**Preparando o ambiente: primeiras instalações**

Neste curso usaremos o Node.js para construir uma API REST utilizando o framework Express. Para isso, é necessário fazer a instalação de algumas ferramentas. Se você já fez algum dos cursos que constam como pré-requisito, é provável que o ambiente já esteja pronto na sua máquina. Use as instruções abaixo para conferir!

## Node.js

1 - Abra uma janela do terminal em seu computador. Isso pode ser feito da seguinte forma:

* **Windows**: você pode usar a busca no menu iniciar e procurar por Prompt de Comando ou acessar via Menu Iniciar > Sistema do Windows > Prompt de Comando;
* **MacOs**: o Terminal está disponível no menu de Aplicações, dentro da pasta de Utilitários;
* **Linux (Ubuntu)**: o Terminal está disponível no menu de Programas/Aplicações. Caso não localize, pode estar dentro da pasta Utilitários.

2 - No Terminal, digite node --version ou node -v. Se retornar um número de versão, por exemplo, v18.16.0, o Node.js já está instalado.

Caso precise instalar, siga as instruções para cada sistema operacional que estão na [página inicial do Node.js](https://nodejs.org/en/download/). Este curso foi desenvolvido usando a versão 18.16.0.

O site do Node.js oferece duas opções para download, a LTS e a current (atual). Você pode escolher a versão LTS (long term support ou suporte a longo prazo) e clicar no botão correspondente para baixar e instalar normalmente, como qualquer outro programa.

## Visual Studio Code

O Visual Studio Code (VSC) é o editor que escolhemos utilizar para escrever nossos códigos durante este curso. Você pode utilizar outros editores de código caso prefira, porém recomendamos fortemente o uso do VSC para termos paridade com o que está sendo demonstrado nos vídeos.

Os links para baixar e instalar de acordo com o seu sistema operacional estão na [página inicial do VSC](https://code.visualstudio.com/).

### Utilizando o terminal do Visual Studio Code

Você pode utilizar o terminal integrado do VSC, que vai funcionar de forma bastante similar ao que é feito em vídeo.

Abra o terminal acessando no menu superior Terminal > New Terminal.

**Para saber mais: tipos de API**

Neste curso vamos desenvolver uma API REST, que é um tipo de API extremamente comum em programação web.

O termo REST (*representational state transfer* ou transferência de estado representacional) representa um padrão para desenvolvimento de APIs web utilizando o protocolo HTTP para transmissão de dados.

Para saber mais sobre o que é o REST, confira este [artigo sobre fundamentos do REST](https://www.alura.com.br/artigos/rest-conceito-e-fundamentos" \t "_blank) e também um Alura+ sobre o assunto.

* [vídeo Alura+ sobre REST](https://cursos.alura.com.br/extra/alura-mais/o-que-e-rest--c119)

Porém, pensando que APIs são interfaces que usamos em programação, existem várias outras formas de integrar programas ou serviços diferentes, que utilizam outros protocolos de comunicação, casos de uso e modos de acesso.

APIs podem ser desenvolvidas para diversos usos, por exemplo:

* Para uso interno de uma empresa, por exemplo, para fornecer dados como um serviço para um sistema maior.
* Para uso “externo”, como quando a API é desenvolvida pela empresa como um produto para ser utilizado por clientes.
* Podem ser abertas e de uso livre, desenvolvidas para uso da comunidade, sendo muito comuns para testes ou projetos de estudo.
* Podem ser de terceiros, como as que utilizamos para integrar serviços externos aos nossos produtos.

Além de usos diversos, as APIs também podem ser desenvolvidas seguindo outras arquiteturas além do REST. Seguem alguns exemplos:

* **APIs SOAP**: SOAP (*Simple Object Access Protocol* ou protocolo simples de acesso a objetos) utiliza o formato de dados XML e pode usar HTTP ou outros protocolos na comunicação. É um formato mais antigo que o REST e muito utilizado por aplicações de grande porte, porém mais lento que o REST.
* **Event-Driven APIs**: APIs orientadas a eventos, ao contrário das APIs REST, não dependem de requisições feitas pelo lado cliente para iniciar a comunicação. Nesse caso, o cliente ou clientes “inscritos” na API se comunicam com ela através de gatilhos de eventos, como, por exemplo, um novo registro de usuário.
* **APIs GraphQL**: o GraphQL é uma linguagem de consulta (query) de APIs e também um runtime para executar estas consultas. É uma alternativa ao REST que pode se conectar a diversas APIs e bases de dados diferentes e retornar objetos complexos através de uma única requisição.
* **APIs gRPC**: *Remote Procedure Calls* (ou chamadas procedurais remotas), desenvolvido pelo Google, é um framework baseado em HTTP2 para comunicação síncrona e assíncrona, que visa facilitar comunicação rápida e simplificada entre diversos serviços.

APIs são assuntos bastante extensos e as tecnologias estão em constante evolução. A partir do modelo REST você pode expandir para o estudo de outros tipos.

### Criação de um projeto Node

Com o terminal aberto na pasta criada para a construção do projeto, iniciaremos a criação de um novo projeto Node com o comando de terminal npm init -y, sendo -y de yes.

npm init -y

COPIAR CÓDIGO

Esse comando criará, na raiz do projeto, um novo arquivo chamado package.json com algumas informações padrão, que é o que acontece quando usamos a flag (bandeira) -y.

Feito isso, vamos abrir o Visual Studio Code na pasta do projeto, onde o arquivo JSON foi criado com as informações básicas. Com isso, podemos começar a trabalhar.

*package.json*:

{

"name": "3266-express-mongo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC"

}

COPIAR CÓDIGO

É importante que você utilize a mesma versão do Node.js que a instrutora. Com o terminal limpo, podemos usar o comando node -v para obter a versão, que no caso é v18.16.0.

node -v

COPIAR CÓDIGO

Em **Preparando o ambiente**, ensinamos a gerenciar as versões do Node.

A única coisa que precisamos fazer neste momento no arquivo package.json é, em qualquer parte do objeto, adicionar uma linha com a propriedade type que será do tipo module.

{

"name": "3266-express-mongo",

"version": "1.0.0",

"description": "",

"main": "index.js",

"type": "module",

// código omitido

COPIAR CÓDIGO

É importante lembrar que em JSON tudo é inserido como string, ou seja, entre aspas duplas.

A propriedade type definida como module será usada para **importar e exportar** as partes, as funções, os módulos do nosso projeto, isto é, da nossa API, utilizando a sintaxe mais moderna do JavaScript.

Também deixaremos material extra para você entender mais sobre como funciona a importação e exportação de módulos no JavaScript, algo que abordamos em cursos anteriores.

### Criação do servidor

A primeira coisa que fazemos quando vamos criar uma API que precisa fornecer informações para outras partes do sistema é **criar um servidor** para justamente fornecer os dados, servindo como ponto de conexão.

O primeiro arquivo que vamos criar, além do package.json, será um arquivo chamado server.js na raiz do projeto. Neste arquivo, criaremos um **servidor HTTP local** para podermos publicar os dados que a API precisa fornecer.

Existem diversas ferramentas e frameworks que utilizamos no dia a dia para simplificar esse processo de criação de servidor. Nós faremos isso neste projeto, mas para entender o passo a passo, faremos de uma forma um pouco mais nativa do Node, sem utilizar bibliotecas neste momento.

Primeiramente, no topo do arquivo, vamos fazer a importação de http de "http", como string.

*server.js*:

**import** http **from** "http";

COPIAR CÓDIGO

http é uma **biblioteca** nativa do Node; não é necessário instalação ou download com o comando npm no terminal, pois ao chamar no topo do arquivo, o Node já acessa os dados dessa biblioteca.

### Protocolo HTTP

O **protocolo HTTP** (Hypertext Transfer Protocol, ou seja, Protocolo de Transferência de Hipertexto) é um dos protocolos mais comuns na internet de comunicação.

Existem outros, como o protocolo de transferência de e-mail e de transferência de arquivos. Também deixaremos mais informações para você sobre isso. Porém, o protocolo HTTP é o mais comum, o que utilizamos na internet para nos comunicar, para que nossos sites possam acessar as informações e exibir as coisas na tela.

A comunicação HTTP ocorre **entre cliente e servidor**. Nesse caso, cliente não é a pessoa usuária, mas sim o computador que faz uma requisição do tipo HTTP para um servidor.

O **servidor** é um computador onde estão armazenados os arquivos que precisamos receber. No caso, por exemplo, se nosso navegador faz uma comunicação HTTP de uma requisição para "google.com", o Google vai ao servidor, pega o HTML e envia uma resposta para nosso cliente, ou seja, para o computador.

É crucial ter em mente que o protocolo HTTP, que usaremos na API e é utilizado em grande parte da internet, é baseado em **requisições** feitas de um **cliente** para um **servidor**. Nestas requisições, o servidor envia respostas para o cliente.

Vale reforçar que cliente e servidor são computadores que se comunicam através desse protocolo, definindo quais dados serão recebidos e enviados, entre muitas outras informações que vamos explorar durante o curso.

### Criando um servidor local HTTP

Após este breve esclarecimento sobre o HTTP, conseguimos retornar ao arquivo server.js e entender melhor o que iremos criar. Vamos criar um **servidor local HTTP** que simula um servidor na internet fornecendo essas informações. Para isso, usaremos os métodos da biblioteca HTTP, que é uma biblioteca do próprio Node.

Já importamos a biblioteca, então o próximo passo é criar uma constante chamada server, que será o nosso servidor local. Essa const receberá a biblioteca http seguida do método createServer(), que é um método da biblioteca HTTP. Este método requer uma função callback que recebe dois argumentos, denominados req (requisição) e res (resposta).

Feito isso, podemos abrir a função (=>) e adicionar chaves ({}).

*server.js*:

// código omitido

**const** server = http.**createServer**((req, res) => {

});

COPIAR CÓDIGO

Duas coisas vão acontecer quando criarmos um servidor. Primeiro, chamaremos o objeto res, ou seja, o objeto resposta, e dentro dele, a biblioteca HTTP terá um método chamado writeHead(). Este método é referente ao cabeçalho (ou header) da requisição HTTP.

**const** server = http.**createServer**((req, res) => {

res.**writeHead**();

});

COPIAR CÓDIGO

### HTTP headers

Vamos entender um pouco melhor o que é um **cabeçalho**?

Toda comunicação HTTP, tanto a requisição quanto a resposta, tem cabeçalhos. Os cabeçalhos contêm todas as informações necessárias para que a comunicação funcione corretamente.

GET / HTTP/1.1

Host: www.google.com

User-Agent: curl/7.68.0

Accept: text/javascript

X-Test: hello

COPIAR CÓDIGO

Conforme exibido acima, incluem o protocolo usado (neste caso, o HTTP); o Host para o qual a requisição é feita; o User-Agent, que designa quem faz a requisição, podendo ser um navegador, o Curl (programa de terminal) ou o Postman, por exemplo; e o tipo de dado aceito na requisição (Accept), que nesse caso, pode ser texto ou JavaScript.

As respostas também têm seus próprios cabeçalhos. O cabeçalho da resposta da requisição que fizemos, por exemplo, para www.google.com, trouxe a resposta 200 OK. O número 200 é o código de status HTTP, que significa que a comunicação foi bem-sucedida.

HTTP/1.1 200 OK

Date: Thu, 13 Jul 2023 00:19:01 GMT

Expires: -1

Cache-Control: private, max-age=0

Content-Type: text/html; charset=ISO-8859-1

Content-Security-Policy-Report-Only: object-src

COPIAR CÓDIGO

Você provavelmente conhece o famoso código 404, que aparece quando tentamos entrar em um site que não existe ou digitamos algo errado no endereço. Há uma lista extensa de códigos HTTP, mas o que mais gostamos de receber é o 200, que indica que tudo deu certo.

Vamos criar nossos próprios cabeçalhos durante o curso. Um cabeçalho de requisição é uma das partes mais importantes, pois precisa ser corretamente montado para que a comunicação ocorra sem problemas e não retorne erros.

No exemplo de cabeçalho acima, temos a data em que a requisição foi feita, o controle de cache (Cache-Control), e o tipo de conteúdo (Content-Type) definido como text/html, mas existem muitos outros dados que podem ser enviados e recebidos através de cabeçalhos.

### Dando continuidade à escrita da função

Agora que já entendemos o que deve conter em um cabeçalho de requisição, podemos prosseguir para a escrita de nossa função. Após utilizar o método writeHead(), o primeiro parâmetro que ele receberá será o número 200, que corresponde à resposta OK.

O segundo será um objeto JavaScript ({}) que terá um conjunto de chave e valor. Ambos serão strings. A chave será Content-Type e o valor será text/plain, que é o tipo de dado que iremos utilizar na nossa primeira requisição de teste.

*server.js*:

**const** server = http.**createServer**((req, res) => {

res.**writeHead**(200, { "Content-Type": "text/plain" });

});

COPIAR CÓDIGO

Em seguida, vamos chamar res novamente e utilizar o método end(), onde passaremos o texto que desejamos transmitir. Encerraremos a resposta com o texto "Curso de Node.js".

**const** server = http.**createServer**((req, res) => {

res.**writeHead**(200, { "Content-Type": "text/plain" });

res.**end**("Curso de Node.js");

});

COPIAR CÓDIGO

Com isso, na função createServer(), apenas passamos o cabeçalho da resposta, que será 200, e incluímos o tipo de conteúdo enviado nessa resposta. Por fim, passamos o próprio conteúdo, "Curso de Node.js".

### Criando uma conexão com o servidor

Finalizada a constante server, temos a variável que armazena todas as informações do servidor que está sendo criado. Em seguida, chamaremos o server na linha de código 8, que será um objeto grande com vários métodos e propriedades, junto ao método listen().

Esse método receberá dois parâmetros. O primeiro será um número, 3000, e falaremos mais sobre ele a seguir. O segundo será uma função callback.

Essa função não precisa receber nenhum parâmetro, então apenas abrimos e fechamos parênteses, passamos a arrow function (=>), e abrimos e fechamos chaves.

No escopo da função callback, vamos incluir somente um console.log() com a string "servidor escutando!". Esta etapa serve apenas para testes.

*server.js*:

server.**listen**(3000, () => {

console.**log**("servidor escutando!");

});

COPIAR CÓDIGO

O que o nosso server faz com o método listen()? "Listen" ("ouvir" em inglês) é um termo que utilizamos bastante quando trabalhamos com eventos. Um evento que vai acontecer em um servidor, por exemplo, é uma **conexão**. Alguém se conectou a esse servidor para fazer uma requisição e receber uma resposta.

Nesse caso, o método ouvirá o servidor para conexões que acontecerem nele na porta 3000. Então, o número 3000 é o número da **porta lógica** onde a conexão vai acontecer.

Para tornar o código mais legível, faremos uma refatoração no início do arquivo, logo após o import. Na linha 3, vamos criar uma constante chamada PORT e atribuir o valor 3000 a ela.

**const** PORT = 3000;

COPIAR CÓDIGO

Normalmente, usamos esse padrão do nome da variável com todos caracteres maiúsculos quando queremos passar informações fixas, informações estáticas.

Feito isso, dentro de server.listen(), podemos substituir o 3000 por PORT.

server.**listen**(PORT, () => {

console.**log**("servidor escutando!");

});

COPIAR CÓDIGO

A porta 3000 é a porta de comunicação que será utilizada na API. Um computador tem milhares de portas lógicas que podem ser utilizadas. Algumas são padrão para certos tipos de comunicação.

Por exemplo: navegadores usam a 443 ou a 80; bancos de dados também têm suas próprias portas. Algumas portas são de uso geral, sendo a 3000 uma delas.

Falaremos mais sobre **portas** no material extra!

Resultado do arquivo *server.js* até o momento:

**import** http **from** "http";

**const** PORT = 3000;

**const** server = http.**createServer**((req, res) => {

res.**writeHead**(200, { "Content-Type": "text/plain" });

res.**end**("Curso de Node.js");

});

server.**listen**(PORT, () => {

console.**log**("servidor escutando!");

});

COPIAR CÓDIGO

### Executando o arquivo

Agora precisamos apenas executar o arquivo e verificar se o nosso servidor está no ar e servindo arquivos. Para isso, vamos retornar ao terminal na pasta correta e pedir para o Node executar server.js com o comando abaixo:

node server.js

COPIAR CÓDIGO

O terminal deverá retornar o console.log() de "servidor escutando!". Mas, além disso, precisamos verificar se algum arquivo é servido.

Para isso, podemos usar o navegador comum para acessar "localhost:3000". Será exibida a informação "Curso de Node.js", único dado transferido no nosso servidor por enquanto.

Agora nosso servidor está ativo, recebe requisições na porta 3000, e retorna para nós a string "Curso de Node.js"!

**Para saber mais: import e export**

Durante seus estudos de JavaScript, você pode ter encontrado códigos diferentes que fazem o mesmo trabalho. Por exemplo:

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

**const** **LivroController** = require('../controllers/livroController.js');

COPIAR CÓDIGO

**import** express **from** 'express';

**import** **LivroController** **from** '../controllers/livroController.js';

COPIAR CÓDIGO

Ambos os códigos fazem a importação de **módulos** para um arquivo. Estes módulos podem vir de uma biblioteca externa (como express) ou de outros módulos desenvolvidos internamente no projeto (como LivroController).

Este detalhe aparece com bastante frequência no dia a dia do desenvolvimento com JavaScript, tanto na documentação de bibliotecas quanto em fóruns de discussão e exemplos de código.

Para entender com mais detalhes a razão de existirem as duas formas e as diferenças entre elas, confira este [artigo completo sobre importação e exportação de módulos](https://www.alura.com.br/artigos/guia-importacao-exportacao-modulos-javascript).

**Para saber mais: componentes de um projeto Node.js**

Até este momento, já criamos um projeto Node.js e um servidor local.

Sempre que criamos um projeto Node.js do zero ou começamos a trabalhar em um projeto já existente, o primeiro arquivo que criamos ou que consultamos é o package.json.

O arquivo package.json é o arquivo principal de qualquer projeto em Node.js. É utilizado como **arquivo manifesto**, centralizando todas as informações necessárias para entender os componentes de um projeto, por exemplo:

* Bibliotecas e dependências utilizadas em produção e desenvolvimento.
* Scripts utilizados pelo projeto para executar o projeto, como subir servidores, rodar testes, compilar arquivos para produção etc.
* Informações sobre versionamento, repositório do código, contato do time responsável.
* Informações sobre configurações diversas.
* Dados necessários para a publicação do pacote no NPM (gerenciador de pacotes do Node.js).

