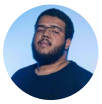
Artigos > **Programação**

Lidando com erros no Node.js



Emerson Laranja
20 de Outubro

Esse artigo faz parte da
Formação JavaScript para back-end

Introdução

Erros fazem parte da rotina de qualquer pessoa desenvolvedora, bem como saber como resolvê-los.

Quando falamos de Node.js, a maioria das informações que nos ajudam a resolver erros estão espalhadas em vários vídeos e postagem em fóruns. Por isso, podemos ter a sensação de que estamos levando mais tempo do que o necessário para resolver esses problemas. Além disso, nos deparamos com siglas que, inicialmente, parecem "misteriosas" como: `EADDRINUSE`, `ENOTFOUND`, `ECONNREFUSED`, dentre outras, e isso nos faz passar ainda mais tempo pesquisando o que são essas siglas.

Por isso, pensando em facilitar seu dia a dia, neste artigo entenderemos como ler um erro, quais são os principais erros que vemos no Node.js, as possíveis causas desses erros e como podemos resolvê-los.

Lendo um erro

Imagine o seguinte cenário: você está começando na programação, usa o `console.log` e passa como parâmetro uma variável que esqueceu de criar. Crendo que tudo está certo, você executa o seu código, como na imagem a seguir:

<> console.log(nome) Logo abaixo é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js C:\Users\Emerson\Desktop\erros

no node\index.js:1 console.log(nome) ^ ReferenceError: nome is not defined at Object. <anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:15) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47 Onde o trecho escrito “ReferenceError: nome is not defined” está sublinhado em vermelho.] (assets/lidando-com-erros-node-js/imagem2.jpg)
Um mar de palavras surge, parece ter erros até mesmo em pastas que você nem sabia que existia (`node:internal/modules` ? `Module.load` ?) e o desespero de não saber o que fazer aumenta. Mas calma! Vamos entender por partes o que está acontecendo na imagem acima.

A primeira informação que recebemos é o ponto onde nosso código foi finalizado, indicando **em qual arquivo aconteceu o erro** e o sinal de dois pontos (:), seguido do valor 1, indicando **a linha onde o erro aconteceu**:

<> console.log(nome) Logo abaixo é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js C:\Users\Emerson\Desktop\erros no node\index.js:1 console.log(nome) ^ ReferenceError: nome is not defined at Object. <anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:15) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47 Onde o trecho escrito “C:\Users\Emerson\Desktop\erros no node\index.js:1” está selecionado com uma borda vermelha. Há uma seta vermelha direcionada para o trecho, indicando qual o conteúdo principal.](assets/lidando-com-erros-node-js/imagem3.jpg)
Em seguida, você pode observar que, na linha de código que escrevemos, existe um acento circunflexo embaixo da palavra **nome** que é o que indica **o que causou o erro**.

<> console.log(nome); Logo abaixo é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js C:\Users\Emerson\Desktop\erros no node\index.js:1 console.log(nome) ^ ReferenceError: nome is not defined at Object. <anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:15) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47 Onde o trecho escrito “console.log(nome);” está selecionado com uma borda vermelha. Há uma seta vermelha direcionada para o trecho, indicando qual o conteúdo principal.] (assets/lidando-com-erros-node-js/imagem4.jpg)
Com isso, já sabemos que o possível problema está com essa palavra **nome**, mas o que de fato aconteceu podemos observar na linha seguinte:

<> console.log(nome); Logo abaixo é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js C:\Users\Emerson\Desktop\erros no node\index.js:1 console.log(nome) ^ ReferenceError: nome is not defined at Object. <anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:15) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load

(node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47
Onde o trecho escrito “ReferenceError: nome is not defined” está selecionado com uma borda vermelha. Há uma seta vermelha direcionada para o trecho, indicando qual o conteúdo principal.](assets/lidando-com-erros-node-js/imagem5.jpg)
A informação antes dos dois pontos nos diz **a classe de erro que aconteceu**, nesse caso, um erro de referência. Após os dois pontos, temos uma **mensagem de erro** mais descritiva para o usuário, explicando o que aconteceu: “nome is not defined”, em português, “nome não está definido”.

