objetos de requisição e resposta.
* Encerrar o ciclo de requisição-resposta.
* Chamar a próxima função de middleware na pilha.

Se uma função de middleware não encerrar o ciclo, ela **deve** chamar next() para passar o controle para a próxima função. Caso contrário, a requisição ficará "pendurada".

**Exemplo de um middleware simples que registra a data de cada requisição:**

JavaScript

const express = require('express');  
const app = express();  
  
const meuLogger = (req, res, next) => {  
 console.log('LOG:', new Date().toISOString());  
 next(); // Passa para o próximo middleware ou rota  
};  
  
app.use(meuLogger); // Aplica o middleware a todas as requisições  
  
app.get('/', (req, res) => {  
 res.send('Página Inicial');  
});  
  
app.listen(3000);

### **6. Como você roteia solicitações HTTP em um aplicativo Express?**

O roteamento em Express define como um aplicativo responde a uma solicitação de cliente para um endpoint específico, que é uma URI (ou caminho) e um método de solicitação HTTP específico (GET, POST, etc.).

A estrutura básica de uma rota é: app.METHOD(PATH, HANDLER)

Onde:

* app é uma instância do Express.
* METHOD é um método de requisição HTTP em letras minúsculas (ex: get, post, put, delete).
* PATH é o caminho no servidor.
* HANDLER é a função a ser executada quando a rota é correspondida.

**Exemplo:**

JavaScript

const express = require('express');  
const app = express();  
  
// Responde a requisições GET na raiz ('/')  
app.get('/', (req, res) => {  
 res.send('Olá, Mundo! (GET)');  
});  
  
// Responde a requisições POST em '/usuarios'  
app.post('/usuarios', (req, res) => {  
 res.send('Usuário criado com sucesso! (POST)');  
});  
  
app.listen(3000);

Para aplicações maiores, o express.Router é usado para agrupar e modularizar as definições de rotas.

### **7. O que é o middleware de análise de corpo (body-parser) e por que é útil em um aplicativo Express?**

O middleware de análise de corpo (conhecido historicamente como body-parser) é responsável por extrair a parte do corpo de um fluxo de requisição de entrada e expô-la no objeto req.body.

**Por que é útil?** Quando um cliente envia dados para o servidor através de uma requisição POST ou PUT (por exemplo, a partir de um formulário HTML ou de uma requisição de API), esses dados são enviados no corpo da requisição. Por padrão, o Node.js não sabe como interpretar esses dados.

Este middleware "analisa" (faz o *parsing* do) corpo da requisição em diferentes formatos, como JSON ou urlencoded (dados de formulário), e os torna disponíveis como um objeto JavaScript em req.body, facilitando muito a manipulação desses dados no servidor.

**Uso:** Em versões recentes do Express (4.x e superiores), o body-parser vem integrado. Você pode usá-lo da seguinte forma:

JavaScript

const express = require('express');  
const app = express();  
  
// Para analisar corpos de requisição em JSON  
app.use(express.json());  
  
// Para analisar corpos de requisição com dados de formulário  
app.use(express.urlencoded({ extended: true }));  
  
app.post('/api/usuarios', (req, res) => {  
 const novoUsuario = req.body; // Agora você pode acessar os dados aqui  
 console.log(novoUsuario);  
 res.status(201).send(novoUsuario);  
});  
  
app.listen(3000);

### **8. Quais são os principais métodos HTTP e como eles são usados em rotas Express?**

Os métodos HTTP, ou "verbos", indicam a ação desejada a ser executada para um determinado recurso. Os principais são:

* **GET:** Solicita a representação de um recurso específico. Usado para buscar dados.
  + app.get('/usuarios', (req, res) => { /\* Lógica para buscar todos os usuários \*/ });
* **POST:** Submete uma entidade a um recurso específico, frequentemente causando uma mudança no estado ou efeitos colaterais no servidor. Usado para criar novos recursos.
  + app.post('/usuarios', (req, res) => { /\* Lógica para criar um novo usuário \*/ });
* **PUT:** Substitui todas as representações atuais do recurso de destino pela carga de dados da requisição. Usado para atualizar um recurso por completo.
  + app.put('/usuarios/:id', (req, res) => { /\* Lógica para atualizar um usuário específico \*/ });
* **PATCH:** Aplica modificações parciais a um recurso. Usado para atualizar parcialmente um recurso.
  + app.patch('/usuarios/:id', (req, res) => { /\* Lógica para atualizar um campo específico de um usuário \*/ });
* **DELETE:** Remove um recurso específico.
  + app.delete('/usuarios/:id', (req, res) => { /\* Lógica para deletar um usuário específico \*/ });

Esses métodos são mapeados diretamente para funções no Express, tornando a criação de APIs RESTful intuitiva.