O package.json está estruturado conforme o exemplo abaixo:

{

"name": "alura-curso-node",

"version": "1.0.0",

"description": "Módulo de primeiros passos no curso de Node.js",

"main": "index.js",

"scripts": {

"start": "node index.js",

"test": "jest"

},

"author": "Juliana",

"license": "ISC",

"dependencies": {

"modulo-1": "^1.0.0",

"modulo-2": "^1.0.3",

"modulo-3": "^2.1.0"

},

"devDependencies": {

"nodemon": "^2.0.15",

"jest": "^27.2.1"

}

}

COPIAR CÓDIGO

Vamos entender e detalhar alguns desses atributos:

* name: é onde se define o nome pelo qual seu módulo será chamado.
* version: versão atual do módulo. O NPM utiliza o chamado *versionamento semântico* (SemVer). Você pode saber mais sobre [versionamento e como ele é utilizado neste artigo](https://www.alura.com.br/artigos/versionamento-semantico-breve-introducao" \t "_blank).
* description: define o que será este módulo. É ideal que seja uma descrição curta e clara sobre o objetivo principal.
* main: define o ponto de entrada da aplicação.
* scripts: essa sessão tem alguns scripts pré-definidos, mas você também pode definir os seus personalizados. [Leia sobre scripts no npm Docs](https://docs.npmjs.com/cli/v8/using-npm/scripts" \t "_blank) ter mais informações.
* keywords: é um array de palavras-chave sobre o projeto.
* author: são dados de autoria, podendo conter nome, e-mail, site etc.
* license: é a licença escolhida para o projeto. Existem diversas opções de licenças, caso tenha curiosidade você pode conferir [a lista completa de licenças no site do SPDX](https://spdx.org/licenses/" \t "_blank).
* dependencies: define a lista de pacotes necessários para executar seu projeto num ambiente de produção.
* devDependencies: define a lista de pacotes necessários para executar o projeto em um ambiente de desenvolvimento e testes. Existem outras configurações que podem existir nesse arquivo, como repositório do Git, homepage, peerDependencies, entre outras que podem ser visualizadas na [documentação oficial do NPM](https://docs.npmjs.com/cli/v8/configuring-npm/package-json" \t "_blank).

## Criando rotas

Acessaremos a [página da Alura](https://www.alura.com.br/) no navegador. Ao clicar em qualquer parte da página inicial, por exemplo, em "Escola de Programação", a barra de navegação mostrará "[/escola-programacao](https://www.alura.com.br/escola-programacao)".

Isso significa que acessamos uma parte específica da "alura.com.br", onde estão as informações que precisamos sobre a escola de programação. Você já deve ter percebido que isso funciona para qualquer site na internet.

Quando construímos APIs, como neste curso, utilizamos um conceito parecido, chamado **rotas**. É basicamente o mesmo princípio, onde temos endereços. No vídeo anterior, por exemplo, acessamos o endereço localhost:3000. Esse endereço é local, porque nosso servidor é local, mas quando estiver na internet, terá um endereço na internet.

Para a API, precisamos definir o **recurso** que queremos acessar, que pode ser livros, editoras, admin, e assim por diante. Tudo isso é feito através de rotas. Afinal, o produto final da API é um conjunto de endereços HTTP, um conjunto de links, isto é, de rotas. É isso que começaremos a construir agora.

### Conjunto de rotas

Neste primeiro teste, vamos estabelecer no arquivo server.js um **conjunto de rotas** através de um **objeto**. Para isso, criaremos abaixo de PORT uma constante chamada rotas, que receberá uma abertura de chaves ({}).

Por enquanto, será apenas um conjunto de chave e valor. Então, entre as chaves, passaremos uma string contendo apenas uma barra (/), e após dois-pontos passaremos a mesma informação anterior ("Curso de Node.js").

*server.js*:

**const** rotas = {

"/": "Curso de Node.js"

};

COPIAR CÓDIGO

Quando usamos apenas /, normalmente, nos referimos à **rota base**, que não disponibiliza nenhum recurso específico. Nesse caso, declaramos que, ao acessar a rota base, ou seja, a URL base, será exibida a informação "Curso de Node.js".

Agora vamos passar essa informação para o escopo de createServer(). A primeira linha da função writeHead continua a mesma. No entanto, em vez de passarmos a string diretamente para o método res.end(), passaremos rotas[req.url].

**const** server = http.**createServer**((req, res) => {

res.**writeHead**(200, { "Content-Type": "text/plain" });

res.**end**(rotas[req.url]);

});

COPIAR CÓDIGO

A notação de colchetes é usada para passar uma **variável** como propriedade dentro do objeto rotas. Normalmente, usamos ponto, mas como passamos uma informação variável, usamos a notação de colchetes, que é um recurso próprio de objetos JavaScript.

Porém, o que é req.url? req é um dos argumentos recebidos pela função createServer(). Ou seja, toda vez que uma requisição é recebida, ela traz consigo, entre várias outras informações, uma propriedade chamada url.

### Reiniciando o servidor

Para verificar isso na prática, faremos o teste novamente, diretamente no navegador. Mas antes, vamos retornar ao terminal e reiniciar o servidor.

Podemos usar o comando node server.js, da mesma forma que fizemos anteriormente, mas para facilitar o processo de desenvolvimento, instalaremos uma biblioteca para não precisar desativar e ativar o servidor toda vez que alterarmos algo no projeto.

Uma vez que o servidor é iniciado, ele não detecta mais as alterações que fazemos no código. Para resolver esta questão, vamos instalar uma biblioteca muito conhecida para todas as pessoas que trabalham com Node, chamada ***Nodemon***.

Para instalar, utilizamos o comando npm install ou npm i, seguido de nodemon@3.0.1 para instalar exatamente a versão usada no curso.

npm install nodemon@3.0.1

COPIAR CÓDIGO

Esta é uma biblioteca muito leve, que instala rapidamente e já fica disponível no arquivo package.json. Agora vamos editar a parte de scripts do arquivo package.json e adicionar um script para o Nodemon gerenciar o servidor.

Na seção scripts do package.json, adicionaremos um script que chamaremos de dev, que pode ser posicionado tanto antes quanto depois do script test já existente. O valor dele será nodemon server.js. Dessa forma, utilizamos o Nodemon para executar o arquivo server.js.

*package.json*:

"scripts": {

"dev": "nodemon server.js",

"test": "echo \"Error: no test specified\" && exit 1"

}

COPIAR CÓDIGO

De volta ao terminal, executaremos o script através do comando npm run dev.

npm run dev

COPIAR CÓDIGO

Com isso, o Nodemon está iniciado e irá monitorar todas as nossas alterações. A mensagem "servidor escutando!" foi exibida, que é o console.log() inserido anteriormente.

### Fazendo uma requisição

Agora podemos voltar ao navegador e realizar uma **nova requisição** para localhost:3000. A página deverá carregar com sucesso exibindo a mensagem "Curso de Node.js".

Caso queira confirmar que a informação foi modificada, você pode alterar no objeto rotas de "Curso de Node.js" para "Curso de Express API", salvar e recarregar a página no navegador para visualizar o resultado.

*server.js*:

**const** rotas = {

"/": "Curso de Express API"

};

COPIAR CÓDIGO

A modificação já terá sido feita, ou seja, o servidor está de pé e nossa rota / também está funcionando, porque não inserimos nada depois de localhost:3000, após a URL base.

Dessa forma, concluímos que o navegador fez uma **requisição HTTP** para o **servidor** que está em localhost na porta 3000, e esse servidor devolveu a informação referente à rota /. Isso quer dizer que url é uma propriedade do objeto req, ou seja, da requisição que está sendo gerida através do servidor HTTP.

### Visualizando outras rotas

Vamos fazer mais um teste no objeto rotas para visualizarmos algumas outras rotas em andamento. Adicionaremos duas rotas após a rota /. A primeira será chamada /livros, que terá como valor a string "Entrei na rota livros". Por fim, a terceira rota será chamada /autores e terá um valor de string "Entrei na rota autores".

**const** rotas = {

"/": "Curso de Express API",

"/livros": "Entrei na rota livros",

"/autores": "Entrei na rota autores"

};

COPIAR CÓDIGO

Podemos recarregar a página e testar novamente.

No terminal, você pode verificar todas as vezes em que a biblioteca Nodemon foi executada e que o servidor foi reiniciado.

Com o navegador aberto, em vez de localhost:3000, vamos para localhost:3000/livros. Será acessada a rota /livros através de req.url e exibida a string "Entrei na rota livros" na tela. Com a rota /autores acontecerá o mesmo, retornando a string "Entrei na rota autores."

Resultado do arquivo *server.js* até o momento:

**import** http **from** "http";

**const** PORT = 3000;

**const** rotas = {

"/": "Curso de Express API",

"/livros": "Entrei na rota livros",

"/autores": "Entrei na rota autores"

};

**const** server = http.**createServer**((req, res) => {

res.**writeHead**(200, { "Content-Type": "text/plain" });

res.**end**(rotas[req.url]);

});

server.**listen**(PORT, () => {

console.**log**("servidor escutando!");

});

COPIAR CÓDIGO

### Criando o arquivo .gitignore

Agora que já temos rotas para a nossa API, vamos criar um novo arquivo na raiz e chamá-lo de .gitignore. Dentro dele, vamos inserir a informação node\_modules e salvar.

*.gitignore*:

node\_modules

COPIAR CÓDIGO

Fazemos isso porque, ao realizar a primeira instalação de biblioteca no projeto, a pasta "node\_modules" é criada, e ela não deve ser enviada para o GitHub.

Dito isso, agora você pode salvar seu projeto no GitHub sabendo que a pasta "node\_modules", que ainda vai crescer com outras coisas que instalaremos, não será enviada para seu repositório.

## Conclusão

Neste vídeo, criamos rotas locais por extenso para nosso servidor trabalhar com diferentes rotas e exibir informações distintas. Agora temos preparo para evoluir com as bibliotecas que serão utilizadas durante o curso. **Vamos lá!**

**Para saber mais: o que são portas**

Além da URL, requisições HTTP também precisam de **portas** específicas para que a comunicação ocorra com sucesso.

O que são portas? Também chamadas de **portas lógicas**, elas são gerenciadas pelo sistema operacional e agem como pontos onde conexões de rede acontecem. Por exemplo, a comunicação com a API que estamos desenvolvendo se dá através da porta 3000.

O número padrão de portas é 65.535. As portas são identificadas por números padronizados e muitas portas estão associadas a processos específicos, por exemplo:

* 5432 é usada para conexão com serviços do banco de dados PostgreSQL.
* 3306/3307 são usadas em conexões com o banco MySQL.
* 587 utilizada para conexões SMTP (Simple Mail Transfer Protocol, o protocolo de envio e recebimento de e-mails).
* 80 é a porta padrão de conexões HTTP, usada pelos navegadores.

Algumas portas são de “uso comum” e podem ser utilizadas por qualquer serviço, como a 3000 que usamos no curso.

Portas são assunto importante quando falamos de protocolos de comunicação. Não deixe de conferir nosso curso de [HTTP](https://cursos.alura.com.br/course/http-entendendo-web-por-baixo-dos-panos) para entender mais.

**Para saber mais: gerenciadores de pacotes**

Sempre que vamos trabalhar com um projeto em Node.js do zero, uma das primeiras coisas que fazemos é criar um arquivo de configuração utilizando o comando npm init ou yarn init, assim como para todas as instalações de libs externas utilizamos o comando npm install <nome do pacote> ou yarn add <nome do pacote>.

Ambos são **gerenciadores de pacotes**, sendo que NPM é acrônimo de **Node Package Manager** ou gerenciadores de pacotes do Node.

## Mas o que são exatamente estes gerenciadores?

Gerenciadores de pacotes são repositórios de código aberto nos quais devs disponibilizam soluções para o uso da comunidade. Estas soluções são programas que outras pessoas desenvolveram e que utilizamos para ganhar tempo no desenvolvimento de nosso próprio código, e vão desde libs (bibliotecas pequenas e específicas) até frameworks com vários recursos prontos. Independente do tamanho, chamamos esses conjuntos de código de pacote (ou módulo).

Algumas dessas bibliotecas são desenvolvidas por times para resolver algum problema específico que tiveram que enfrentar. Depois, elas são disponibilizadas para que outras pessoas com o mesmo contratempo aproveitem e também utilizem.

Outras bibliotecas são disponibilizadas por empresas de software, que utilizam as plataformas dos gerenciadores para a distribuição de suas soluções de código. E sendo de código aberto, você também pode criar e publicar a sua lib para outras pessoas baixarem e instalarem em seus projetos.

## Instalação local vs global

Estes pacotes de código podem ser instalados **localmente**, estando disponíveis somente para o projeto no qual foi instalado através da pasta node\_modules, e **globalmente**, sendo instalados em um diretório geral do NPM em seu computador e disponibilizados de forma geral, sem a necessidade de instalar separadamente em cada projeto.

Na maior parte das vezes, você vai utilizar a opção local com os comandos npm install <nome do pacote> ou yarn add <nome do pacote>, pois fica mais fácil fazer o gerenciamento de versão das libs que utilizamos e, muitas vezes, um pacote que instalamos “puxa” um ou vários outros pacotes auxiliares que ele precisa para funcionar. O ideal é não poluir o diretório global com libs que poderão ser utilizadas em somente um projeto.

Algumas libs e frameworks mais complexas vão solicitar que a instalação seja feita globalmente para funcionar. Sempre vale a pena consultar a documentação de cada uma! Para fazer uma instalação global de pacotes, utilizamos os comandos npm install -g <nome do pacote> ou yarn add global <nome do pacote>.

O NPM e o YARN têm algumas pequenas diferenças nos comandos e na forma como lidam com os pacotes. Você pode ler a [documentação do NPM](https://docs.npmjs.com/) e do [YARN](https://yarnpkg.com/) para ter mais informações sobre como os comandos funcionam em cada um.

**Criando rotas**

Marcos está criando um novo projeto no Node.js e resolveu usar o módulo HTTP para auxiliá-lo na criação do servidor e suas rotas.

Depois de usar o comando import http from “http”; para carregar o módulo HTTP, qual seria a forma adequada de criar o servidor?

Top of Form

* Alternativa correta
* **const** server = http.**listening**((req, res) =>{ /\* implementar o código \*/ });
* Alternativa correta
* **const** server = http.**createServer**((req, res) => { /\* implementar o código \*/ });

Através do método createServer toda vez que o servidor HTTP receber uma requisição, a função de callback passada para o método será executada.

* Alternativa correta

**const** server = http.**listen**((req, res) =>{ /\* implementar o código \*

Bottom of Form

**niciando com Express**

E para tudo que é recorrente e repetitivo, nós utilizamos **bibliotecas**.

As bibliotecas são porções de código predefinidas que facilitam nosso trabalho no desenvolvimento.

No nosso caso, o Express é um framework que vamos utilizar para nos auxiliar no gerenciamento de rotas e do servidor HTTP.

Chamamos de framework um conjunto maior de bibliotecas ou de códigos.

As bibliotecas costumam ser um pouco específicas no que fazem. Então, quando há um grupo maior de código que executa muitas tarefas, nomeamos isso de framework. E o Express é o **framework de Node.js para servidores HTTP** mais famoso e um dos mais utilizados no mercado.

Para começarmos a utilizar o Express, vamos abrir o Terminal. Preciso pressionar "CTRL + C" para interromper o servidor, pois faremos mais algumas instalações no nosso projeto. Não tem como manter o servidor ativo, pois precisaremos reiniciá-lo.

Já estou na pasta do projeto, agora vou executar npm install express@4.18.1. Essa é a última versão disponível no momento da criação deste curso e gostaria que instalassem exatamente a mesma versão para evitar possíveis problemas com versões futuras.

A instalação é rápida, pois o pacote é leve. Vamos retornar ao Visual Studio Code e começar a utilizar os métodos do Express para aprimorar o nosso código, tornando-o mais eficiente e preparado para futuras ampliações da nossa API.

A primeira coisa que farei é criar uma pasta chamada "src" na raiz do projeto. "src" é a abreviação de source code (código fonte), que é o nome padrão para a pasta onde ficam todos os arquivos da nossa API. Dentro de "src", criarei um arquivo "app.js", onde iniciaremos o Express.

Dentro desse arquivo, a primeira coisa que farei é incorporar o Express com o comando import express from "express";. Essa linha importa toda a biblioteca que instalamos com o npm, nos fornecendo todos os métodos para utilizarmos.

Em seguida, criarei uma constante chamada app. Essa constante será uma instância do Express e receberá como valor o resultado de express(). Portanto, ao executar a função express, todas as funções do Express serão atribuídas à variável app.

**import** express **from** "express";

**const** app = **express**();

COPIAR CÓDIGO

Com o Express iniciado, conseguiremos realizar alguns testes para entender como os métodos do Express funcionam com as rotas. Criarei nossa primeira rota com app.get().

Esse método get() receberá dois parâmetros, assim como antes. O primeiro será uma string e passarei apenas uma barra ("/"), indicando a URL base da nossa API. O segundo parâmetro será uma função de retorno (ou callback), que também recebe os dois parâmetros que já vimos antes: req e res (abreviações para requisição e resposta). Também criarei uma arrow function, abrindo e fechando chaves em seguida.

Dentro das chaves, chamarei res.status(200).send('Curso de Node.js');. Com essas instruções, o servidor retornará o status 200 e enviará a string "Curso de Node.js". Vamos falar mais sobre o status 200 em breve.

Bloco de código final:

**import** express **from** 'express';

**const** app = **express**();

app.**get**("/", (req, res) => {

res.**status**(200).**send**("Curso de Node.js");

});

COPIAR CÓDIGO

Estamos passando para o Express a responsabilidade de gerenciar as rotas que estávamos criando manualmente antes, no arquivo server.js com um objeto JavaScript. Então, a partir de agora, quem vai cuidar de ouvir as rotas sendo chamadas, no caso que estamos fazendo via navegador, administrando as requisições e as respostas, é o próprio Express.

O que antes era feito diretamente pela biblioteca interna do Node, o HTTP, será agora responsabilidade do Express: A mesma operação que fizemos no server.js, ao escrever o cabeçalho e passar o status code 200 de OK e o conteúdo da resposta.

Mas antes de testarmos novamente, preciso fazer o módulo app, que acabamos de criar, se comunicar com o servidor. Então, vou exportar no final do arquivo app.js com export default app, para conseguirmos exportar esse módulo.

Agora, retorno ao arquivo "server.js", vou comentar a linha import http from http por enquanto e, em seu lugar, vou importar o módulo app. Fica assim: import app from "./src/app";.

A porta vai permanecer como a 3000, por enquanto.

// import http from "http";

**import** app **from** "./src/app";

**const** PORT = 3000;

COPIAR CÓDIGO

Na parte da criação do servidor, ele vai deixar de ser create server. Não precisamos mais da constante server, porque agora quem vai criar o servidor é app, ou seja, o Express. Então, por enquanto, vou comentar a linha a seguir, apenas para verificar se está tudo funcionando corretamente.

// const server = http.createServer((req, res) => {

// res.writeHead(200, { "Content Type": "text/plain" })

// });

COPIAR CÓDIGO

No listen, o método que está monitorando as conexões feitas com o servidor, não se conectará mais ao server: a conexão será feita agora com o app, no qual estão todos os métodos do Express.

Portanto, teremos app.listen. Esse método continuará funcionando da mesma forma: o primeiro parâmetro será o número da porta e o segundo será o que desejamos que aconteça quando o servidor escutar qualquer conexão. Então, teremos console.log("servidor escutando!").

app.**listen**(PORT, () => {

console.**log**("servidor escutando!");

});

COPIAR CÓDIGO

Agora podemos retornar ao terminal, reiniciar o servidor com o comando npm run dev. Recebi um erro, que já identifiquei: em server.js, quando importei o app from "./src/app", esqueci de adicionar .js. Então, acrescento apenas .js em app (import app from "./src/app.js"). Não podemos esquecer disso, porque muitas vezes o Visual Studio Code não adicionará isso automaticamente para nós.

Ao salvar o arquivo, o servidor notou a alteração e não está mais mostrando nenhum erro na tela. O servidor está escutando, podemos testar no próprio navegador.

A rota /autores que testamos anteriormente não existe mais, mas a rota / existe. Então, ao recarregar, o curso de node continua funcionando. Ou seja, passamos com sucesso os métodos que estavam sendo executados pela biblioteca HTTP para o Express.

Tudo que já havíamos comentado antes sobre o código HTTP, podemos agora deletar do nosso "server.js". Da mesma forma, o objeto rotas, criado para também criar algumas rotas manualmente e manipular suas solicitações, também não será mais usado porque quem o utilizava era a biblioteca HTTP. Portanto, também podemos deletá-lo.