Em seguida, temos o que chamamos de **stacktrace do erro**, (rastreamento de pilha, em português) onde a primeira linha é formatada como : , e seguida por uma série de quadros de pilha (cada linha começando com "at"). Cada quadro informa o caminho passado até chegar ao erro que está sendo gerado. A palavra “at” aqui tem o significado de “em” ou “no”, então podemos ler estas linhas como “no arquivo, linha e coluna x, no arquivo, linha e coluna y, no arquivo, linha e coluna z”, etc. Como podemos observar na imagem abaixo:

<> console.log(nome); Logo abaixo é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js C:\Users\Emerson\Desktop\erros no node\index.js:1 console.log(nome) ^ ReferenceError: nome is not defined at Object. <anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:15) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47
Onde o trecho escrito “at Object.<anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:13) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47” está selecionado com uma borda vermelha. Há uma seta vermelha direcionada para o trecho, indicando qual o conteúdo principal](assets/lidando-com-erros-node-js/imagem6.jpg)
Vemos que na primeira linha da stacktrace é informado o arquivo em que ocorreu o erro. Já os sinais de dois pontos acompanhados por números nos indicam a linha em que ocorreu o erro e o caractere onde começa o erro, por exemplo, `index.js:1:15`. Perceba que se você contar os caracteres da esquerda para direita, incluindo os espaços, o décimo terceiro caractere é o **n** da palavra “nome”.

Agora está muito mais claro! Sabemos que ocorreu um erro de referência porque a variável `nome` não foi criada antes de passarmos para o `console.log` na primeira linha do nosso código. Ou seja, o Node.js não tem *referência* de onde está esse dado para utilizá-lo.

A classe Error

No geral, quando ocorre um erro no Node.js, ele será de uma das quatro categorias de erros:

- Erros padrão de JavaScript, como **SyntaxError**, **RangeError**, **ReferenceError**, **TypeError**;
- Erros do sistema, chamados pelo sistema operacional quando, por exemplo, tentamos abrir um arquivo que não existe;
- Erros personalizados pelo usuário e que serão usados no seu código;
- **AssertionErrors**, uma classe especial de erro que pode ser acionada quando o Node.js detecta uma violação lógica que não deveria ocorrer, como em um teste que falhou. Mais à frente, ainda neste artigo, vamos estudar sobre essa classe.

Porém, antes de conhecermos essas classes, é preciso entender que todos os erros de JavaScript e de sistema gerados pelo Node.js são herdados ou são instâncias da classe JavaScript `Error`. Com isso, todas essas categorias de classes possuem propriedades em comum, como:

error.name

Nos indica a classe de Erro gerada. No exemplo anterior, tivemos `RefenceError`;

error.code

Uma string que representa um identificador para tal erro;

error.message

A propriedade `message` é a descrição da string do erro conforme definido (pelo usuário, ao criar uma instância de erro ou pelo próprio Node.js). Essa mesma mensagem aparecerá na primeira linha da stacktrace do erro;

error.stack

O ponto do código onde a classe `Error` foi instanciada, seguido do caminho percorrido pelo erro;

No exemplo anterior, outra forma de visualizar seria:

```
try {
  console.log(nome)

} catch (erro) {
  console.log(`O nome do erro é: ${erro.name}\n`)
  console.log(`A mensagem de erro é: ${erro.message}\n`)
  console.log(`A stack do erro é: ${erro.stack}\n`)}
 // usamos o \n acima para pular uma linha extra e visualizarmos melhor
```