### **9. Como você lida com erros em um aplicativo Express?**

O Express possui um manipulador de erros padrão. No entanto, para um controle mais robusto, você pode criar um middleware de tratamento de erros customizado.

Um middleware de tratamento de erros é definido da mesma forma que outros middlewares, mas com quatro argumentos em vez de três: (err, req, res, next).

**Passos para lidar com erros:**

1. **Criação do Middleware:** Crie uma função que receba os quatro argumentos. Esta função deve ser o último middleware adicionado à sua pilha de app.use().
2. **Captura de Erros:**
   1. Para erros síncronos, basta usar throw new Error('...'); dentro de um manipulador de rota. O Express irá capturá-lo.
   2. Para erros assíncronos (dentro de Promises ou async/await), você deve passar o erro para a função next(), ou seja, next(error).

**Exemplo de um manipulador de erros customizado:**

JavaScript

// ... (outros middlewares e rotas)  
  
// Manipulador para rotas não encontradas (404)  
app.use((req, res, next) => {  
 const erro = new Error('Não encontrado');  
 erro.status = 404;  
 next(erro);  
});  
  
// Middleware de tratamento de erros  
app.use((err, req, res, next) => {  
 res.status(err.status || 500);  
 res.json({  
 erro: {  
 mensagem: err.message  
 }  
 });  
});  
  
app.listen(3000);

### **10. O que é uma API RESTful e como o Express pode ser usado para criar uma?**

Uma **API RESTful** (Representational State Transfer) é um estilo de arquitetura para a criação de serviços web. Ela utiliza os métodos HTTP padrão (GET, POST, PUT, DELETE, etc.) para realizar operações CRUD (Create, Read, Update, Delete) em recursos. A comunicação é geralmente feita através de formatos de dados leves como o JSON.

**O Express é ideal para criar APIs RESTful porque:**

* Seu sistema de roteamento mapeia diretamente para os métodos HTTP e as URIs dos recursos.
* O uso de middleware facilita a implementação de funcionalidades como autenticação, validação e logging.
* É fácil enviar respostas em JSON usando res.json().

**Exemplo de uma API RESTful simples para gerenciar "tarefas" com Express:**

JavaScript

const express = require('express');  
const app = express();  
app.use(express.json());  
  
let tarefas = [  
 { id: 1, descricao: 'Estudar Node.js', concluida: false },  
 { id: 2, descricao: 'Criar API RESTful', concluida: false }  
];  
  
// GET - Obter todas as tarefas  
app.get('/tarefas', (req, res) => {  
 res.json(tarefas);  
});  
  
// POST - Criar uma nova tarefa  
app.post('/tarefas', (req, res) => {  
 const novaTarefa = {  
 id: tarefas.length + 1,  
 descricao: req.body.descricao,  
 concluida: false  
 };  
 tarefas.push(novaTarefa);  
 res.status(201).json(novaTarefa);  
});  
  
// PUT - Atualizar uma tarefa  
app.put('/tarefas/:id', (req, res) => {  
 const tarefa = tarefas.find(t => t.id === parseInt(req.params.id));  
 if (!tarefa) return res.status(404).send('Tarefa não encontrada.');  
  
 tarefa.descricao = req.body.descricao;  
 tarefa.concluida = req.body.concluida;  
 res.json(tarefa);  
});  
  
// DELETE - Deletar uma tarefa  
app.delete('/tarefas/:id', (req, res) => {  
 const tarefaIndex = tarefas.findIndex(t => t.id === parseInt(req.params.id));  
 if (tarefaIndex === -1) return res.status(404).send('Tarefa não encontrada.');  
  
 const tarefaRemovida = tarefas.splice(tarefaIndex, 1);  
 res.json(tarefaRemovida);  
});  
  
  
app.listen(3000, () => console.log('Servidor rodando na porta 3000'));