No entanto, até o momento só temos uma rota "/" que retorna uma string para nós. Agora é a hora de começarmos a criar rotas verdadeiras na nossa aplicação. Vamos lá.

# Para saber mais: partes da requisição e resposta

O protocolo de comunicação HTTP é baseado em **requisições e respostas** e na **comunicação cliente-servidor**.

Todo processo de requisição-resposta usando HTTP é **sempre iniciado pelo lado cliente da requisição**. O lado servidor da requisição nunca envia “ativamente” nenhuma resposta para o lado cliente sem ser como resultado de uma requisição.

## Partes da requisição

As partes que podem compor uma requisição são:

* **URL**, também chamado de caminho ou rota, sempre iniciado com http:// ou https://.
* **Header**, também chamado de cabeçalho, envia informações referentes ao cliente ou ao tipo de requisição. Alguns dados enviados através dos cabeçalhos são:
  + host é o domínio do servidor que receberá a requisição.
  + User-agent identifica o cliente, por exemplo, dados do navegador de onde está saindo a requisição.
  + Content-Type é o formato do dado que está sendo enviado no body da requisição, por exemplo JSON, string etc. Confira a [lista completa de tipos de dados e como devem ser declarados no header](https://www.iana.org/assignments/media-types/media-types.xhtml).
  + Authorization são as credenciais de autenticação para acesso a recursos protegidos.
  + Accept especifica o formato de retorno esperado na resposta, por exemplo, JSON.
* **Body** ou corpo da requisição, onde são trafegados dados enviados pelo cliente para serem recebidos pelo servidor, normalmente utilizado para dados mais estruturados e em requisições POST, PUT or PATCH. O tipo de dado enviado pelo body deve ser o mesmo especificado no Content-Type, por exemplo, application/json.
* **Parâmetros** são inseridos na URL para envio de dados específicos, muito utilizados, por exemplo, para envio de informações variáveis como termos de buscas, IDs etc.
* **Método HTTP**, entre os aceitos pela rota, especifica o tipo de operação solicitado pelo cliente. Os mais comuns são os métodos GET, POST, PUT e DELETE. Vamos trabalhar com estes métodos com mais profundidade durante o curso.

Confira a [lista completa de informações dos cabeçalhos na documentação do MDN](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers).

## Partes da resposta

A resposta a uma requisição HTTP é enviada pelo lado servidor da comunicação, de volta ao lado cliente. A resposta contém informações referentes à requisição, que podem ser uma confirmação de operação, dados solicitados ou mesmo mensagens de erro pertinentes em caso de falha em algum ponto da comunicação.

As partes que podem compor a resposta são:

* **Status da resposta**, que contém a versão HTTP utilizada, o código de status e a mensagem de status. Por exemplo, HTTP/1.1 200 OK.
* **Headers** ou cabeçalhos, com informações adicionais sobre a resposta ou o conteúdo da resposta. Por exemplo:
  + Content-Type;
  + Content-Length, que corresponde ao tamanho em bytes do corpo da resposta;
  + Cache-Control, que são as instruções de cache para a resposta;
  + Set-Cookie, que adiciona um valor de cookie ao navegador. Caso queira saber mais, confira [este artigo sobre o que são cookies e como são utilizados](https://www.alura.com.br/artigos/o-que-sao-cookies-como-funcionam).
* **Body**, o corpo da resposta, que contém os dados ou o conteúdo solicitado pelo cliente através da requisição e enviado pelo servidor. O formato de dados do body vai depender do formato especificado em Content-Type, por exemplo JSON.

Veja a [lista completa de propriedades dos cabeçalhos neste artigo](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields).

Caso você ainda não tenha feito, acesse nosso [curso de HTTP](https://cursos.alura.com.br/course/http-entendendo-web-por-baixo-dos-panos) para aprender ainda mais sobre tudo que acontece por baixo dos panos em uma requisição HTTP.

**Criando a rota livros**

Dentro de "app.js", logo abaixo de onde iniciamos o express(), vou criar uma constante chamada livros, que por enquanto será igual a um array. Portanto, vou criar colchetes []. Dentro deste array, vou criar dois livros. Portanto, será um array contendo dois objetos livro, cada um com um id e um titulo. Por exemplo, um com id: 1 e título: "O Senhor dos Anéis", e outro com id: 2 e título: "O Hobbit".

const livros = [

{

id: 1,

título: "O Senhor dos Anéis"

},

{

id: 2,

título: "O Hobbit"

}

];

COPIAR CÓDIGO

Agora temos um array que está simulando alguns dados, como se fosse nossa base de dados de livros.

Em relação às rotas, temos a rota que bate na URL base. Podemos adicionar mais uma rota para retornar nossa lista de livros. Então, abaixo do primeiro app.get("/"), vou criar outro app.get() e, agora, o primeiro parâmetro desta função get(), em vez de ser "/", será "/livros".

O segundo parâmetro permanece o mesmo. Ele é uma callback function que recebe dois parâmetros: req e res, que já conhecemos.

Criarei uma arrow function {} e o retorno será res.status(). Novamente, vou passar aqui como valor de status 200, que é o status de OK.

Agora, não estamos mais trabalhando com uma string, mas com um tipo de dado estruturado: um array de objetos. Então, precisamos passar o método json(), em vez do send(), que é usado para dados mais simples. Dentro de json(), vou referenciar nossa variável livros, que é nosso array de livros.

app.**get**("/", (req, res) => {

res.**status**(200).**send**("Curso de Noje.js");

});

app.**get**("/livros", (req, res) => {

res.**status**(200).**json**(livros);

});

COPIAR CÓDIGO

Vou verificar no terminal se o nodemon gerou algum erro. Não há nenhum erro, então, nosso servidor está funcionando. Agora, podemos voltar ao navegador e fazer um novo teste, acessando "localhost:3000/livros".

A visualização pode variar um pouco dependendo do navegador que você está usando. No Firefox, por exemplo, já temos um visualizador para objetos do tipo JSON. Mas, se fizermos o mesmo teste no Google Chrome, a visualização fica um pouco diferente.

### O que é o JSON?

Vamos voltar um pouco para entender o que é esse .json(). O JSON é uma **notação de objeto** que se baseia em um objeto JavaScript. Deixarei mais informações detalhadas sobre ele nas atividades (JSON é usado para estruturar dados em um formato que tanto humanos quanto máquinas possam ler).

Mas por que temos que usar o .json()? Para que a resposta seja interpretada corretamente.

Então, nossa resposta é uma resposta 200, ou seja, de sucesso. Agora, com o método JSON, é sinalizado que o que está sendo enviado nesta resposta não é mais uma string, como anteriormente, mas um tipo JSON. Este método pega nosso array simples em JavaScript, o converte para JSON (que é o formato de dados padrão, atualmente, de APIs REST).

Utilizamos os métodos do Express para gerenciar duas rotas: a rota base ("/") e a rota livros ("/livros").

### O que é o .get()?

O get() é o método que estamos utilizando, significa que queremos **obter dados**.

Existem outros métodos para diferentes ações que precisamos executar em nossa API, como enviar ou atualizar dados. A seguir, vamos descobrir quais são as diferentes formas pelas quais podemos interagir com a nossa API usando outros métodos HTTP.

# Criando registros com POST

Lá no início do curso, comentamos que um sistema, por exemplo, de uma livraria, deve ser capaz não só de consultar livros, mas de adicionar livros também. É isso que vamos fazer agora.

Por enquanto, vamos continuar trabalhando no array local, que é o array da memória que estamos utilizando.

Então, vamos pensar como fazer para criar uma nova rota e conseguir não só buscar itens, mas adicionar itens, **criar registros** no nosso array, no nosso "banco de dados".

Vou criar uma nova rota para isso. Embaixo de app.get, vou criar app.post, o nome do método HTTP que utilizamos para criar recursos, criar novos recursos.

Qual rota o post vai acessar? Também vai acessar a mesma rota, /livros, já que estamos falando de adicionar livros. E o segundo parâmetro continua sendo os mesmos que estávamos trabalhando, que é req de requisição, res de resposta, criando uma função de seta (arrow function), abrindo e fechando chaves.

app.**post**("/livros", (req, res) => {

})

COPIAR CÓDIGO

Então, o que vai dentro do post?

Se estamos trabalhando com um array normal do JavaScript, vamos utilizar os métodos do JavaScript para manipulação de array. Então, o nosso app.post em /livros vai fazer um livros.push. E o que vai receber?

De onde vão sair os dados que vão ser utilizados para criar um novo livro? Eles vão sair de req.body, encerrado com ponto e vírgula.

app.**post**("/livros", (req, res) => {

livros.**push**(req.body);

})

COPIAR CÓDIGO

Então, o que é req.body?

Já abordamos anteriormente o req.url. Sempre que falamos de req estamos falando do objeto requisição, que está sendo recebido dentro da função que é chamada em app.post. Todo o gerenciamento de ir na rota, pegar as informações e criar esses dois objetos, tanto o req quanto o res, é responsabilidade do Express.

Assim, o Express cria esse objeto req e dentro dele tem uma propriedade que chama body, ou seja, corpo. Que corpo é esse? É o corpo da requisição.

Vamos fazer novamente uma pausa para entender o body.

Falamos anteriormente sobre cabeçalhos (headers): Toda requisição e toda resposta HTTP tem cabeçalhos com as informações necessárias.

Porém, algumas requisições e respostas HTTP também têm body, também têm corpo. Quando só queremos buscar coisas com get, nossas requisições não precisam ter corpo, só precisam ter cabeçalho com as informações.

Porém, quando queremos enviar dados para serem criados, por exemplo, post cria um novo registro, nossa requisição também tem que ter um body, também tem que ter um corpo. No caso, a requisição HTTP tem que enviar os dados que queremos que sejam armazenados.

Normalmente, fazemos isso no formato de objeto e a resposta da requisição também tem body, que é o corpo, que normalmente é onde mandamos a informação de volta se isso deu certo ou se deu errado. Então, lembre-se sempre de que toda requisição HTTP tem cabeçalho e quando queremos enviar dados, por exemplo, em um post, temos que ter um corpo da requisição.

E esse valor fica guardado dentro do objeto requisição, dentro da propriedade body, criada pelo Express.

Então, voltando para o código, agora sabemos que o Express vai pegar o corpo dessa requisição e vai conseguir fazer um push, porque isso vai ser um objeto de livro, dentro do nosso array.

Antes de testarmos, tem um detalhe bem importante que temos que incluir lá no começo do nosso arquivo app.js, embaixo de onde criamos a const app. Trata-se do app.use, uma função, e vou passar como parâmetro uma outra função, que vai ser express.json.

**app**.use(express.json());

COPIAR CÓDIGO

Assim, temos aqui uma função executando outra função. Isso se chama **middleware**.

No caso do Express, esses middlewares são utilizados para ter acesso às requisições e às respostas no momento em que elas estão sendo feitas, e para fazer algumas ações nelas, como por exemplo, modificar o objeto, passar informações extras etc.

Nós conseguimos encadear esses mini programas, esses middlewares, para fazer as alterações nas requisições e respostas conforme necessário. No meu caso, estou utilizando um chamado express.json, que serve para executar esse express.json em todas as requisições manipuladas pelo Express.

Ou seja, qualquer requisição cujo corpo é um objeto compatível com JSON, como um objeto com id e título ou um array de objetos, passará por esse middleware e será convertido e analisado (ou 'parseado') para JSON.

Mas por que precisamos fazer essa conversão? Não estamos já trabalhando com objetos? Sim, estamos.

No entanto, toda vez que recebemos dados via corpo em uma requisição, eles chegam convertidos como string. Embora eles tenham o formato JSON, formato de objeto, com pares de chave-valor, eles viajam na conexão HTTP no formato string. Para conseguirmos utilizar os dados como JSON, ou seja, acessar as propriedades deles, precisamos converter essa string novamente para JSON.

Importante lembrar que toda requisição deve ter uma resposta. Então, abaixo do local onde fizemos o push do novo livro na nossa array livros, precisamos enviar uma resposta de volta para quem fez a requisição.

Portanto, teremos res.status. Agora, ao invés de 200, vou passar 201, que é o código de status HTTP para registro criado. Quando fazemos uma operação com sucesso em qualquer requisição, o código é 200. Quando criamos algo com sucesso, o código é 201.

E utilizarei também .send, porque vou passar uma string dizendo "Livro cadastrado com sucesso". Neste caso, não precisamos de JSON, podemos usar apenas .send.

app.**post**("/livros", (req, res) => {

livros.**push**(req.body);

res.**status**(201).**send**("Livro cadastrado com sucesso");

})

COPIAR CÓDIGO

Agora, não conseguimos mais fazer o teste no navegador, porque o navegador só realiza requisições GET, ele não realiza outros métodos, ou seja, não posso passar um POST via navegador.

Portanto, agora precisaremos do **Postman**. As instruções de como baixar e instalar estão no começo dessa aula, no "Preparando o ambiente".

Precisaremos de um JSON para fazer uma requisição do tipo POST. Então, vamos lá. Eu já deixei o Postman aberto no meu computador. Vamos criar uma nova requisição com o Postman.

À esquerda, temos um menu com os métodos HTTP que podem ser utilizados. Já conhecemos o GET, que o próprio navegador realiza para nós, mas também podemos fazer pelo Postman. Selecionaremos o segundo, que é o POST.

A URL a ser utilizada é a nossa localhost:3000/livros.

E precisamos enviar um corpo nessa requisição. Então, abaixo do endereço, temos um menu, o "body". Clicarei nele, selecionarei a opção RAW e no último menu, que começa com texto, selecionarei JSON, que é o tipo de dado que queremos enviar na nossa requisição.

Abaixo, temos uma área para escrevermos nosso objeto. Criarei um novo objeto livro aqui, que tem "id": 3 e o "título": "O Silmarillion", outro livro de J. R. R. Tolkien.

Ao trabalharmos com JSON, sempre utilizamos aspas duplas nas chaves, assim como nos conjuntos de chave-valor, e também para as strings.

"id": 3,

"titulo": "O Silmarillion"

COPIAR CÓDIGO

Vou voltar ao terminal para ver se não há nenhum erro, o servidor está de pé.

Agora, no Postman, podemos clicar no botão que está à direita, "Send", e ver se ele retorna uma resposta. O servidor retornou "Livro cadastrado com sucesso".

Mas como saber se o livro foi mesmo cadastrado com sucesso? No Postman, há um botão com o símbolo de soma ('+') para abrir uma nova aba, onde podemos criar uma nova requisição e confirmar se o cadastro ocorreu como esperado.

Podemos ir para /livros em localhost/livros e fazer uma requisição do tipo GET (que é a primeira opção de método). Lembrando que GET não tem body, então apenas necessitamos enviar essa requisição clicando no botão Send.

Agora o Postman já retornou o meu array atualizado, com id 3, título "O Silmarillion". Isso indica que, por enquanto, nossa API está funcionando.

Fizemos uma requisição do tipo POST para /livros, ela adicionou o livro no array de livros, utilizando os dados recebidos pela requisição e manipulados pelo Express.

Ou seja, Express acessou a requisição, acessou a propriedade body, pegou o que havia dentro de body, que é o nosso objeto livro - id 3, título "Silmarillion" - e adicionou com o push no array. Em seguida, retornou apenas com um Send a mensagem "Livro cadastrado com sucesso".

Entretanto, fizemos uma nova requisição GET e o array veio atualizado. Qualquer alteração que fizermos e salvarmos o arquivo, assim que o nodemon reiniciar o servidor, essas alterações vão desaparecer, porque esse array está em memória.

Mas, por enquanto nós vamos ficar com essa configuração. Mais tarde, conseguiremos persistir nossos dados com um banco de dados.

Por enquanto, temos todo o processo funcionando, tanto para obter livros, quanto para criar um novo livro. Se pensarmos um pouco no que normalmente fazemos com dados, nós criamos, recuperamos, deletamos e atualizamos. É isso que nós vamos fazer em seguida, já utilizando o Express e o nosso array de livros. Então, vamos continuar.

Observe o resultado a seguir, onde o array de livros é atualizado após a operação POST:

GET /livros

[

{

"id": 1,

"titulo": "O Senhor dos Anéis"

},

{

"id": 2,

"titulo": "O Hobbit"

},

{

"id": 3,

"titulo": "O Silmarillion"

}

]

COPIAR CÓDIGO

**Nota:** Lembre-se de que todas as alterações feitas serão perdidas assim que o servidor for reiniciado porque o array de livros é apenas armazenado na memória.

**Buscando e atualizando livros**

O próximo passo é criar uma nova rota para acessar apenas um livro, que é algo muito comum para quando quero localizar um registro específico.

Nesse sentido, vou criar uma nova rota. Já sabemos que é uma rota do tipo get, pois queremos obter apenas um livro. Portanto, abaixo do último app.get, vou criar um novo app.get. Este será um array, mas a rota que vamos acessar não será mais /livros.

Terá que ser um pouco diferente, pois já temos um get/livros. Portanto, terá que ser /livros, seguido por um parâmetro que permitirá a localização de um livro específico.

No caso de nossa API, temos o título, mas temos uma informação mais precisa: o ID. O ID é um localizador único, que utilizamos para qualquer registro que precise de um localizador sem duplicação. Portanto, o que podemos passar em nossa rota é get("/livros/") com o ID que queremos localizar.

Como posso indicar ao Express que esse dado será variável? Eu prefixo isso com dois pontos. Portanto, teremos /livros/:id. Com esses dois pontos, estou informando ao Express que o ID será uma informação variável, que será processada de acordo com o valor que for passado.

O segundo parâmetro continua o mesmo, é a nossa conhecida callback function, que recebe req e res (requisição e resposta). Abro uma arrow function, seguida por chaves.

app.**get**("/livros/:id", (req, res) => {

})

COPIAR CÓDIGO

E o que acontecerá aqui dentro desse GET? Lembre-se de que ainda estamos trabalhando com um array normal do JavaScript por enquanto, então continuamos usando os métodos do JavaScript. Portanto, o que o nosso GET livros fará? Ele precisa buscar, usando o JavaScript, um elemento nesse array na propriedade ID.

Para facilitar o processo, vou separar em uma função distinta que possamos chamar dentro do nosso app.get. Depois do array, vou criar uma função normal do JavaScript com a declaração function, eu vou nomear essa função de buscaLivro.

A função buscaLivro vai receber um ID, que é o que vai chegar via requisição, naquela variável que mencionei com dois pontos. Essa função vai procurar dentro do array livros, se existe um livro com esse ID e em qual índice ele está do array, para que possamos recupera-lo.

Portanto, ela vai retornar o resultado de livros.findIndex, que é um método do JavaScript para arrays que retornam o índice onde está o elemento desejado, baseado no que passamos a ele.

Então, findIndex é um método callback também, que vai receber cada livro do nosso array de livros e vai retornar, por sua vez, se livro.id for estritamente igual ao ID recebido no parâmetro.

No entanto, lembre-se de que eu comentei que os dados que trafegam via HTTP, também trafegam no formato string. E nossos IDs no array de livros, são number, ou seja, são números.

Para fazer essa comparação corretamente com os três iguais do JavaScript, primeiro precisamos converter esse ID para number. Então, Number(id), caso contrário, ele vai tentar comparar uma string com um número usando a comparação estrita e não vai dar certo.

**function** **buscaLivro**(id) {

**return** livros.**findIndex**(livro => {

**return** livro.id === **Number**(id);

})

}

COPIAR CÓDIGO

Portanto, após a conversão, ele vai retornar o índice onde está esse livro dentro do array. Já que criamos a nossa função, podemos chamar essa função dentro da nossa requisição, dentro do app.get.