E teriamos como saída:

```
O nome do erro é: ReferenceError

A mensagem de erro é: nome is not defined

A stack do erro é: ReferenceError: nome is not defined
    at Object.<anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:2:
    at Module._compile (node:internal/modules/cjs/loader:1149:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1203:10)
    at Module.load (node:internal/modules/cjs/loader:1027:32)
    at Module._load (node:internal/modules/cjs/loader:868:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/rur
    at node:internal/main/run_main_module:23:47
```

Ou seja, além de lermos os erros, é possível "capturar" o conteúdo de um erro (suas propriedades) e com isso criarmos nossas próprias formas de lidar com eles - por exemplo, criar mensagens personalizadas, gerar um aviso para o usuário, entre várias outras opções.

Entendendo as diferentes classes de erro

Você concorda comigo que o Node.js poderia chamar toda falha inesperada de erro? Afinal, se ocorreu um erro, nada mais justo que criar a classe Error e tratá-la com os métodos que vimos no tópico anterior.

Mas, pensando na infinidade de erros que podem acontecer, o Node.js usa do conceito de herança para criar diferentes classes de erros. Assim fica muito mais claro para a pessoa desenvolvedora que tipo de erro é esse, cada um com uma mensagem personalizada indicando o porquê do erro ter acontecido.

As classes que vamos apresentar neste tópico estendem a classe Error. Ou seja, têm a classe Error como pai e usam as funcionalidades dela, porém, cada nova classe que veremos tem diferentes aplicabilidades.

> *Quer saber mais sobre o que são classes e heranças em JavaScript? Confira o nosso curso de Programação Orientada a Objetos com JavaScript!*

RangeError

Essa classe de erro é padrão de JavaScript e ocorre quando passamos um argumento fora do intervalo (ou em inglês, *range*) esperado de uma função.

No exemplo abaixo, nós estamos importando o módulo `net` e usando a função `createConnection`, responsável por criar uma conexão socket. Não se preocupe se não souber o que é uma conexão socket, nesse exemplo não entraremos nas suas funcionalidades. Mas se você quiser aprender um pouco mais sobre, te sugiro esse Alura+ que você entenderá de forma prática.

O primeiro parâmetro da função `createConnection` é a porta que estará conectada e os valores aceitos de portas vão de 0 até 65536. Então, se passamos um valor como -1, recebemos um erro informando que este valor está fora do intervalo permitido.

<> const net=require('net'); net.createConnection(-1) À esquerda da imagem é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node index.js node:internal/validators:334 throw new ERR_SOCKET_BAD_PORT(name, port, allowZero); ^ RangeError [ERR_SOCKET_BAD_PORT]: Port should be >= 0 and < 65536. Received -1. at new NodeError (node:internal/errors:393:5) at validatePort (node:internal/validators:334:11) at lookupAndConnect (node:net:1147:5) at Socket.connect (node:net:1113:5) at Object.connect (node:net:234:17) at Object.<anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:3:5) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) { code: 'ERR_SOCKET_BAD_PORT' } Onde o trecho escrito “RangeError [ERR_SOCKET_BAD_PORT]: Port should be >= 0 and < 65536. Received -1.” está sublinhado em vermelho.](assets/lidando-com-erros-node-js/imagem7.jpg)

ReferenceError

Essa classe de erro, também padrão de JavaScript, ocorre quando tentamos acessar uma variável que não está definida. Ela, geralmente, indica erros de digitação no código ou um programa quebrado (como algum bug nas dependências do seu projeto).

<> console.log(variavelNaoExiste) À esquerda é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js C:\Users\Emerson\Desktop\erros no node\index.js:1 console.log(variavelNaoExiste) ^ ReferenceError: variavelNaoExiste is not defined at Object.<anonymous> (C:\Users\Emerson\Desktop\erros no node\index.js:1:13) at Module._compile (node:internal/modules/cjs/loader:1149:14) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47 Onde o trecho escrito “ReferenceError: variavelNaoExiste is not defined.” está sublinhado em vermelho.](assets/lidando-com-erros-node-js/imagem8.jpg)

SyntaxError

Essa classe de erro, padrão de JavaScript, ocorre ao tentar interpretar código sintaticamente inválido. Ou seja, quando digitar algo que não está conforme a sintaxe da linguagem.

No exemplo abaixo, foi criada e chamada a função `gerandoSyntaxError` em que propositalmente não foi colocada a vírgula para separar as propriedades do objeto. A regra sintática do JavaScript é que as propriedades de um objeto devem ser separadas por vírgula, uma vez que a pessoa desenvolvedora não segue essa regra, recebe um erro de sintaxe.

<> function gerandoSyntaxError(){ const objeto = { nome: 'João' idade: 20 } } gerandoSyntaxError(); À direita é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js C:\Users\Emerson\Desktop\erros no node\index.js:5 idade: 20 ^^^^^ SyntaxError: Unexpected identifier at Object.compileFunction (node:vm:360:18) at wrapSafe (node:internal/modules/cjs/loader:1078:15) at Module._compile (node:internal/modules/cjs/loader:1113:27) at Module._extensions..js (node:internal/modules/cjs/loader:1203:10) at Module.load (node:internal/modules/cjs/loader:1027:32) at Module._load (node:internal/modules/cjs/loader:868:12) at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12) at node:internal/main/run_main_module:23:47 Onde o trecho escrito “SyntaxError: Unexpected identifier.” está sublinhado em vermelho.] (assets/lidando-com-erros-node-js/imagem9.jpg)

TypeError

Essa classe de erro indica que um argumento fornecido não é um tipo permitido. Por exemplo, uma função que espera receber um tipo específico como parâmetro, mas recebe outro, seria um TypeError, como mostrado abaixo - onde o método `parse` espera uma string, mas recebe um número.

<> import url from 'url'; url.parse(10); À esquerda é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js node:internal/errors:484 ErrorCaptureStackTrace(err); ^ TypeError [ERR_INVALID_ARG_TYPE]: The "url" argument must be of type string. Received type number (10) at new NodeError (node:internal/errors:393:5) at validateString (node:internal/validators:161:11) at Url.parse (node:url:185:3) at Object.urlParse [as parse] (node:url:156:13) at file:///C:/Users/Emerson/Desktop/erros%20no%20node/index.js:3:5 at ModuleJob.run (node:internal/modules/esm/module_job:193:25) at async Promise.all (index 0) at async ESMLoader.import (node:internal/modules/esm/loader:526:24) at async loadESM (node:internal/process/esm_loader:91:5) at async handleMainPromise (node:internal/modules/run_main:65:12) { code: 'ERR_INVALID_ARG_TYPE' }] (assets/lidando-com-erros-node-js/imagem10.jpg)

> *Cuidado para não confundir `TypeError` (erro de tipo) com a palavra "type" que também pode significar "digitar".*

AssertionError

O assert é, de forma resumida, um módulo JavaScript que permite testar nossas expressões.

<> import {ok} from 'assert'; function verificaParidade(numero){ if(numero % 2 === 0){ return true; }else{ return false; } } ok(verificaParidade(2), 'O número deveria ser par'); ok(verificaParidade(3), 'O número deveria ser par'); À esquerda é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js node:internal/process/esm_loader:97 internalBinding('errors').triggerUncaughtException(^ AssertionError [ERR_ASSERTION]: O número deveria ser par at file:///C:/Users/Emerson/Desktop/erros%20no%20node/index.js:13:1 at ModuleJob.run (node:internal/modules/esm/module_job:193:25) at async Promise.all (index 0) at async ESMLoader.import (node:internal/modules/esm/loader:526:24) at async loadESM (node:internal/process/esm_loader:91:5) at async handleMainPromise (node:internal/modules/run_main:65:12) { generatedMessage: false, code: 'ERR_ASSERTION', actual: false, expected: true, operator: '==' } Onde os trechos "AssertionError [ERR_ASSERTION]: O número deveria ser par.","index.js/13.1", "actual:false, expected:true" estão selecionados com uma borda vermelha.](assets/lidando-com-erros-node-js/imagem11.jpg)
No exemplo acima, eu importei um método chamado `ok`, que verifica se dada informação possui o valor `true`. Caso seja true, nada acontece.