Voltando ao app.get, vamos criar uma const index para buscar esse índice e salvar na variável. Ele será o resultado de buscaLivro, que é a nossa função, recebendo o quê? req.params.id.

app.**get**("/livros/:id", (req, res) => {

**const** index = buscaLivro(req.**params**.id);

})

COPIAR CÓDIGO

Params é outra propriedade que vem dentro do nosso objeto requisição, montado pelo Express. 'Params', por quê? Porque é um **parâmetro**. Estamos passando ID como um parâmetro da rota e usamos os dois pontos para indicá-lo como um parâmetro variável. O ID é o nome que nós atribuímos a esse parâmetro.

Então, se nós identificássemos com livros/:id, qualquer outro nome ou identificador, ele seria recebido como essa propriedade do objeto dentro de params. Para esclarecer req.params - porque pode haver mais de um - req.params.id vai puxar a requisição, passar para dentro de buscaLivro e, então, buscaLivro vai retornar algo e guardar esse número na constante index.

A partir disso, podemos acessar o nosso array e passar isso adiante. Então, agora podemos criar a nossa resposta. res.status vai ser 200 (de OK), res.status(200).json e o resultado será o array livros[index], porque index é a posição que foi localizada pela função.

Vamos fazer um teste para esclarecer: Primeiro, vou verificar no terminal se o servidor está funcionando, sem nenhum erro. Com tudo em ordem, podemos voltar ao Postman e acessar a aba da requisição GET.

Agora, o nosso id 3, criado anteriormente, não existe mais porque o servidor já foi reiniciado. Porém, id 1 e id 2 permanecem, pois estão fixados no array.

Lembrando que a nossa rota é /livros/, e precisamos passar algum índice. Vou passar 1, que é o ID para o título "O senhor dos anéis", e, ao enviar, recebemos apenas o objeto com o id 1.

{

"id": 1,

"titulo": "O Senhor dos Anéis"

}

COPIAR CÓDIGO

Ou seja, a requisição recebeu esse ID como parâmetro, ele foi passado para dentro da função buscaLivro, a função buscaLivro localizou a posição no array onde livro.id é igual a 1, retornou esse índice e, provavelmente, nesse momento, temos um livro no índice 0, que é o primeiro do array. Este é retornado para nós no formato JSON na resposta da requisição, tudo certinho.

Vamos aproveitar que estamos aqui e fazer, com mais agilidade agora, um método para alterar o nome de um livro, que é o método put. Utilizamos get para obter, post para criar e put ou patch para alterar um registro que já existe. Deixarei material extra sobre esses dois métodos para você conferir as diferenças entre eles.

Então, app.put, similarmente, vai ser na rota /livros/, mas aqui, novamente, temos que passar qual é o livro que queremos alterar. Também devemos passar o parâmetro para localizar qual é o livro que será alterado.

Segundo parâmetro permanece req, res, abre arrow function, abre chaves e, porque estamos trabalhando com o array de JavaScript, começamos criando uma busca-livro. Podemos até copiar de app.get, nosso const index = buscaLivro, porque da mesma forma que tivemos que buscar um livro para retornar no GET, temos que buscar um livro para alterar.

Iniciamos buscando esse livro na nossa base, encontramos o livro na nossa base e, agora, simplesmente utilizamos um método JavaScript padrão para alterar o objeto. Então, livros[index], por exemplo, não podemos alterar o ID, por padrão. O ID é um identificador único, mas podemos alterar o título. Então, podemos passar livros[index].titulo que será recebido e substituí-lo por req.body.titulo.

Lembrando que esta linha está apenas usando JavaScript padrão para encontrar uma propriedade qualquer de um objeto e substituir o valor dela.

Então, o valor atual está sendo substituído, no índice que está sendo passado, pelo valor de título que vamos receber no corpo da requisição, sem esquecer de enviar a resposta. Portanto, res.status, passaremos, aqui, (200).json. Vou simplificar esse exemplo, e pedir para retornar o array inteiro novamente, para facilitar o retorno nesse teste. Tudo certo?

Vamos fazer mais um teste no Postman. No terminal, está tudo certo, sem nenhum erro. No Postman, podemos criar uma nova requisição. Vou copiar a URL que usamos no GET, clicar no ícone de soma ("+") para criar uma nova requisição do tipo put, que sempre precisa corresponder ao tipo que estamos passando no Express.

Irei fazer uma alteração, por exemplo, no livro, continuando a editar o ID1. O put, por necessitar o envio de dados, deve possuir um corpo. Portanto, no menu de corpo do Postman, seleciono raw, escolho o tipo JSON e aqui passo apenas o título. Logo, crio aqui um objeto, passo título, entre aspas duplas, e aqui vou dizer que errei o título. O título estava errado, o Senhor dos Anéis, só o primeiro episódio. Então, "O Senhor dos Anéis, a Sociedade do Anel".

Pronto. Em seguida, vamos testar nossa requisição com send, put, livros, barra 1, e ele retornou o array novamente, como havíamos solicitado que retornasse o array inteiro como resposta, e ele alterou o título do ID1, "O Senhor dos Anéis, a Sociedade do Anel".

O que fizemos aqui foi praticar como utilizamos o Express para vários métodos, os métodos do HTTP para várias operações que precisamos ser capazes de realizar em uma API. Buscar vários dados, buscar um dado, criar um dado novo, alterar um dado. Só falta conseguirmos excluir um dado, o que faremos em seguida.

E, além do corpo, passamos dados, enviamos dados de uma requisição via corpo da requisição, também conseguimos passar certos tipos de dados específicos de outras maneiras, por exemplo, via param, parâmetro da rota, que utilizamos aqui, assinalando com dois pontos.

Em seguida, o Express pegou esse valor, alterou na rota, e assim conseguiu processar a requisição conforme precisávamos, com o número do ID, com o dado correto.

Portanto, só falta o delete, vamos finalizar aqui nosso primeiro teste com rotas do Express.

**Deletando livros**

Já que utilizamos app.put, app.post e app.get para as operações anteriores, para realizar a exclusão, utilizaremos app.delete.

Os métodos delete, post e put são provenientes do *Express*. Os parâmetros para o app.delete são exatamente os mesmos das outras operações. O primeiro parâmetro será uma *string* com a rota que será acessada com o método delete. Será algo como "/livros/:id", onde id é o identificador único do livro que desejamos excluir.

O segundo parâmetro é uma função que recebe req e res (requisição e resposta, respectivamente). Como estamos ainda utilizando um *array* do *JavaScript*, continuaremos a usar os métodos nativos para manusear esse *array*.

app.**delete**("/livros/:id", (req, res) => {

// código

});

COPIAR CÓDIGO

Exemplo de como a função poderia ser implementada:

app.**delete**('/livros/:id', (req, res) => {

**const** index = **buscaLivro**(req.params.id);

livros.**splice**(index, 1);

res.**status**(200).**send**('Livro removido com sucesso.');

});

COPIAR CÓDIGO

Como você pode observar acima, utilizamos a mesma função buscaLivro que utilizamos nas operações anteriores para localizar o índice do livro pelo id no *array*. Copiaremos da função app.put a linha que contém a chamada da função buscaLivro(req.params.id) e a inseriremos dentro do app.delete.

Uma vez que temos o índice do livro, podemos deletá-lo utilizando o método nativo do *JavaScript* splice, que consegue localizar um elemento em qualquer parte do *array* e deletá-lo.

Como queremos só deletar o registro, passaremos dois parâmetros para o splice: o primeiro será o index e depois 1, pois queremos deletar apenas um elemento do *array*.

app.**delete**('/livros/:id', (req, res) => {

**const** index = **buscaLivro**(req.params.id);

livros.**splice**(index, 1);

})

COPIAR CÓDIGO

No final da operação, precisamos enviar uma resposta, que pode ser um status 200 (OK) ou 204 (Sem conteúdo). Optei por utilizar o status 200 e mandar uma mensagem informando que o livro foi removido com sucesso. A resposta será então res.status(200).

Em seguida, usaremos .send em vez de .json, pois não tem motivo para retornarmos algum objeto nesse caso. Dentro de .send passaremos a mensagem "livro removido com sucesso".

Como sempre, após realizar mudanças no código, é importante testar se elas estão funcionando conforme o esperado. Portanto, primeiro verificaremos se o servidor está funcionando corretamente, e em seguida utilizaremos o *Postman* para testar a operação que acabamos de criar.

No *Postman*, faremos um GET para buscar todos os livros e verificar se o *array* é retornado corretamente. Em seguida, realizaremos um POST para criar um novo livro, e por fim, realizaremos um DELETE para excluir um dos livros.

O método POST é usado para criar um novo livro. Durante este processo, nós utilizamos o corpo da requisição, formatado em JSON, para que o Postman envie um novo registro de livro. Ao clicar em "Send", um novo teste será realizado.

Quando o Postman retorna, ele também mostra o status code utilizado, neste caso foi o 201 ("Created"), usado quando um novo registro é criado. Com isso, o livro foi cadastrado com sucesso.

Em seguida, implementamos o método GET para obter um livro. No exemplo, vou solicitar o livro 3, que acabamos de criar. Então, utilizando o get, combinado com a URL "[http://localhost:3000/livros/3](http://localhost:3000/livros/3" \t "_blank)", e ao apertar "Send", o Postman retorna apenas o objeto livro criado.

Agora podemos testar rapidamente os dois últimos métodos, PUT e DELETE. Vou solicitar a alteração do livro 1, que já havíamos criado anteriormente, alterando a URL para "[http://localhost:3000/livros/1](http://localhost:3000/livros/1" \t "_blank)".

No caso de alteração, precisamos enviar, via corpo da requisição, os novos dados que desejamos substituir. Mantenho as informações anteriores e apenas altero o título. Ao clicar em "Send", o título é modificado e retorna o array completo, conforme solicitado quando criamos o método PUT.

Por último, vamos testar o DELETE. Para isso, vou copiar a URL "[http://localhost:3000/livros/1](http://localhost:3000/livros/1" \t "_blank)" para utilizá-la no método DELETE. Após selecionar o método delete e colar a URL, vamos solicitar a exclusão do livro 1, que é "O Senhor dos Anéis".

Nesse processo, não é necessário enviar nada no corpo da requisição. Ao clicar em "Send", o retorno é "Livro removido com sucesso". Agora, se retornarmos à requisição GET e fizermos uma nova solicitação apenas para /livros, o livro de id 1, "O Senhor dos Anéis", deverá ter sido removido do array. Após o teste, o resultado é o seguinte:

{

"id": 2,

"titulo": "O Hobbit"

},

{

"id": 3,

"titulo": "O Silmarillion"

}

COPIAR CÓDIGO

O teste foi bem-sucedido, agora no array só contém o livro de id 2 e o livro de id 3.

Portanto, todo o nosso processo foi bem-sucedido e todas as operações que podemos realizar em um registro foram concluídas. O que fizemos nesta aula é conhecido como **CRUD**, que é um acrônimo que representa as operações básicas que são realizadas em um banco ou com um registro, que são:

* Criar (*Create*)
* Ler (*Read*)
* Alterar (*Update*)
* Deletar (*Delete*)

O "C" de criar é realizado pelo POST, o "R" de *read* (ler) é feito pelo GET, o "U" de *update* (atualizar) é feito pelo PUT e o "D" de *delete* (deletar) é feito pelo DELETE. Portanto, o CRUD é uma das bases do back-end, que engloba basicamente tudo o que fazemos com um registro.

Se você pensar bem, quando interage com uma página da internet, é, em essência, o que nós fazemos. Nós obtemos registros, criamos coisas, alteramos o que criamos e deletamos o que também criamos.

Você pode encontrar mais informações sobre o CRUD nas atividades extras. Agora nós vamos continuar evoluindo a nossa API e utilizando mais métodos do Express, agora com um banco de dados, para conseguirmos salvar o que estamos criando.

Vamos em frente!

**Para saber mais: o CRUD**

**CRUD** (Create, Read, Update, Delete) é um acrônimo para as operações mais comuns feitas em qualquer base de dados: **Criar, ler, atualizar e excluir**.

No dia a dia, quando falamos em CRUD normalmente também nos referimos a aplicações, APIs ou serviços responsáveis por estas operações. Além da lógica básica de GET, POST, PUT/PATCH e DELETE, as aplicações normalmente implementam um conjunto de outras funcionalidades necessárias ao CRUD, como métodos de validação, autenticação, paginação, testes, tratamento de erros etc.

Muitos serviços e aplicações complexas têm por trás algum tipo de operação de CRUD, o que faz parecer uma tarefa “simples”. Na verdade, o termo CRUD descreve apenas os tipos de operação, por outro lado a implementação pode ser muito complexa! Cada produto tem sua especificidade, além das escolhas de arquitetura e de ferramentas, ou seja, cada implementação tem um número enorme de variáveis e sempre há coisas novas a aprender e novos problemas para resolver.

Nossa implementação de CRUD é o primeiro passo, a partir daqui já vai ser possível expandir para diversas outras funcionalidades. Após finalizar o projeto deste curso, você pode continuar explorando esta API com o [próximo curso da formação](https://cursos.alura.com.br/course/node-js-buscas-filtros-paginacao-erros-api).

**Para saber mais: métodos HTTP**

Durante a aula, trabalhamos com os métodos HTTP mais conhecidos: GET, POST, PUT e DELETE. É o chamado **CRUD** (Create, Read, Update e Delete; ou criar, ler, atualizar e apagar, em português).

Além desses quatro métodos, existem outros que também são utilizados com relativa frequência, de acordo com a necessidade da API.

Você pode conferir a [lista de métodos e uma breve descrição do uso de cada um deles na documentação do MDN](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods).

## PUT vs PATCH

No projeto foi utilizado o método PUT para fazer alterações em um registro (o Update do CRUD). Porém, existe o método PATCH que também faz atualizações em um registro. Então qual é a diferença entre eles?

* **PUT** substitui totalmente o recurso atual pelos novos dados que estão sendo recebidos na requisição. Caso não exista o recurso anterior, ele será criado. Apesar disso, não é o método indicado para a criação de novos recursos, para isso existe o método POST.
* **PATCH** atualiza parcialmente um recurso já existente. Ao contrário do PUT que precisa receber um recurso completo para fazer a substituição completa, PATCH pode receber apenas os dados a serem modificados para atualizar apenas estes campos.

### Quando utilizar um ou outro?

* PUT pode ser usado em situações em que há acesso ao recurso completo (por exemplo, todos os campos do documento a ser atualizado) ou a necessidade de substituir totalmente o recurso. É necessário enviar sempre o recurso completo (com todos os campos);
* PATCH pode ser usado para atualizações parciais e pode receber apenas o campo que será atualizado, o que pode significar menor volume de tráfego de dados.

Quando estiver consumindo (ou seja, utilizando) APIs em seu projeto, sempre consulte a documentação para saber quais são os métodos permitidos, pois a escolha entre PUT e PATCH é feita pelo time que desenvolve a API, assim como estamos fazendo durante o curso.

### Por que utilizamos PUT em nosso projeto?

A princípio é possível utilizar tanto PUT quanto PATCH para fazer atualizações no formato em que fazemos em nosso projeto, pois não estamos trabalhando com quantidades significativas de dados no payload (“pacote” de todos os dados enviados na requisição).

Como a API utiliza um método “pronto” do Mongoose para fazer a atualização, o findByIdAndUpdate, o método já lida internamente com a questão de resgatar o registro completo e também de fazer a atualização. Assim, neste caso, é possível utilizar PUT e enviar no corpo da requisição apenas os dados de substituição.

Ainda há duas diferenças: PUT é considerado um método idempotente e sem efeitos colaterais, ao contrário de PATCH, e por isso um método “seguro”. Você pode conferir mais sobre [o conceito de idempotente, que não vamos abordar neste curso, na documentação do MDN](https://developer.mozilla.org/pt-BR/docs/Glossary/Idempotent).

**Os métodos HTTP do CRUD**

Durante esta aula, utilizamos os métodos HTTP para criar, buscar, alterar e excluir livros, o famoso CRUD.

Baseado no que estudamos, analise os códigos e as afirmações para assinalar a opção correta:

Top of Form

* Alternativa correta

O código abaixo fará a inclusão de um novo livro apenas com a informação de título, utilizando para isso os dados de título recebidos na requisição via req.body.titulo.

app.**put**("/livros/:id", (req, res) => {

**const** index = **buscaLivro**(req.params.id);

livros[index].titulo = req.body.titulo;

res.**status**(200).**json**(livros);

});

* Alternativa correta

Observando o código abaixo:

app.**get**("/livros", (req, res) => {

res.**status**(200).**json**(livros);

});

app.**post**("/livros", (req, res) => {

livros.**push**(req.body);

res.**status**(201).**send**("livro cadastrado com sucesso");

});

É possível concluir que é correto os métodos diferentes GET e POST possam acessar exatamente a mesma rota /livros, pois cada método resultará em uma resposta diferente.

O uso da mesma rota para todas as operações referentes a uma mesma entidade (exemplo livro e suas subrotas) faz parte do padrão REST. O método utilizado (GET ou POST) faz parte da estrutura da requisição e é utilizado pela API para o processamento correto da requisição e envio da resposta de acordo.

* Alternativa correta

Para criarmos um novo livro, devemos fazer uma requisição do tipo PUT para a rota /livros conforme o código abaixo:

app.**put**("/livros", (req, res) => {

livros.**push**(req.body);

res.**status**(201).**send**("livro cadastrado com sucesso");

});

Bottom of Form

**Faça como eu fiz: revisando as rotas de livros**

Entre as funcionalidades que adicionamos nesta aula, está a busca de livros via ID, criação de novos livros e atualização de livros. Vamos revisar o processo ponto por ponto:

**No arquivo: src/app.js**

* Adicione uma nova função chamada buscaLivro com um parâmetro chamado id;
* Dentro da função buscaLivro, utilize o método findIndex para encontrar o índice do livro que possui o mesmo ID passado como parâmetro;
* Retorne o resultado encontrado pelo método findIndex dentro da função buscaLivro.

**function** **buscaLivro**(id) {

**return** livros.**findIndex**(livro => {

**return** livro.id === **Number**(id);

})

}

COPIAR CÓDIGO

* Adicione uma nova rota com o método GET e o caminho /livros/:id;
* Dentro da função de callback da rota /livros/:id, chame a função buscaLivro passando o valor do parâmetro id como argumento;
* Envie uma resposta com o status 200 e o conteúdo em formato JSON do livro encontrado no array livros usando o índice retornado pela função buscaLivro.

app.**get**("/livros/:id", (req, res) => {

**const** index = **buscaLivro**(req.params.id);

res.**status**(200).**json**(livros[index]);

});

COPIAR CÓDIGO

* Adicione uma nova rota com o método POST e o caminho /livros;
* Dentro da função de callback da rota /livros, adicione o corpo da requisição (livro) ao array livros com o método push;
* Envie uma resposta com o status 201 e o conteúdo "livro cadastrado com sucesso".

app.**post**("/livros", (req, res) => {

livros.**push**(req.body);

res.**status**(201).**send**("livro cadastrado com sucesso");

});

COPIAR CÓDIGO

* Adicione uma nova rota com o método PUT e o caminho /livros/:id;
* Dentro da função de callback da rota /livros/:id, chame a função buscaLivro passando o valor do parâmetro id como argumento;
* Atualize o título do livro encontrado usando o corpo da requisição (req.body.titulo);
* Envie uma resposta com o status 200 e o conteúdo em formato JSON do array livros atualizado.

app.**put**("/livros/:id", (req, res) => {

**const** index = **buscaLivro**(req.params.id);

livros[index].titulo = req.body.titulo;

res.**status**(200).**json**(livros);

})

**Para saber mais: mais sobre NoSQL**

Quer saber ainda mais sobre alguns tipos de bancos NoSQL, suas características e em quais cenários podem ser melhor utilizados, confira o Alura+ sobre o assunto!