Para testar essa funcionalidade, foi criada uma função chamada `verificaParidade` que retorna true para valores pares e false para valores ímpares.

Quando passamos como parâmetro do método `ok` a função `verificaParidade` de um número par, nada acontece (conforme o esperado). Porém, na linha 13 da imagem acima, ao passarmos para a função um valor que retornará *false* (o número 3), o AssertionError entra em ação.

O programa é finalizado e um `AssertionError` é lançado com a mensagem que escrevemos ("O número deveria ser par"), que podemos ver à direita da imagem. Além disso, podemos

observar uma indicação do valor atual (`actual:false`) e o valor esperado (`expected:true`).

Em resumo, o AssertionError será exibido sempre que nossa verificação do assert falhar. Esta é uma das bases de funcionamento dos chamados testes unitários.

SystemError

O SystemError (em português, erro de sistema) acontece quando cometemos alguma violação do sistema operacional enquanto executamos nosso código, como ler um arquivo que não existe.

<> import fs from 'fs'; fs.readFile('./arquivo_nao_existe','utf-8',(err,data)⇒{ if(err){ console.error(err); return; } console.log(data); }); À esquerda é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js [Error: ENOENT: no such file or directory, open 'C:\Users\Emerson\Desktop\erros no node\arquivo_nao_existe'] { errno: -4058, code: 'ENOENT', syscall: 'open', path: 'C:\\Users\\Emerson\\Desktop\\erros no node\\arquivo_nao_existe' } PS C:\Users\Emerson\Desktop\erros no node> Onde os trechos “[Error: ENOENT: no such file or directory” e “ros no node\arquivo_nao_existe'] { errno: -4058, code: 'ENOENT', syscall: 'open', path: 'C:\\Users\\Emerson\\Desktop\\erros no node\\arquivo_nao_existe'}” estão selecionados com uma borda em vermelho.] (assets/lidando-com-erros-node-js/imagem12.jpg)
Olha só que interessante, na imagem acima, o SystemError `retornou um objeto com algumas propriedades como` errno , code , syscall `e` path`. Esse retorno visa ajudar a pessoa desenvolvedora a entender as características do erro que está sendo recebido. Vamos entender o que cada um significa?

- `errno` : representa o número do erro fornecido pelo sistema, um identificador do erro;
- `code` : é uma string que representa o código de erro, no exemplo acima, ENOENT, indicando que não foi encontrado arquivo com o nome fornecido;
- `syscall` : é uma string que descreve a chamada de sistema que falhou, no exemplo acima foi a chamada `open` , que foi literalmente o que o programa tentou fazer: abrir (*open*) um arquivo;
- `path` : o caminho do arquivo que fornecemos.

Além dessas propriedades, também encontramos, para a classe `SystemError` , as seguintes propriedades:

- `address` : o endereço para o qual uma conexão de rede falhou;
- `dest` : o destino do caminho do arquivo ao relatar um erro do sistema de arquivos;
- `info` : detalhes extras sobre a condição de erro;
- `message` : uma descrição legível (uma frase, não um código) do erro fornecida pelo sistema;

- `port` : a porta de conexão de rede que não está disponível.