**Criando coleção no MongoDB**

Entre os materiais deste curso, há algumas instruções iniciais para a criação de sua conta no [site do MongoDB](https://www.mongodb.com/pt-br). Agora, vamos criar uma instância gratuita.

## Criando uma Instância do MongoDB

Vamos ao navegador, no qual devemos estar conectados à conta, na qual criamos uma organização chamada Alura. Contudo, você pode nomeá-la como preferir.

Vamos acessar essa organização. Na página "Projects", daremos início ao primeiro projeto ao clicar em "New Project", botão localizado à direita do título da página. Recomendamos que você acompanhe o processo.

Depois do vídeo, haverá uma atividade explicativa em texto para ser consultada, caso tenha alguma dificuldade, ou caso a interface do Atlas tenha sido modificada após a gravação do curso.

Ainda assim, é importante acompanhar o vídeo, pois o procedimento deverá ser igual.

Ao clicar em "New Project", veremos a página "Create a Project" com duas guias: "Name your Project" (Nomeie seu projeto) e "Add members" (Adicione membros). Na primeira, haverá um campo de texto para o nome do projeto. Vamos nomeá-lo de "Livraria" e, em seguida, clicaremos em "Next" (Próximo), o que nos levará para a segunda guia.

No que diz respeito à adição de membros, não será necessário adicionar ninguém para o contexto deste projeto, pois trata-se de um exemplo para nosso aprendizado. Portanto, apenas vamos prosseguir clicando no botão "Create Project" (Criar Projeto), no canto inferior direito da página.

Após criado o projeto, será direcionada para a tela "Database Deployments" (Implantação da Base de dados) para criar uma nova base de dados. Vamos clicar no botão "Build a Database", no centro da página.

Seremos direcionados à página "Deploy your database" (Implante sua base de dados), na qual haverá uma seção com algumas opções de custo apresentadas. Vamos escolher a última opção à esquerda: "Free", que é gratuita.

Na seção "Provider" (Provedor), abaixo da anterior, manteremos o Provider, como AWS, que já estará selecionado. Não é necessário realizar nenhuma alteração. No campo "Name", é possível nomear o cluster (grupo de dados), mas manteremos o nome padrão, que é "Cluster0".

Manteremos o campo da seção "Tag" vazio, sem adicionar nenhuma tag. No final da página haverá a seção "Free", com o botão "Create". Clicaremos nele para seguir adiante e sermos direcionados para a página "Security Quickstart" (Inicialização Rápida de Segurança".

Nesta página, forneceremos os dados de acesso a esse banco de dados. Existe mais de uma forma de fazer isso.

Optaremos pelo acesso mais simples, que é somente com Username (nome de pessoa usuária) e Password (senha). Para isso, na seção 1, clicaremos na opção "Username and Password" à esquerda da opção "Certificate" (certificado).

No campo Username, escreveremos "admin" e em Password, escreveremos "admin123", para lembrarmos facilmente. Este banco será usado apenas localmente e, provavelmente, quando você estiver assistindo a este curso, ele já terá sido excluído.

Por fim, clicaremos no botão verde "Create User" para criar o login com esses dados e desceremos para a seção 2, que inicialmente nos indaga por onde queremos nos conectar.

A conexão ao banco de dados será feita através do meu ambiente local. Portanto, manteremos a seleção da opção "My Local Environment" (meu ambiente local), à esquerda de "Cloud Environment" (ambiente em nuvem).

A última parte de configuração na seção 2 consiste em selecionar os IPs que terão acesso ao banco. Será adicionado automaticamente o IP "177.8.171.44/32" na lista, contudo, adicionaremos outro, escrevendo "0.0.0.0/0" e clicando no botão "Add Entry" (adicionar entrada).

Este número de IP permite o acesso ao banco de qualquer lugar. Vamos manter esse acesso livre, por enquanto. Este não é um procedimento recomendado para produção, mas é comum em bases de dados para estudos, e serve para facilitar a configuração do banco. Posteriormente,você pode deletar o seu banco.

Após a adição deste IP, clicaremos no botão "Finish and Closed" (Finalizar e Fechar), no canto inferior direito da página.

Após o clique, seremos parabenizados pela criação do primeiro banco de dados em uma janela modal, na qual clicaremos em "Go to Database" (Ir para a Base de dados) para voltar à página "Database Deployments", onde deverá aparecer o "Cluster0".

O que criamos no processo anterior consiste em um banco de dados do MongoDB, hospedado em um serviço em nuvem fornecido gratuitamente pela própria empresa. Precisamos agora conectar esse banco à API.

## Conectando o Banco à API

À direita do nome "Cluster0", vamos clicar no botão "Connect" (Conectar). Ele abrirá uma janela modal com algumas opções para conectar a API ao banco de dados que acabamos de criar na nuvem.

Clicaremos na primeira opção, chamada "Drivers", para fazer a conexão por meio das bibliotecas do MongoDB em nosso projeto e através da connection string (string de conexão).

Após o clique, iremos para a etapa "Connecting with MongoDB Driver" (Conectando-se com o Driver do MongoDB). A primeira coisa que vamos fazer é selecionar o driver e a versão nos campos de lista selecionáveis "Driver" e "Version". Nesse caso, vamos usar o Node.js na versão 5.5 ou posterior, que é a mais recente, portanto, selecionaremos "Node.js" no primeiro campo e "5.5 or later" no segundo.

Agora, vamos instalar o Driver do MongoDB. Basicamente instalaremos a biblioteca ou dependência do MongoDB em nosso projeto com o conhecido comando npm install. Para isso, copiaremos a linha que a plataforma fornece na seção 2, "Install your driver" (Instale seu driver):

npm install mongodb

COPIAR CÓDIGO

Acessando novamente o terminal do computador, faremos um "Ctrl+C" para derrubar o servidor local. Lembrando que ao fazer instalações, precisamos necessariamente derrubar o servidor e reativá-lo.

Vamos colar e rodar no terminal o comando npm install mongodb para instalar. Após a conclusão desse processo, voltaremos à janela modal da plataforma do MongoDB, onde já completamos a etapa 2 de instalar a dependência do MongoDB em nosso projeto.

A última etapa, a 3, envolve a connection string (string de conexão) que o Atlas fornece. Uma connection string é basicamente o endereço que usaremos para conectar nossa base de dados à aplicação. Esse tipo de abordagem é muito comum para conexões de bancos de dados, não apenas para o Mongo.

Vamos copiar a linha que o Mongo Atlas está fornecendo na seção 3 e fechar a janela modal pressionando o botão "Close" (Fechar), no canto inferior direito.

mongodb+srv://admin: <password>@cluster0.uvmwiwx.mongodb.net/?retryWrites=true&w=majority

COPIAR CÓDIGO

Vamos voltar ao projeto pelo Visual Studio Code. Podemos colar esta connection string no arquivo app.js, abaixo do export default app. Vamos comentá-la, apenas para salvá-la e podermos usar mais tarde.

// Código omitido

**export** **default** app

// mongodb+srv://admin: <password>@cluster0.uvmwiwx.mongodb.net/?retryWrites=true&w=majority

COPIAR CÓDIGO

Por enquanto, não iremos à API. Voltando à página do MongoDB no navegador, vamos criar pelo próprio Mongo Atlas um primeiro registro, para termos acesso ao nosso banco com pelo menos um documento e seu registro.

Nossa base de dados "Livraria" existe, mas ainda não contém nenhuma ***coleção***. O que é uma coleção?

Um banco MongoDB é composto por coleções de documentos. Em uma livraria, "Livros" é uma coleção de livros, "Autores" é uma coleção de autores. Vamos criar a nossa primeira coleção.

Para isso, vamos clicar na opção "Browse Collections" (navegar por coleções), à direita do nome "Cluster0" e dos botões "Connect" e "View Monitoring". Essa ação nos redirecionará para a página do "Cluster0" e acessará a guia "Collections".

Ainda não temos nenhuma coleção, mas se descermos a página da guia "Collections", temos um botão que diz "Add My Own Data" (Adicionar meus próprios dados). Vamos clicar nele para abrir a janela modal "Create Database" (Criar Base de dados), que possui três campos digitáveis: "Database name" (Nome da Base de dados), "Collection name" (Nome da coleção) e "Additional Preferences" (Preferências adicionais).

O nome da base de dados será "livraria" e o nome da coleção, por se tratar de uma coleção de livros, será "livros". Não necessitamos de nenhuma preferência adicional, portanto manteremos esse campo vazio. Por fim, clicaremos em "Create", no canto inferior direito do modal.

Aguardaremos um momento para carregar a guia "Collections". Após esse processo, vemos agora a nossa coleção "livraria.livros". Se trata de um nome de objeto, portanto, vemos a estrutura de objetos, separando o banco e a coleção por um ponto.

Ainda não temos nenhum documento, portanto, clicaremos no botão "Insert Document" (Inserir Documento), no canto direito da página. O clique abrirá a janela modal "Insert Document", na qual veremos uma estrutura de objeto.

| **#** | **#** | **#** |
| --- | --- | --- |
| 1 | \_id: 64c2a45ad5a0d7f958add9e0 | ObjectId |
| 2 | : " " | String |

Por meio dessa janela, o MongoDB nos permite criar e adicionar um livro na nossa base. Como se trata de um livro, começaremos adicionando um titulo do tipo String na linha 2, à esquerda dos dois pontos. Esse título será "O Senhor dos Anéis".

| **#** | **#** | **#** |
| --- | --- | --- |
| 1 | \_id: 64c2a45ad5a0d7f958add9e0 | ObjectId |
| 2 | titulo: "O Senhor dos Anéis " | String |

Além do título, um livro pode ter um autor, mas não vamos adicionar neste momento. Vamos clicar no botão "+" abaixo do número das linhas para criar a linha 3, na qual vamos dizer que esse livro tem uma editora, que chamaremos de "Editora Clássicos", apenas para não mencionar um nome real.

Criaremos a linha 5, na qual definiremos um preço de 10reais, para que o livro seja bem acessível.

Por fim, podemos também definir o número de páginas. Criaremos mais uma linha com o campo paginas, que receberá o valor de 200 páginas.

Vamos clicar nos campos de tipos correspondentes aos campos de preço e de páginas, inicialmente marcados como String, e alterá-los para Int32, que representa um tipo numérico. Isso permite que possamos fazer operações com esses números.

| **#** | **#** | **#** |
| --- | --- | --- |
| 1 | \_id: 64c2a45ad5a0d7f958add9e0 | ObjectId |
| 2 | titulo: "O Senhor dos Anéis " | String |
| 3 | editora: "Classicos" | String |
| 4 | preco: 10 | Int32 |
| 5 | paginas: 200 | Int32 |

Portanto, teremos duas informações em formato de String, sendo elas Título e Editora, e outras duas em formato numérico, Preço e Páginas.

Clicaremos no botão "Insert", no canto inferior direito do modal, e um novo livro será criado em nossa base de dados. Veremos na seção da coleção "livraria.livros" o seguinte documento:

\_id: ObjectId('64c2a45ad5a0d7f958add9e0')

titulo: "O Senhor dos Anéis"

editora: "Classicos"

preco: 10

paginas: 200

COPIAR CÓDIGO

Ao criar um novo livro, é gerado automaticamente um documento correspondente dentro da coleção de livros. Cada documento consiste em um livro único com um ID próprio, um título, uma editora, preço e número de páginas.

Criamos um banco de dados do MongoDB na nuvem. Já inserimos até mesmo um registro, de modo que, ao conectarmos o banco à nossa API, já devemos conseguir recuperar esse registro do banco de dados e fazê-lo aparecer no Postman.

Agora, vamos aprender como conectar as duas partes.

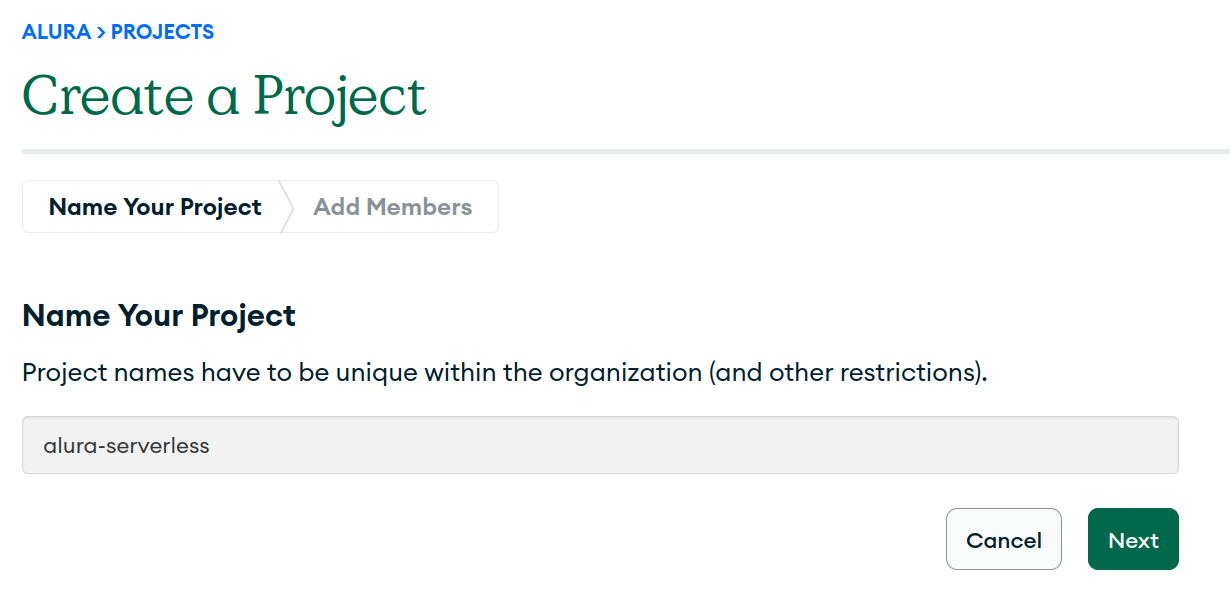
**Para saber mais: criação do banco no Atlas**

Para esta aula, vamos precisar de uma instância de banco de dados para armazenar nossos dados.

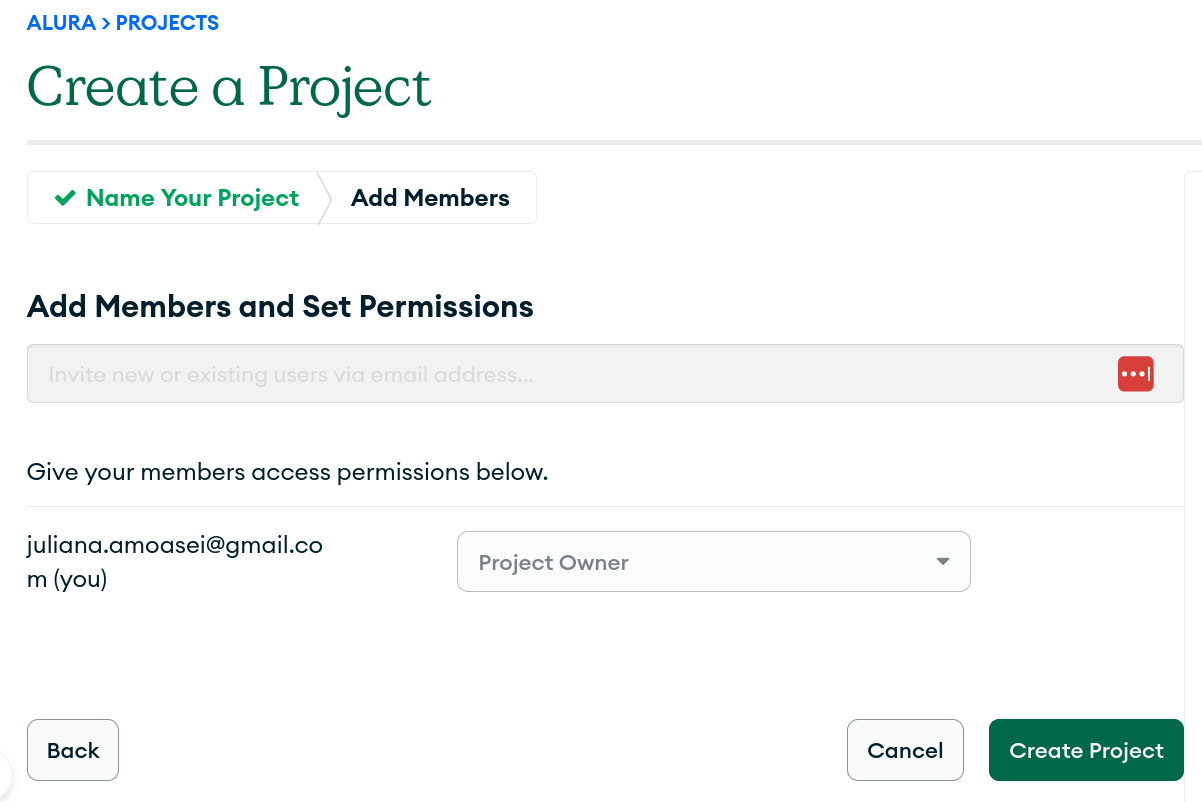
Siga os passos para a criação de um banco de dados utilizando o serviço Mongo Atlas:

1 - Acesse o [site Mongo Atlas](https://www.mongodb.com/atlas) e crie uma conta, caso não possua, ou faça login. Após logar, crie uma nova organização caso seja seu primeiro acesso, com o nome que desejar (por exemplo, alura).

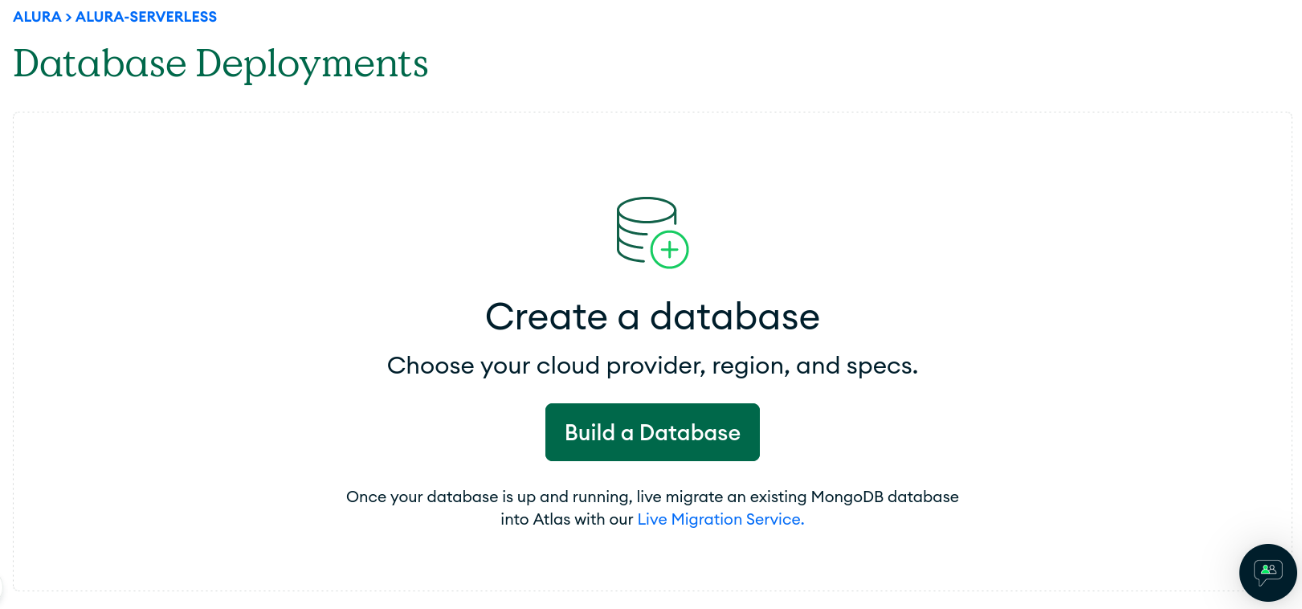
2 - Clique no botão **New Project** à direita da tela e dê um nome ao seu novo projeto. Clique em **Next**.



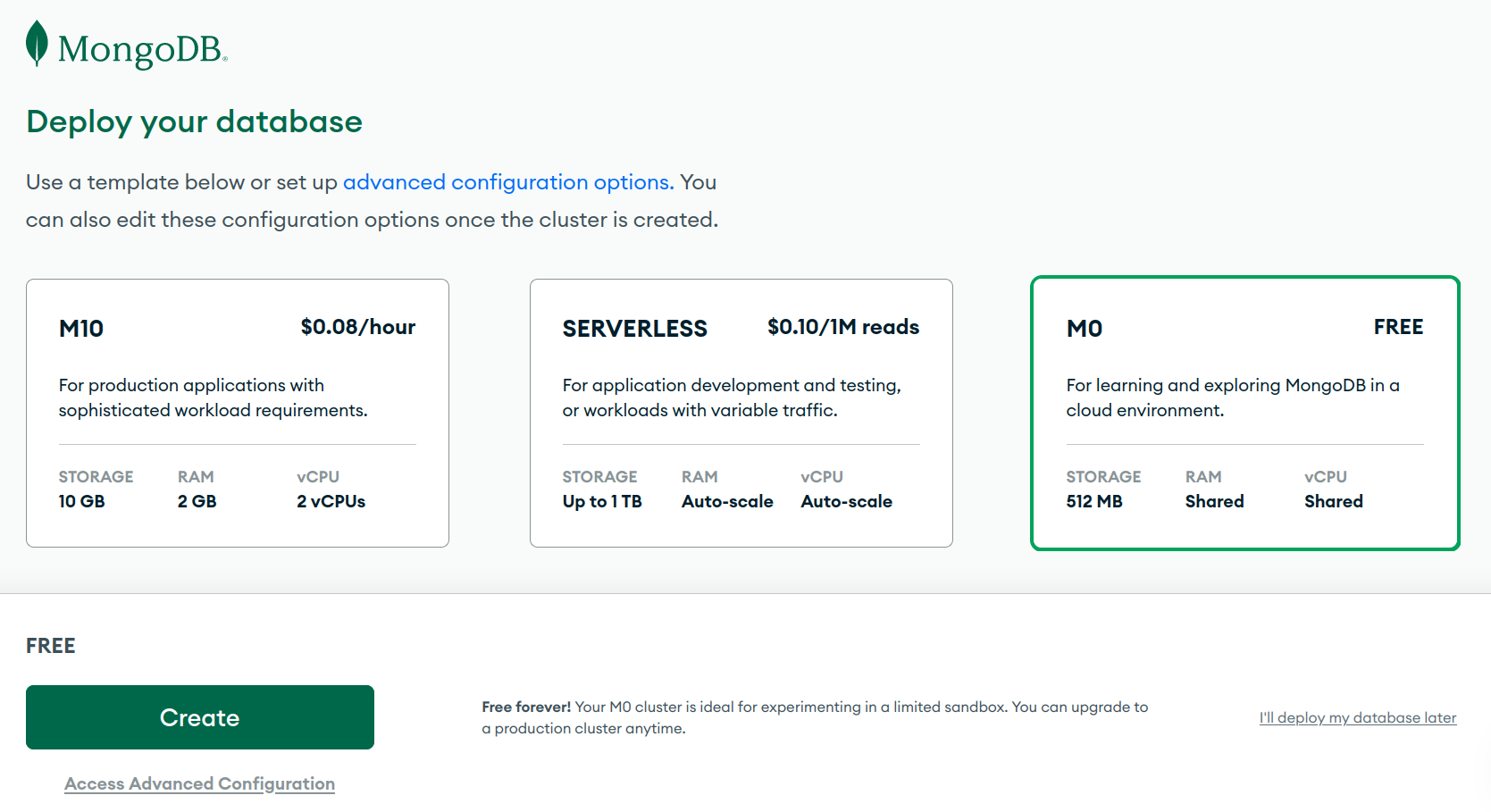
3 - O próximo passo é definir permissões. Você pode clicar em **Create Project** sem alterar nada.

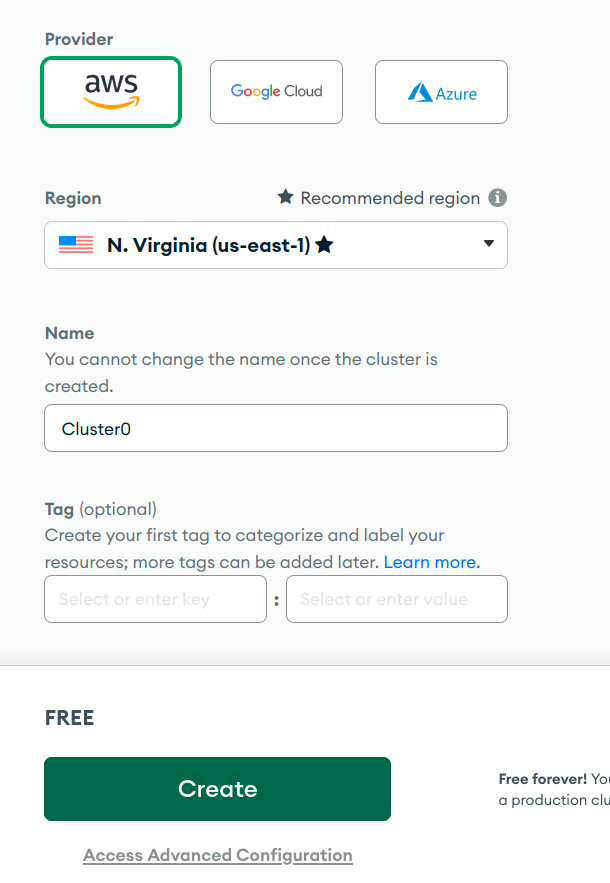


4 - Na tela **Create a database** clique no botão **Build a database** para criar uma nova instância de banco de dados.

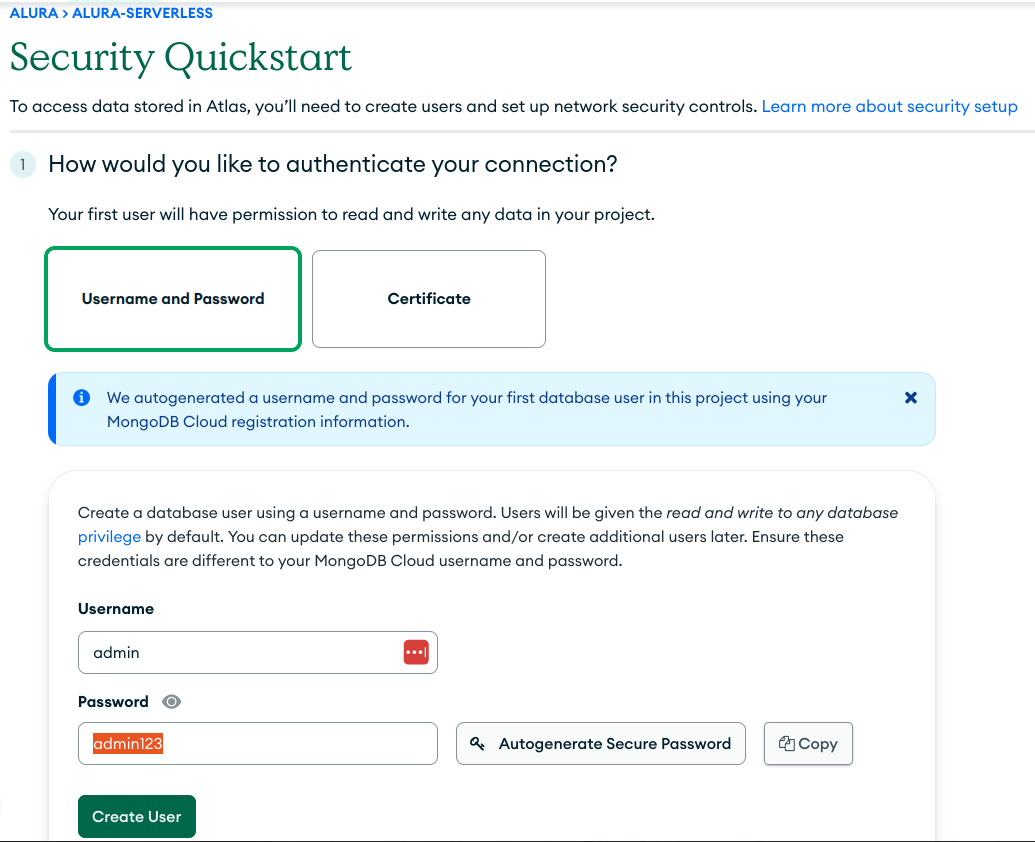


5 - Escolha a opção **M0 FREE**, role a tela e selecione a opção de provider **AWS**, região **us-east-1**. Em seguida, você pode dar um nome ao cluster ou manter a opção **Cluster0**. Clique em **Create**.



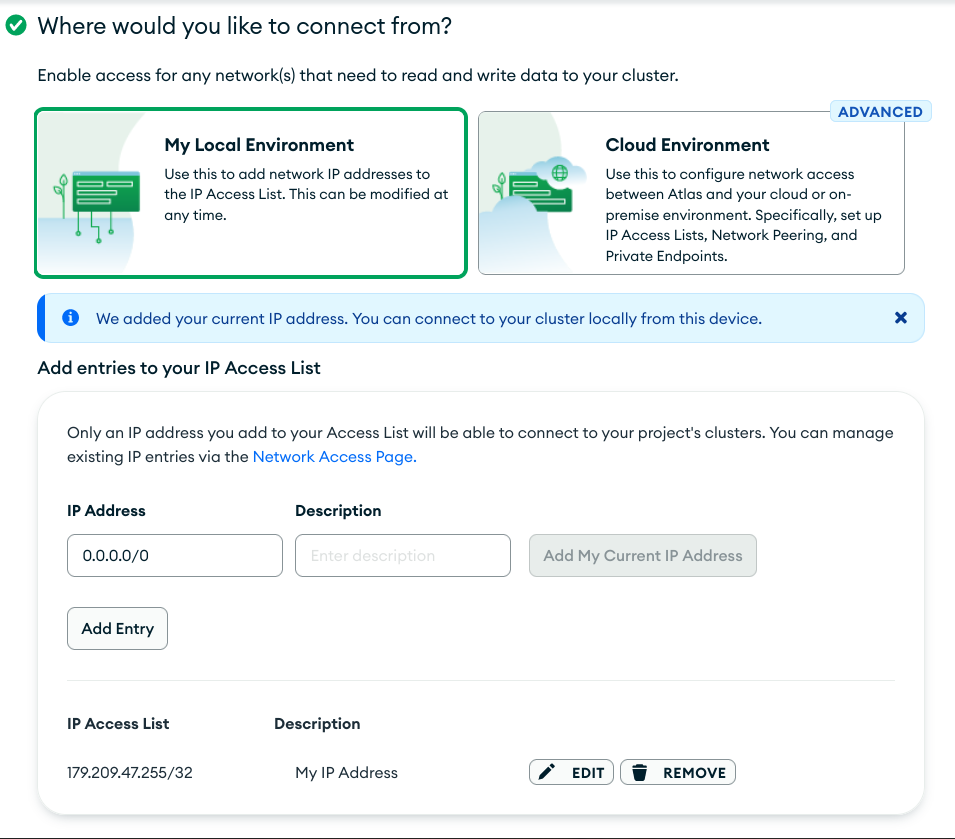


6 - Nas opções de segurança, crie um usuário com a opção **Username and Password** para acesso ao banco. Por exemplo, username admin e senha admin123. Clique em **Create user**.

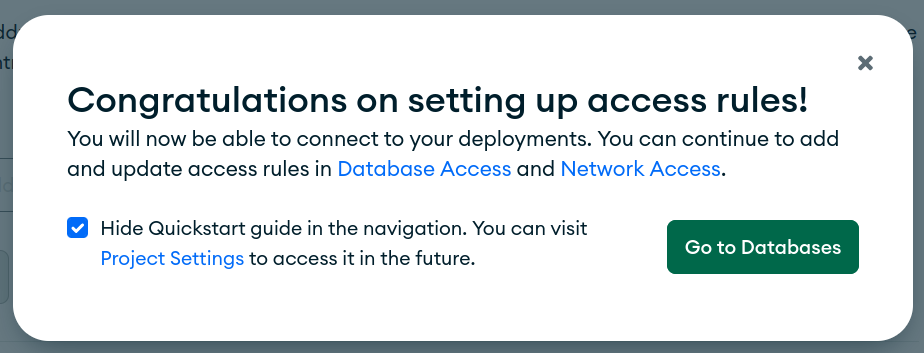


7 - Na seção **Where would you like to connect from?** escolha **My Local Environment**. O IP de sua máquina já está automaticamente adicionado como IP permitido para acesso, mas para ser possível acessar livremente o banco, adicione também o IP 0.0.0.0/0 e clique em **Add entry**. Em seguida conclua clicando em **Finish and close**.

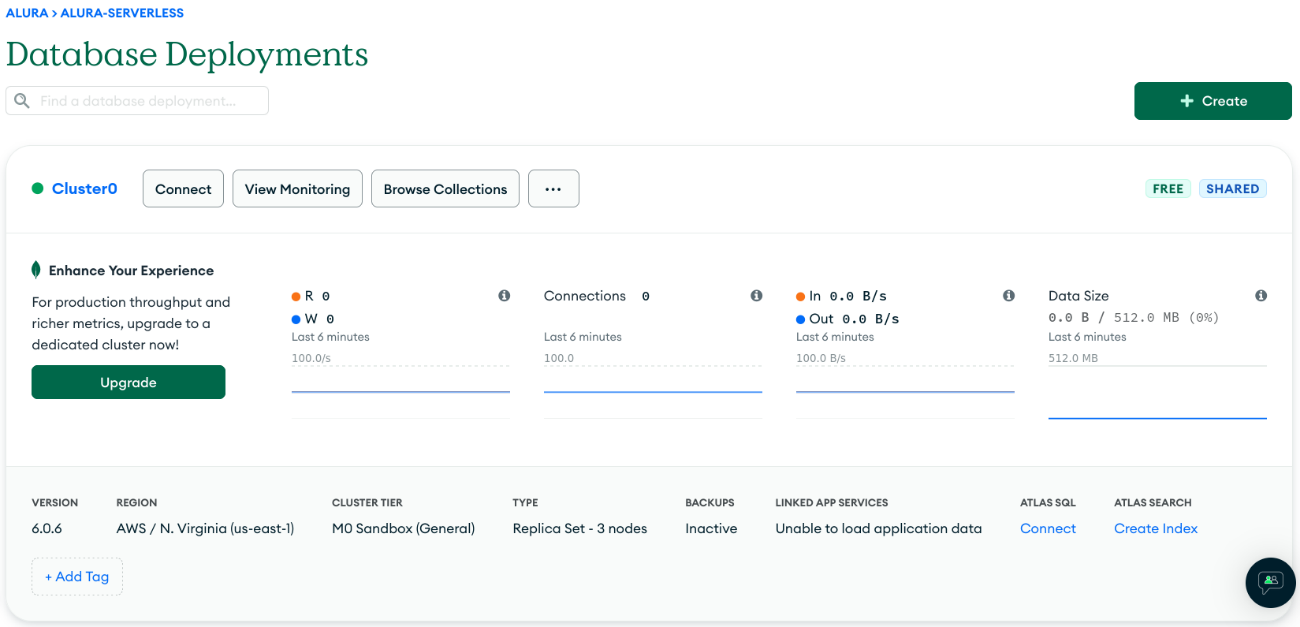
O acesso livre não é indicado para ambientes de produção, porém, no nosso caso podemos deixar dessa forma por ser um projeto de estudo.



8 - Clique em **Go to Databases** no modal para voltar para a tela principal de databases.



9 - Clique em **Connect** no Cluster0, que acabamos de criar.

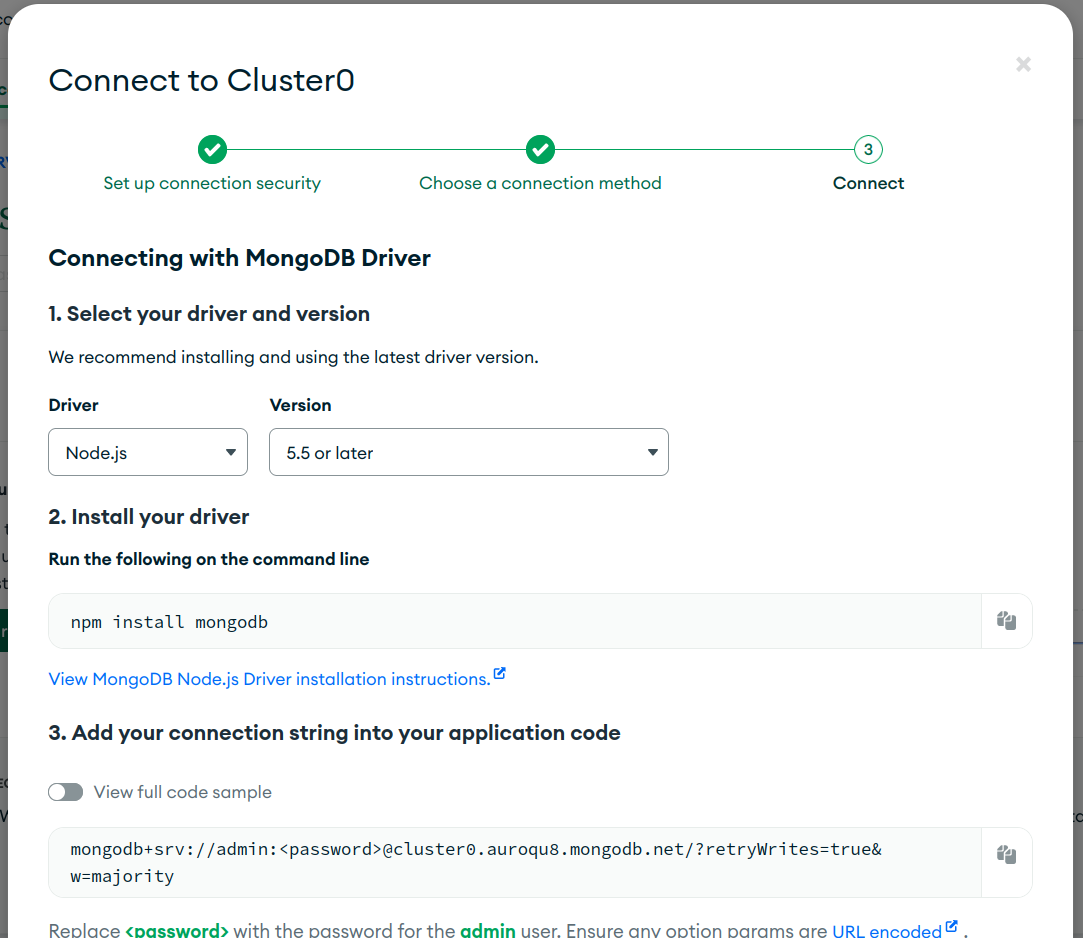


10 - No menu **Connect**, escolha a opção **Driver**. A instalação com npm install mongodb já foi feita pois a lib do MongoDB para Node.js já está adicionada no package.json então este passo não é necessário. Copie a string de conexão abaixo:

mongodb+srv://admin:<password>@cluster0.0dgjke6.mongodb.net/?retryWrites=**true**&w=majority

COPIAR CÓDIGO

Substitua admin:<password> pelos dados de username e password que você criou nos passos anteriores, por exemplo, admin:admin123. Clique em **Close** para fechar a janela.