Uma atenção especial para o SystemError

Outro detalhe do exemplo anterior é que o erro não mostra o nome da classe, `SystemError` , no retorno do erro como visto nos outros exemplos de `TypeError` , `AssertionError` , dentre outros. Ao invés disso, vemos a propriedade *code* acompanhada de uma mensagem e a chamada ao sistema feita:

<> PS C:\Users\Emerson\Desktop\erros no node> node .\index.js [Error: ENOENT: no such file or directory, open 'C:\Users\Emerson\Desktop\erros no node\arquivo_nao_existe'] { errno: -4058, code: 'ENOENT', syscall: 'open', path: 'C:\\Users\\Emerson\\Desktop\\erros no node\\arquivo_nao_existe' } Os o trecho “[Error: ENOENT: no such file or directory, open 'C:\Users\Emerson\Desktop\erros no node\arquivo_nao_existe']” está selecionado com uma borda vermelha](assets/lidando-com-erros-node-js/imagem13.jpg)
Já vimos que `ENOENT` é uma string que representa um código de erro que o sistema operacional nos dá. Mas o que isso significa? Será que existem outras strings que representam erros?

No começo do nosso artigo falamos sobre a perda de tempo que podemos ter buscando o significado de algumas siglas “misteriosas” que são justamente indicadores de erros. Agora vamos conhecer algumas delas e seus significados!

Mas antes de começar, é importante pontuar que aqui vamos dar uma atenção especial para o SystemError, pois são erros extremamente comuns enquanto desenvolvemos uma aplicação Node.js, ok? Vamos nessa!

> *Todos os `E` no início das siglas significam "Error" (erro).*

ENOENT

`ENOENT` ("No Entity", uma tradução em português seria "sem entidade") acontece quando não existe a entidade esperada (arquivo ou diretório) no caminho que especificamos.

É mais comum encontrarmos em uma operação com o módulo `fs` ou executar um script que espera uma estrutura de diretório específica.

Para solucionar este erro, certifique que você informou o caminho correto para o arquivo ou diretório necessário no código, verificando erros de digitação ou alterando seu código para um caminho com o arquivo existente. Algumas funções e métodos, como, por exemplo, os usados pelo módulo `fs` do Node.js, podem exigir o uso de caminhos absolutos ou relativos.

EISDIR

`EISDIR` , ("Is a Directory", em português "é um diretório") ocorre quando uma operação espera um arquivo, mas recebe, no caminho informado, um diretório. Podemos observar isso na imagem abaixo:

<> Ao centro da imagem está o código, onde temos: import fs from 'fs'; fs.readFile('./pasta_vazia','utf-8',(err,data)⇒{ if(err){ console.error(err); return; } console.log(data); }); Já à direita, é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js [Error: EISDIR: illegal operation on a directory, read] { errno: -4068, code: 'EISDIR', syscall: 'read' } PS C:\Users\Emerson\Desktop\erros no node> Onde o trecho "[Error: EISDIR: illegal operation on a directory, read]" está selecionado com uma borda em vermelho.](assets/lidando-com-erros-node-js/imagem15.jpg)
No exemplo acima o Node.js nos fala que estamos fazendo uma operação ilegal, isso porque ele tentou fazer uma leitura (read) mas não tinha nenhum arquivo, apenas uma pasta vazia. Logo, para solucionar o erro, devemos informar a nossa função `readFile` um caminho para um arquivo!

ENOTDIR

Esse erro é o inverso do `EISDIR` . Isso significa que foi fornecido o caminho de um arquivo, esse arquivo existe (caso contrário, recebemos um erro `ENOENT`), mas o esperado era um

diretório.

<> Ao centro da imagem temos o código:import fs from 'fs'; fs.opendir('./pasta_com_arquivo/texto.txt','utf-8',(err,data)⇒{ if(err){ console.error(err); return; } console.log(data); }); Já à direita, é apresentado o terminal escrito: PS C:\Users\Emerson\Desktop\erros no node> node .\index.js [Error: ENOTDIR: not a directory, opendir 'C:\Users\Emerson\Desktop\erros no node\pasta_com_arquivo\texto.txt'] { errno: -4052, code: 'ENOTDIR', syscall: 'opendir', path: 'C:\\Users\\Emerson\\Desktop\\erros no node\\pasta_com_arquivo\\texto.txt' } PS C:\Users\Emerson\Desktop\erros no node> Onde o trecho “[Error: ENOTDIR: not a directory, opendir 'C:\Users\Emerson\Desktop\erros no node\pasta_com_arquivo\texto.txt'] {” está selecionado com uma borda em vermelho.] (assets/lidando-com-erros-node-js/imagem16.jpg)
No exemplo acima o Node.js nos fala que estamos tentando abrir um diretório (opendir), mas informamos para a função o caminho de um arquivo (texto.txt).