A string de conexão será utilizada para fazer a conexão da API com o banco de dados que acabamos de criar. Copie o endereço e reserve para utilizarmos em seguida!

A partir de agora, continuaremos com o desenvolvimento de nossa API em vídeo.

**Para saber mais: tipos de servidores**

Durante nossos primeiros estudos, já evoluímos nosso conhecimentos em APIs com os temas:

* comunicação cliente-servidor via protocolo HTTP;
* tipos de bancos de dados;
* criação de uma instância de banco MongoDB no Mongo Atlas, **um serviço hospedado em nuvem**.

Também criamos um **servidor local** para acessar as rotas de nossa API via localhost:3000. Mas do que exatamente se trata quando falamos de servidores em uma comunicação cliente-servidor? De que **tipo** de servidores estamos falando?

O servidor que criamos para “servir” (ou fornecer) os dados da nossa API é voltado para uma aplicação capaz de receber dados, processá-los e enviar respostas.

Existem diversos tipos de servidores, além dos utilizados para hospedar e processar informações através de APIs REST. Cada tipo de servidor é pensado, construído e otimizado para sua função específica, por exemplo:

* **Servidores de banco de dados**: armazenam, gerenciam e acessam bancos de dados, como um servidor Oracle, MySQL ou MongoDB (não incluindo aqui servidores em nuvem como o Atlas).
* **Servidores de arquivos**: armazenam arquivos e dados. Ao contrário de um serviço de armazenamento de arquivos em “bucket”, oferecido pelos provedores de nuvem (como o AWS S3) e que normalmente pode ser acessado de qualquer lugar via internet, os servidores de arquivos geralmente são acessados apenas através de uma rede interna.
* **Servidores de arquivos estáticos**: armazenam e distribuem arquivos “estáticos”, como CSS, imagens e arquivos JavaScript utilizados em sites e conteúdos web. Ao contrário dos servidores que hospedam APIs (que precisam processar inputs e servir os resultados deste processamento), servidores de arquivos estáticos apenas “entregam” os arquivos solicitados pela requisição. Mesmo arquivos JavaScript, nesse caso, são considerados estáticos, pois toda a execução é feita pelo lado cliente da requisição (através, por exemplo, do navegador do usuário).

Ainda há vários outros tipos de servidores específicos, como os voltados para a análise de grandes volumes de dados, servidores de backup, de DNS, entre outros.

Mesmo quando utilizamos os serviços de armazenamento em nuvem, devemos sempre lembrar que a nuvem é, entre muitas aspas, “o servidor de outra pessoa”. Ao utilizarmos a nuvem, terceirizamos muito do trabalho de manutenção dos servidores para os provedores destes serviços, como AWS, Google Cloud, MS Azure, entre outros. Mas os princípios ainda são os mesmos.

**Criando um schema**

ando continuidade ao curso de Node, Marcos resolveu incluir um novo arquivo em models para representar o schema de um cliente da livraria.

Marcos criou o schema de acordo com o código abaixo:

**import** mongoose **from** “mongoose”;

**const** clienteSchema = **new** mongoose.**Schema**({

id: {type: **String**},

nome: {type: **Number**, required: **true**},

cpf: {type: **Number**},

endereco: {type: **String**}

});

**const** clientes = mongoose.**model**('clientes', clienteSchema);

**export** **default** clientes;

COPIAR CÓDIGO

Marque as alternativas corretas.

Top of Form

* Alternativa correta

Apesar de estar definido como Number, ao cadastrar um cliente indicando o CPF no formato 123.456.789-00, não haverá erro.

* Alternativa correta

Somente o nome será um campo obrigatório ao tentar cadastrar o um cliente através de uma requisição POST na API criada.

Ao incluir o parâmetro required como true, uma requisição feita sem o campo nome resultará em um erro de validação emitido internamente pelo Mongoose.

* Alternativa correta

O campo nome está definido de forma incorreta.

O campo nome está definido com o tipo Number, que é numérico. O correto seria defini-lo com o tipo String.

Bottom of Form

**Faça como eu fiz: revisando a criação de models**

Nesta aula conhecemos os **models** e criamos um model para livros utilizando os métodos do Mongoose.

Vamos revisar o código do modelo e do schema correspondente.

**No arquivo src/models/Livro.js**

* Importe o módulo mongoose;
* Crie um esquema chamado livroSchema usando o construtor mongoose.Schema;
* Defina as seguintes propriedades no esquema:
* id como tipo mongoose.Schema.Types.ObjectId;
  + titulo como tipo String e obrigatório;
  + editora como tipo String;
  + preco como tipo Number;
  + paginas como tipo Number.
* Passe um objeto como segundo parâmetro para mongoose.Schema com a propriedade versionKey definida como false;
* Crie um modelo chamado livro usando o método mongoose.model, passando a string livros como nome da coleção e o esquema livroSchema;
* Exporte o modelo livro como default (padrão).
* Confira como deve ficar o código do arquivo src/models/Livro.js:
* **import** mongoose **from** "mongoose";
* **const** livroSchema = **new** mongoose.**Schema**({
* id: { type: mongoose.Schema.Types.ObjectId },
* titulo: { type: **String**, required: **true** },
* editora: { type: **String** },
* preco: { type: **Number** },
* paginas: { type: **Number** }
* }, { versionKey: **false** });
* **const** livro = mongoose.**model**("livros", livroSchema);
* **export** **default** livro;
* COPIAR CÓDIGO
* Se tiver qualquer dúvida sobre o código, revise o que foi feito em vídeo, nas seções de Troubleshooting ou nos demais materiais extras.
* Caso a dúvida persista, você pode consultar o fórum ou conversar com a [comunidade no nosso Discord](https://discord.gg/Af6FGyAnyr)!

**Para saber mais: links da aula**

Seguem os links para os conteúdos mencionados durante a aula:

* [Documentação oficial do Mongoose](https://mongoosejs.com/) com exemplos, casos de uso e guia de uso da ferramenta.
* [Prisma](https://www.prisma.io/mongodb), opção como ferramenta de conexão do MongoDB a APIs Node.js. O Prisma pode ser usado com diversos gerenciadores de banco e oferece diversas funcionalidades avançadas.

**Para saber mais: Troubleshooting**

### Erro de resolução de promessas

Se, ao invés dos dados solicitados (por exemplo, um array de objetos ou um objeto livro), alguma das variáveis estiver retornando o valor Promise { <pending> }, é possível que exista alguma promessa não resolvida em alguma parte do código.

Verifique se todas as funções assíncronas do projeto, em especial as que fazem chamadas a métodos do Mongoose, estão devidamente assinaladas com async/await. Por exemplo:

app.**get**('/livros', **async** (req, res) => {

**const** listaLivros = **await** livro.**find**({});

res.**status**(200).**json**(listaLivros);

});

COPIAR CÓDIGO

Note que, no caso de funções callback, a palavra-chave async deve ser adicionada antes dos parâmetros e await antes da instrução da variável.

No caso de declarações de função, async deve ser adicionada antes de function:

**async** **function** **conectaNaDatabase**() {

mongoose.**connect**(process.env.DB\_CONNECTION\_STRING);

**return** mongoose.connection;

}

COPIAR CÓDIGO

Note que, no exemplo acima, não é necessário adicionar await pois return já é implicitamente assíncrono.

Em caso de dúvidas, basta consultar os commits feitos no [repositório do curso no GitHub](https://github.com/alura-cursos/3266-express-mongo/commits/aula-3).

### Nome do banco

Não esqueça de adicionar o nome da database livraria na string de conexão do Mongo Atlas, após o caminho principal, conforme feito em vídeo:

mongodb+srv://admin:<suasenha>@cluster0.uvmwiwx.mongodb.net/livraria?retryWrites=**true**&w=majority

COPIAR CÓDIGO

Isso garante que a conexão será feita para a database correta e a coleção livros será criada nesta database, especialmente se você tiver outras databases de teste em seu cluster.

**Para saber mais: sobre models e controllers**

Durante o desenvolvimento vimos na prática dois componentes-chave no desenvolvimento de uma API REST: **models** e **controllers**.

Quando desenvolvemos uma aplicação, seja uma API REST como a nossa livraria ou outros tipos de API, é normal organizarmos o código em “camadas”, ou seja, em conjuntos de código separados de acordo com sua função.

As possibilidades de organização do código em camadas dentro de uma aplicação são inúmeras e fazem parte de grandes áreas de estudo dentro da programação, como a **arquitetura de software**.

Para nossa livraria, escolhemos como base um dos modelos clássicos de arquitetura para APIs REST, que é o **MVC ou Model-View-Controller**. Neste curso, utilizaremos uma forma mais simplificada do MVC e não abordaremos o view, que é a camada de visualização dos dados.

Agora que implementamos nossos primeiros models e controllers, vamos entender mais sobre o papel de cada um desses componentes.

## Models

Os models são responsáveis pela representação dos dados e da lógica de negócio da aplicação, ou seja, é responsabilidade do model gerenciar o relacionamento entre a API e a camada de dados, incluindo validação, armazenamento e manipulação dos dados, interação com a base de dados e suas regras.

Em nossa API, a conexão entre o model e o banco de dados está sendo feito através dos métodos da biblioteca Mongoose, como no exemplo abaixo:

// importação do modelo Livro

// para ser chamado pelo controller

**import** livro **from** "../models/Livro.js";

**class** **LivroController** {

**static** **async** **listarLivros**(req, res) {

**try** {

// controller chama o model Livro através

// do método livro.find({})

**const** listaLivros = **await** livro.**find**({});

res.**status**(200).**json**(listaLivros);

} **catch** (erro) {

res

.**status**(500)

.**json**({ message: `${erro.message} - falha na requisição` });

}

}

};

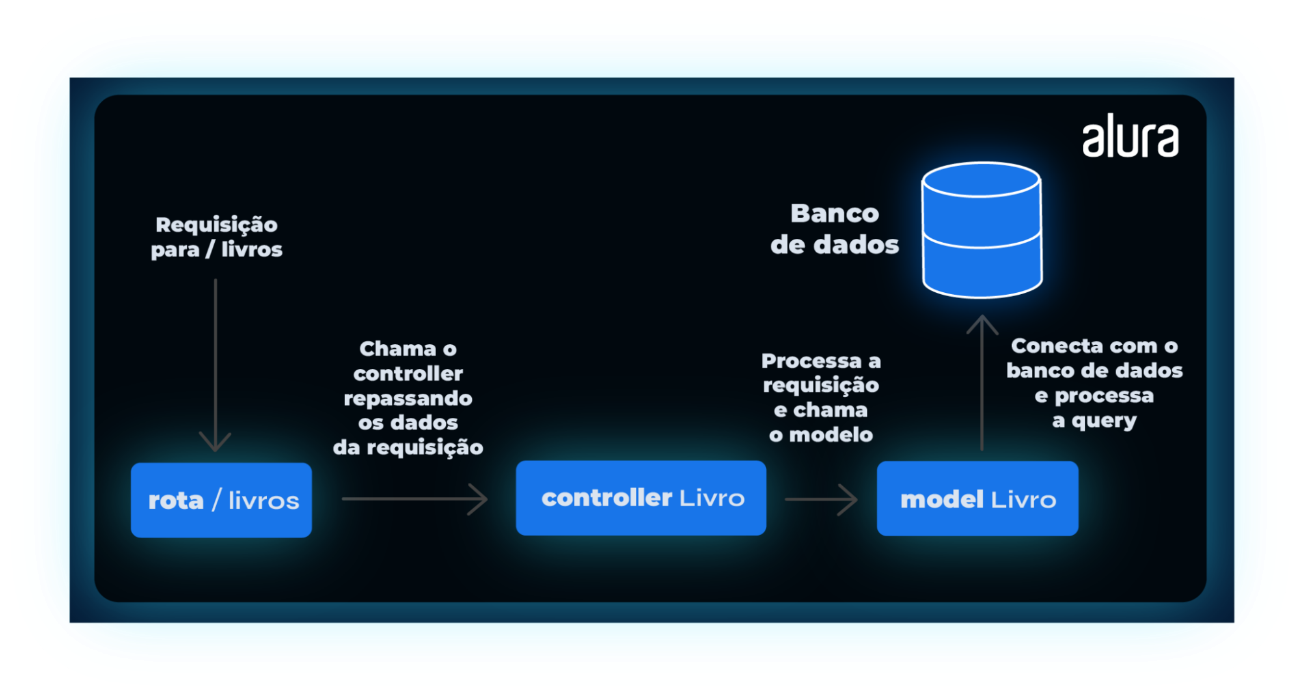
COPIAR CÓDIGO

Nossa API utiliza um banco MongoDB e a biblioteca Mongoose, então o código que escrevemos para criar o modelo utiliza a “sintaxe” da biblioteca. O código utilizado para definir um model varia de acordo com as diversas bibliotecas e frameworks que podem ser utilizados em uma API.

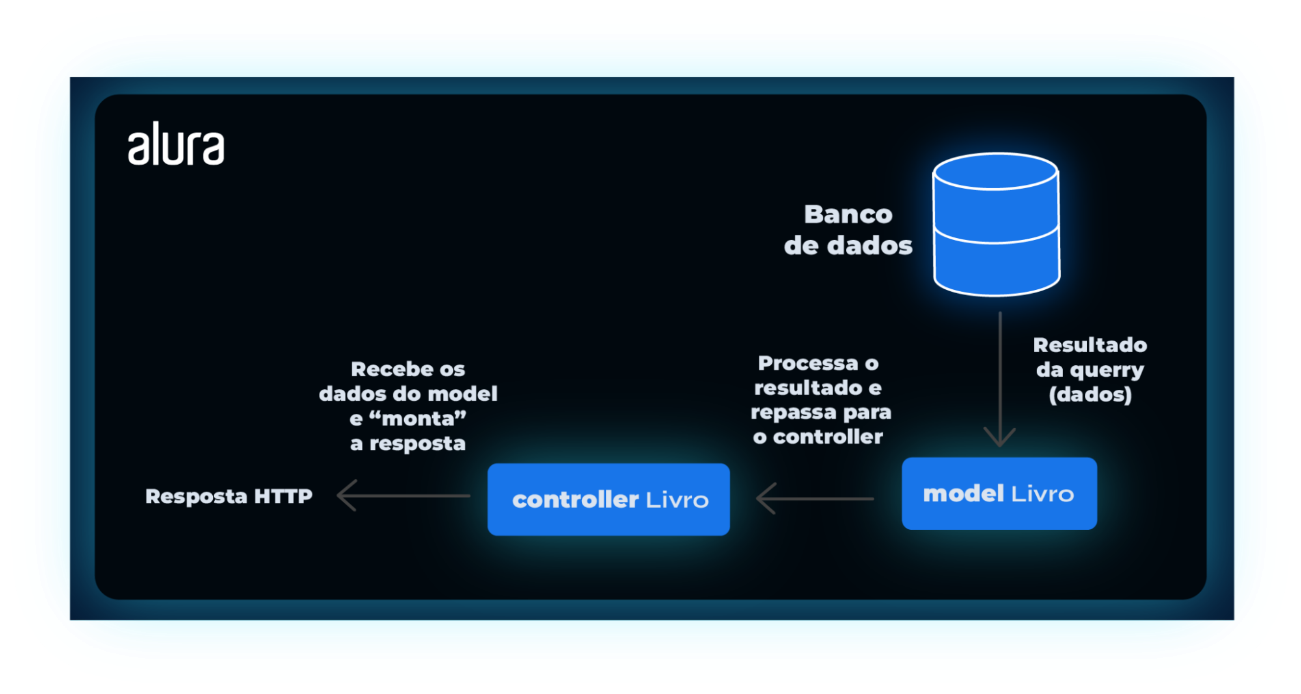
## Controllers

Os controllers atuam como intermediários entre a camada responsável pelo recebimento dos inputs de dados, que em nossa API está sendo feito através das rotas, e os models. Controllers também são a camada responsável pelo “caminho inverso”, pegando o resultado do processamento feito pelos models e “repassando” de volta.

Uma forma simplificada de ilustrar a integração entre as camadas da nossa API pode ser conferida neste diagrama:



Na imagem acima, ilustramos o fluxo da API REST usando o modelo MVC, desde a requisição HTTP até a consulta (query) feita ao banco de dados.

A comunicação entre as camadas de model e controller é uma caminho de ida e volta, pois o model vai repassar o resultado da consulta para o controller para que seja processada e transformada na resposta HTTP no formato adequado e com as informações esperadas

Para além do MVC, os conceitos de model e controller são amplamente usados em programação web, assim como a implementação e utilização de cada uma destas camadas. Você vai praticar mais com a arquitetura MVC nos próximos cursos da formação e terá contato com os models e os controllers em diversos outros tipos de API.

**Para saber mais: bloco try/catch**

Implementamos no código o que chamamos de **bloco try/catch** para “separar” os casos de sucesso das falhas no processamento das requisições.

O try/catch é uma estrutura usada justamente quando é necessário capturar e manejar erros de forma mais precisa, permitindo que os erros sejam “capturados e tratados” internamente pela função e evitando que a aplicação pare de funcionar ou que o cliente receba respostas inesperadas.

* No **bloco try** inserimos o código que é necessário “monitorar” para que potenciais erros sejam capturados, como operações em bases de dados ou APIs externas. Em caso de sucessos, apenas o código dentro do try é executado e seus resultados retornados para fora da função, conforme o caso.
* No **bloco catch** inserimos o código que vai ser executado em caso de erros ocorridos no código que está dentro do try. Qualquer erro que aconteça dentro do try é automaticamente lançado para dentro do catch e recebido através do parâmetro que normalmente chamamos de (error) ou (e). Uma vez dentro do catch o erro pode ser tratado de acordo e debugado, e o lado cliente pode receber uma resposta que faça sentido.

O try/catch funciona também para captura e tratamento de possíveis erros em operações assíncronas, como as que envolvem justamente operações em bancos ou outras APIs.

Tratamento de erros é uma parte importante do desenvolvimento de uma API. Após finalizarmos nossa API, você vai ter a oportunidade de continuar praticando no [próximo curso da formação de Node.js com Express](https://cursos.alura.com.br/course/node-js-buscas-filtros-paginacao-erros-api).

# Definindo as rotas

Vamos considerar que Marcos, neste momento, criou uma classe chamada EditoraController e nela incluiu métodos estáticos para manipulação dos dados das editoras dos livros.

Qual opção exemplifica a forma mais adequada de configurar a rota para atualizar os dados de uma editora?

Top of Form

* Alternativa correta
* .**put**(“/editora/:id”, **EditoraController**.atualizarEditora);

Aqui está sendo usado corretamente o verbo PUT, pois o endpoint recebe um id como parâmetro para identificar qual editora será atualizada, e o nome do método remete à operação que realmente desejamos realizar.