Para evitar esse erro, verifique em seu código se o caminho informado leva a um diretório e não a um arquivo.

ENOTFOUND

Esse erro acontece quando não conseguimos estabelecer uma conexão com algum host devido a um erro de Domain Name System – Sistema de nome de domínio. Isso geralmente significa um valor incorreto de host, ou localhost não está mapeado corretamente para 127.0.0.1 e até mesmo se o domínio está inativo ou não existe mais.

Se você receber esse erro, verifique se não cometeu um erro de digitação ao digitar o nome de domínio e se o domínio existe.

ETIMEDOUT

Quando fazemos uma solicitação de conexão HTTP, esperamos um bom tempo e não obtemos uma resposta, recebemos esse erro.

A solução geral para esse problema é capturar o erro e repetir a solicitação até que a solicitação seja bem-sucedida ou o número máximo de tentativas seja atingido. Se você encontrar esse erro com frequência, verifique as configurações de tempo limite da solicitação e, se possível, escolha um valor mais apropriado.

ECONNREFUSED

A partir do nome podemos imaginar que se trata de um erro (já que começa com E), mas e o restante do nome te dá alguma dica sobre qual o tipo de erro?

Conn me lembra bastante de **connection** (em português, conexão), concorda? E refused é, em português, recusado. A partir disso, do que você imagina se tratar esse erro?

Se você chegou a conclusão de que se trata de um erro de conexão, pensou certo!

Esse erro ocorre quando tentamos nos conectar a uma máquina de destino e ela rejeita nossa solicitação. Isso geralmente acontece quando tentamos nos conectar a um endereço que não estava acessível ou um serviço que está inativo. As causas para isso podem ser várias, mas você pode começar verificando no seu código se está se conectando a um serviço existente e online, e se sua aplicação tem as permissões necessárias para fazer essa conexão.

EADDRINUSE

Você provavelmente já entrou em um estabelecimento e quando precisou ir ao banheiro, tentou abrir a porta e não pode entrar, pois já tinha outro cliente usando. Essa situação é um bom exemplo do que esse erro significa.

Recebemos esse erro ao iniciar ou reiniciar um servidor web, onde tentamos acessar uma porta que já está ocupada por algum outro servidor.

Uma forma rápida de verificarmos esse erro é criando um servidor que está usando em uma porta, como no exemplo abaixo:

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((_, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Olá mundo!');
});

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

Esse código está dentro de um arquivo `index.js`, então para executá-lo no terminal, digitamos: `node index.js`

Teremos como saída a mensagem: “Servidor rodando: http://127.0.0.1:3000/”. Mas o que acontece se abrirmos outro terminal e executarmos o mesmo código?

Como você deve imaginar, receberemos um erro EADDRINUSE:

<> PS C:\Users\Emerson\Desktop\erros no node> node .\index.js node:events:491 throw er; // Unhandled 'error' event ^ Error: listen EADDRINUSE: address already in use 127.0.0.1:3000 at Server.setupListenHandle [as _listen2] (node:net:1485:16) at listenInCluster (node:net:1533:12) at doListen (node:net:1682:7) at process.processTicksAndRejections (node:internal/process/task_queues:83:21) Emitted 'error' event on Server instance at: at emitErrorNT (node:net:1512:8) at process.processTicksAndRejections (node:internal/process/task_queues:82:21) { code: 'EADDRINUSE', errno: -4091, syscall: 'listen', address: '127.0.0.1', port: 3000 } Node.js v18.10.0 Onde os trechos “Error: listen EADDRINUSE: address already in use 127.0.0.1:3000”, “code: 'EADDRINUSE', errno: -4091, syscall: 'listen', address: '127.0.0.1', port: 3000” estão selecionados com uma borda em vermelho.](assets/lidando-com-erros-node-js/imagem17.jpg)
Fazendo isso, vamos receber uma mensagem informando que o endereço já está em uso (“address in use”, daí a sigla `EADDRINUSE`) e também um objeto indicando a chamada ao sistema, assim como o endereço que já está em uso, a porta que tentamos ouvir e demais informações.

EADDRNOTAVAIL

Este erro é parecido com o `EADDRINUSE` porque é lançado ao executar um servidor Node.js em uma porta específica. Geralmente indica um problema de configuração com seu endereço IP, como vincular seu servidor a um IP estático. Pegando o código do exemplo anterior e mudando para um IP estático, temos o erro:

<> const http = require('http'); const hostname = '192.168.0.101'; const port = 3000; const server = http.createServer((_, res) ⇒ { res.statusCode = 200; res.setHeader('Content-Type', 'text/plain'); res.end('Olá mundo!'); }); server.listen(port, hostname, () ⇒ { console.log(`Servidor rodando: http://${hostname}:${port}/`); }); À direita é apresentado o terminal escrito: node:events:491 throw er; // Unhandled 'error' event ^ Error: listen EADDRNOTAVAIL: address not available 192.168.0.101:3000 at Server.setupListenHandle [as _listen2] (node:net:1468:21) at listenInCluster (node:net:1533:12) at doListen (node:net:1682:7) at process.processTicksAndRejections (node:internal/process/task_queues:83:21) Emitted 'error' event on Server instance at: at emitErrorNT (node:net:1512:8) at process.processTicksAndRejections (node:internal/process/task_queues:82:21) { code: 'EADDRNOTAVAIL', errno: -4090, syscall: 'listen', address: '192.168.0.101', port: 3000 } Onde o trecho do código “const hostname = '192.168.0.101'” e o trecho do terminal “Error: listen EADDRNOTAVAIL: address not available 192.168.0.101:3000” selecionados com uma borda em vermelho.](assets/lidando-com-erros-node-js/imagem18.jpg)
Com a mensagem indicando que o endereço informado não está disponível (em inglês, “address not available” - `EADDRNOTAVAIL`).

Para resolver esse problema, verifique se você tem o endereço IP correto.

Existem diversos outros códigos de erro de sistema! Listamos alguns dos mais comuns, mas você pode encontrar eles na página de manual do Linux caso queira consultar a lista completa.

Conclusão

Neste artigo, aprendemos como identificar quando um erro aparece na sua tela e como solucionar os erros mais comuns em Node.js. Você pode utilizar deste conhecimento para estruturar e desenvolver melhorias nas suas aplicações e tornar seu processo de trabalho mais ágil!

Existem vários outros códigos de erros e não tenho dúvidas que surgirão outros no futuro. Por isso, tenha a documentação sobre erros do Node.js como sua aliada para quando quiser saber mais detalhes.

Aqui na Alura, você pode dar o seu primeiro mergulho em programação back-end com a formação JavaScript para back-end, onde você verá desde as partes fundamentais de qualquer linguagem de programação até o consumo de APIs e criação de bibliotecas com Node.js.

E para mergulhos em regiões mais profundas com o Node.js, confira a formação Node.js com Express para aprender a construir backends para sites escaláveis.

Confira neste artigo:

Emerson Laranja

Emerson é um entusiasta do JavaScript, graduando em Engenharia da Computação pela Universidade Federal do Espírito Santo (Ufes) e faz parte do Scuba Team, um time de apoio