* Alternativa correta
* .**get**(“/editora/:id”, **EditoraController**.atualizarEditora);
* Alternativa correta
* .**put**(“/editora”, **EditoraController**.atualizarEditora);
* Alternativa correta

.**post**(“/editora/:id”, **EditoraController**.atualizarEditora);

Bottom of Form

**Faça como eu fiz: revisando a criação do controller**

Nesta aula conhecemos os **controllers** e entendemos sua função na API. Então, vamos revisar o processo de criação do livroController.

**No arquivo src/controllers/livroController.js:**

* Importe o modelo livro no início do arquivo;
* Crie a classe LivroController;
* Crie o método estático listarLivros, que recebe as requisições e as respostas como parâmetros;
* Dentro do método listarLivros, use o método find do modelo livro para buscar todos os livros no banco de dados;
* Retorne a lista de livros em formato JSON com o status 200.

**No arquivo src/routes/livrosRoutes.js:**

* Importe o módulo express no início do arquivo;
* Importe o controlador LivroController no início do arquivo;
* Crie um objeto routes que irá armazenar as rotas relacionadas aos livros;
* Defina a rota /livros dentro do objeto routes chamando o método get do módulo express e passando como argumentos /livros e o método LivroController.listarLivros;
* Exporte o objeto routes como um módulo.

Confira o código do arquivo src/controllers/livroController.js:

**import** livro **from** "../models/Livro.js";

**class** **LivroController** {

**static** **async** listarLivros (req, res) {

**const** listaLivros = **await** livro.**find**({});

res.**status**(200).**json**(listaLivros);

}

};

COPIAR CÓDIGO

Confira o código do arquivo src/routes/livrosRoutes.js:

**import** express **from** "express";

**import** **LivroController** **from** "../controllers/livroController.js";

**const** routes = express.**Router**();

routes.**get**("/livros", **LivroController**.listarLivros);

**export** **default** routes;

COPIAR CÓDIGO

Se tiver qualquer dúvida sobre o código, revise o que foi feito em vídeo, nas seções de Troubleshooting ou nos demais materiais extras.

Caso a dúvida persista, você pode consultar o fórum ou conversar com a comunidade no nosso [Discord](https://discord.gg/Af6FGyAnyr)!

**Para saber mais: Troubleshooting**

## Erro em HTML?

Se ao testar alguma rota no Postman, retornar uma mensagem de erro formatada com tags HTML, similar a este exemplo:

<!DOCTYPE **html**>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Error</title>

</head>

<body>

<pre>Cannot DELETE /livros/64b6b49e85a04a44cad5cdb6</pre>

</body>

</html>

COPIAR CÓDIGO

Verifique se a rota que você está testando foi adicionada corretamente ao projeto:

1. Se está corretamente adicionada ao arquivo correto na pasta routes;
2. Se o método HTTP escolhido no Postman (por exemplo, GET) corresponde ao informado em router, como router.get();
3. Se o módulo com as rotas correspondentes foi adicionado ao ponto de entrada das rotas routes/index.js.

**Para saber mais: relacionamentos**

Neste curso não vamos abordar com mais profundidade as questões de modelagem de dados e relacionamentos em bancos de dados.

Em bancos SQL, a avaliação de requisitos e modelagem dos dados costuma ser feita de forma adiantada com relação à aplicação, pois o SQL é, por definição, mais “estrito” com relação à estrutura de um banco, aos tipos de dados associados a cada campo e ao relacionamento entre as tabelas.

Já em um banco de objetos como o MongoDB, esta estrutura é menos rígida e pode ser atualizada de acordo com as necessidades da aplicação, então, dois documentos livro podem ser diferentes entre si sem que para isso sejam necessárias alterações significativas na base de dados.

Porém, em ambos os tipos de banco existem três tipos de relacionamentos entre dados. São eles:

1. **Relacionamento “um para um” (one-to-one ou 1:1)**, quando um registro está conectado a somente outro registro em outro conjunto de dados. **Exemplo**: uma pessoa registrada no sistema da nossa livraria tem somente um CPF relacionado a ela, e não é possível que duas pessoas diferentes tenham o mesmo CPF ou uma pessoa ter dois CPFs.
2. **Relacionamento “um para muitos” (one-to-many ou 1:n)**, quando um registro pode ser conectado a mais de um registro em outro conjunto de dados. **Exemplo**: as pessoas cadastradas em nossa livraria podem fornecer mais de um número de telefone celular, porém, cada um destes números de celular pode estar associado a apenas uma pessoa por vez.
3. **Relacionamento “muitos para muitos” (many-to-many ou n:m)**, quando mais de um registro pode estar relacionado a mais de um registro em outro conjunto. **Exemplo**: um livro pode ter mais de um autor, ao mesmo tempo que este mesmo autor pode ter escrito vários livros.

Durante o curso utilizamos a forma simplificada 1:n para o relacionamento entre livro e autor para focarmos nas funcionalidades da API.

A prática com bancos de dados envolve também a análise do produto que queremos e da aplicação como um todo para estabelecermos todos os relacionamentos entre as partes.

Além disso, existem as duas formas usuais de fazer estes relacionamentos: *embedding* e *referencing*, que falaremos mais adiante no curso.

# Para saber mais: MongoDB - Embed x Reference

Durante o último vídeo, aplicamos na prática a “união” entre dados de duas coleções diferentes usando o conceito de **embedding** ou “incorporação” e de **reference** ou “referência” (tradução livre). Confira abaixo mais informações sobre cada um deles:

## Embedding

Ao contrário do SQL, o MongoDB segue o princípio de **“dados que são acessados juntos devem ser armazenados juntos”**.

Pensando assim, embedding significa incorporar dados que são relacionados e inseri-los em um documento. É usado para simplificar as consultas (queries) aos dados e melhorar a performance geral das ferramentas nas consultas.

Veja abaixo um exemplo de documento autor com um array de livros incorporado ao restante dos dados. Arrays de objetos são uma forma comum de embedding de dados.

{

"\_id": ObjectId("579a7256f245878acabac457c"),

"nome": "JRR Tolkien",

"livros": [

{"titulo": "O Senhor dos Anéis", "paginas": 500},

{"titulo": "O Hobbit", "paginas": 200},

{"titulo": "O Silmarillion", "paginas": 400}

]

}

COPIAR CÓDIGO

Porém, incorporar dados em um único documento pode criar documentos muito grandes, o que pode acabar prejudicando a performance da aplicação, pois um documento deve ser carregado em memória por inteiro. Além disso, pode também fazer com que novos dados sejam incorporados indefinidamente a um único objeto e aumentando o tamanho em bytes além do limite de 16 mb por documento de um [objeto BSON](https://cursos.alura.com.br/extra/alura-mais/json-bson-entendendo-os-documentos-do-mongodb-c84).

## Referencing

Referencing significa fazer referência a documentos em outra coleção. Nesse caso, os dados são guardados em coleções separadas, mas ainda é importante que mantenham vínculo entre eles. A referência é feita através de um campo específico do documento, normalmente o campo id ou equivalente.

A agregação de dados via reference visa evitar duplicação de dados (um aspecto muito importante no SQL, também chamada de “normalização de dados”) e também gerenciar o tamanho dos documentos para evitar a criação de documentos muito grandes, que prejudicariam a performance do sistema.

Por outro lado, a junção de dados via referência faz com que uma consulta se transforme em duas ou várias. Por exemplo, uma consulta aos dados de um livro resultaria em consultas tanto na coleção livros quanto na coleção autores, o que consome mais recursos e pode impactar a performance de leitura.

{

"\_id": ObjectId("9gPOwsIJaf5knkzpvYNlk9flz"),

"nome": "JRR Tolkien",

"livros": [

ObjectId("ctstNHEEKCLwTN7gN7KgNprYb"),

ObjectId("qdQwKNumukFzuSYh58WKLN3TV"),

ObjectId("KNUeheO0UbtpFYwLuJpdwbD5P")

]

}

COPIAR CÓDIGO

Abaixo segue um resumo dos prós e contras de cada um dos métodos.

| | \*\*Embedding\*\* |

|:---:|:-------------------------------------------------:|

| PRO | Dados retornados em uma única consulta |

| PRO | atualização e exclusão de dados em única operação |

| CON | duplicação de dados |

| CON | documentos maiores |

| | \*\*Referencing\*\* |

| PRO | Não há duplicação de dados |

| PRO | documentos menores |

| CON | necessário "unir" dados de múltiplos documentos |

COPIAR CÓDIGO

## Quando utilizar um ou outro?

A resposta é **depende**, pois isso passa pela prática de modelagem de dados e também pelas características de cada tipo de dado e coleção. Neste curso não vamos avançar nesse tópico, pois modelagem é um assunto para seu próprio curso.

Tanto o MongoDB quanto os gerenciadores SQL (MySQL, PostgreSQL, entre outros) têm suas próprias formações nas quais você pode aprender com mais profundidade sobre o funcionamento de cada um deles. Confira abaixo:

* [Formação MongoDB](https://cursos.alura.com.br/formacao-mongodb);
* [Formação SQL com PostgreSQL](https://cursos.alura.com.br/formacao-postgresql);
* [Formação SQL com MySQL](https://cursos.alura.com.br/formacao-oracle-mysql).

Apesar de serem parte importante do desenvolvimento back-end, bancos de dados são assuntos muito vastos por si, então, não se preocupe neste momento em aprender todos os detalhes de cada um deles!

**Para saber mais: estrutura de uma URL**

Quando falamos de APIs como “pontos de contato” entre aplicações, é importante notar que este contato se dá justamente através de cada uma das URLs utilizadas, que dentro do contexto da API chamamos de “rotas”.

URLs (*Uniform Resource Locator* ou “localizador uniforme de recursos”, em tradução livre) são compostas por várias partes. Durante o último vídeo utilizamos uma dessas partes, que chamamos de *query parameters* ou “parâmetros de consulta”.

Vamos revisar e identificar as partes de uma URL utilizando como exemplo uma busca no site da Alura:

https://cursos.alura.com.br:443/search?query=express

COPIAR CÓDIGO

* https:// é o protocolo de comunicação utilizado.
* cursos.alura.com.br é o **host**, composto por **subdomínio** (cursos) e **domínio** (alura.com.br).
* :443/ é a porta utilizada na comunicação. 443 é a porta usada para conexões do tipo HTTPS e o número é normalmente suprimido na visualização padrão do navegador. No caso de conexões HTTP, a porta utilizada é a 80.
* search é a rota ou *path* (“caminho”), assim como nossa API tem as rotas livros e autores. Rotas de APIs, como a que estamos trabalhando, normalmente são “abstrações” referentes aos recursos. Porém, as rotas também podem ser usadas para a localização de recursos no servidor, por exemplo, /pages/sobre.html para exibir a página sobre no navegador.
* ?query=express são os **query parameters** ou parâmetros de busca. Note que os parâmetros de busca sempre iniciam com ?. Uma URL pode conter vários parâmetros encadeados, separados por &. Por exemplo, ?query=express&type=curso&formacao=node.

URLs também podem conter *fragments* ou *anchors*, comumente utilizadas em front-end como marcadores para determinadas partes de uma página e que não são enviadas ao servidor nas requisições.

A partir de agora, observe as URLs dos sites que visita e tente identificar cada uma das partes sendo utilizadas de diferentes formas!

Confira também o [Artigo Estrutura de uma URL](https://www.alura.com.br/artigos/estrutura-url" \t "_blank) para se aprofundar no tema.

# Relacionamentos

Começamos criando nossa livraria com o elemento principal: livros. A partir de **livro** é possível expandir para todo tipo de informação referente a ele, como **autor(es)** e **editora**.

Avalie as alternativas abaixo e marque as corretas.

Top of Form

* Alternativa correta

Durante o curso, foi utilizada a abordagem de **embedding** entre livros e autores apenas para fins de prática, não sendo uma abordagem utilizada no dia a dia em bancos de objetos.

* Alternativa correta

Utilizamos a abordagem do MongoDB para relacionamentos entre livros e autores: em bancos de objetos, informações que são acessadas em conjunto devem ser armazenadas em conjunto.

O MongoDB, como um banco de objetos, ao contrário dos bancos SQL, não se “preocupa” com a repetição de dados. Por exemplo, se temos 5 livros de um mesmo autor no banco, podemos salvar o mesmo objeto autor completo como um subdocumento de cada um dos cinco documentos livro diferentes.

* Alternativa correta

Os conjuntos de dados **autores** e **editoras** podem estar associados a diversos livros. Assim, podem ser representados em suas próprias coleções.

Livros, autores e editoras são grupos de informações (como nome, dados de contato e outros dados relevantes) e podem estar **relacionados** de diferentes formas, como vários livros em uma editora, várias editoras para vários autores etc. Assim, armazenamos esses dados em coleções diferentes para que possam ser gerenciados de forma independente e também relacionados entre si.

Bottom of Form

**Faça como eu fiz: usando Reference**

Durante a aula usamos o formato **embedding** para unirmos os dados de autor ao livro correspondente.

Confira abaixo um exemplo simples do mesmo processo, mas usando o formato **referencing**.

### No arquivo Livro.js:

Ao invés do autorSchema, adicionamos a propriedade autor tendo como valor um **ID** do tipo ObjectId do MongoDB, fazendo referência à coleção autores.

**const** livroSchema = **new** mongoose.**Schema**({

id: { type: mongoose.Schema.Types.ObjectId },

titulo: { type: **String**, required: **true** },

editora: { type: **String** },

preco: { type: **Number** },

paginas: { type: **Number** },

autor: {type: mongoose.Schema.Types.ObjectId, ref: 'autores', required: **true**},

}, { versionKey: **false** });

COPIAR CÓDIGO

### No arquivo livroController.js:

Quando usamos references o autor não faz mais parte do objeto livro. Assim, cada livro deve ser “populado” com as referências do autor.

Os métodos livro.find({}).populate("autor").exec(); vão utilizar o ID informado no campo autor do livro para buscar a referência desse ID e “popular” a propriedade.

**static** **async** listarLivros (req, res) {

**try** {

**const** listaLivros = **await** livro.**find**({}).**populate**("autor").**exec**();

res.**status**(200).**json**(listaLivros);

} **catch** (erro) {

res.**status**(500).**json**({ message: `${erro.message} - falha na requisição` });

}

};

COPIAR CÓDIGO

Já o método de cadastrar livro fica como estava anteriormente, pois agora o schema Livro apenas recebeu a propriedade autor com o tipo de dado ID.

**static** **async** cadastrarLivro (req, res) {

**try** {

**const** novoLivro = **await** livro.**create**(req.body);

res.**status**(201).**json**({ message: "criado com sucesso", livro: novoLivro });

} **catch** (erro) {

res.**status**(500).**json**({ message: `${erro.message} - falha ao cadastrar livro` });

}

}

COPIAR CÓDIGO

### Testes no Postman

Faça um novo teste no Postman com POST em http://localhost:3000/autores adicionando um novo autor:

{

"nome": "Machado de Assis",

"nacionalidade": "Brasil"

}

COPIAR CÓDIGO

O objeto adicionado na coleção autores deverá ter a seguinte estrutura:

{

"nome": "Machado de Assis",

"nacionalidade": "Brasil",

"\_id": "64c4303f71627bda06635b6f"

}

COPIAR CÓDIGO

Copie o ID do autor recém-criado e utilize este dado na criação de um novo livro no Postman com POST em http://localhost:3000/livros:

{

"titulo": "Dom Casmurro",

"autor": "64c4303f71627bda06635b6f"

}

COPIAR CÓDIGO

O retorno deverá ser um novo livro com a seguinte estrutura:

{

"message": "criado com sucesso",

"livro": {

"titulo": "Dom Casmurro",

"autor": "64c4303f71627bda06635b6f",

"\_id": "64c4306d71627bda06635b71"

}

}

COPIAR CÓDIGO

Agora, uma requisição do tipo GET em http://localhost:3000/livros deverá retornar o livro já com o autor “referenciado”:

{

"\_id": "64c4306d71627bda06635b71",

"titulo": "Dom Casmurro",

"autor": {

"\_id": "64c4303f71627bda06635b6f",

"nome": "Machado de Assis",

"nacionalidade": "Brasil"

}

}

**Faça como eu fiz: refatorando os outros métodos**

No vídeo anterior, refatoramos o método listarAutores para utilizar async/await em conjunto com o bloco try...catch do JavaScript. Atualmente o método está assim:

**static** listarAutores = **async** (req, res) => {

**try** {

**const** autoresResultado = **await** autores.**find**();

res.**status**(200).**json**(autoresResultado);

} **catch** (erro) {

res.**status**(500).**json**({ message: "Erro interno no servidor" });

}

};COPIAR CÓDIGO

O Mongoose permite que utilizemos a palavra-chave await antes de métodos como find, findById, findByIdAndUpdate, findByIdAndDelete, save, entre outros. Ao fazer isso, o código irá aguardar a operação ser realizada no banco de dados e o método irá retornar o resultado da operação. No código acima, guardamos os autores buscados na constante autoresResultado.

Além disso, em relação aos métodos que recebem mais parâmetros além da função callback, você irá mantê-los no código e remover apenas a função callback. Por exemplo, o código do método listarAutorPorId ficará assim:

**static** listarAutorPorId = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**const** autorResultado = **await** autores.**findById**(id);

res.**status**(200).**send**(autorResultado);

} **catch** (erro) {

res.**status**(400).**send**({message: `${erro.message} - Id do Autor não localizado.`});

}

}COPIAR CÓDIGO

Perceba que o parâmetro id do método findById foi mantido e removemos apenas a função callback para utilizar o async/await.

Agora é a sua vez de refatorar os outros métodos dos controladores de autores e de livros para utilizarem o async/await! Lembre-se de envolver o código de sucesso em um bloco try e de tratar os casos de erro com o catch.

Confira abaixo os códigos finais de cada método:

## Arquivo autoresController.js

* cadastrarAutor

**static** cadastrarAutor = **async** (req, res) => {

**try** {

**let** autor = **new** **autores**(req.body);

**const** autorResultado = **await** autor.**save**();

res.**status**(201).**send**(autorResultado.**toJSON**());

} **catch** (erro) {

res.**status**(500).**send**({message: `${erro.message} - falha ao cadastrar Autor.`});

}

};COPIAR CÓDIGO

* atualizarAutor

Nesse caso, como a API retorna apenas uma mensagem, e não o autor atualizado, não é necessário guardar o resultado de findByIdAndUpdate em uma variável, mas ainda é necessário utilizar a palavra-chave await, já que a consulta ao banco de dados é uma operação assíncrona. Se não for utilizado o await, a API irá retornar a mensagem de sucesso antes mesmo da operação ser realizada. Ao utilizar o await, o código irá aguardar a operação ser realizada ao banco de dados corretamente e, somente em caso de sucesso, o autorResultado será retornado. Em caso de erro, como você já sabe, o código do bloco catch será executado.

**static** atualizarAutor = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**await** autores.**findByIdAndUpdate**(id, {$set: req.body});

res.**status**(200).**send**({message: "Autor atualizado com sucesso"});

} **catch** (erro) {

res.**status**(500).**send**({message: erro.message});

}

};COPIAR CÓDIGO

* excluirAutor

**static** excluirAutor = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**await** autores.**findByIdAndDelete**(id);

res.**status**(200).**send**({message: "Autor removido com sucesso"});

} **catch** (erro) {

res.**status**(500).**send**({message: erro.message});

}

};COPIAR CÓDIGO

## Arquivo livrosController.js

* listarLivros

**static** listarLivros = **async** (req, res) => {

**try** {

**const** livrosResultado = **await** livros.**find**()

.**populate**("autor")

.**exec**();

res.**status**(200).**json**(livrosResultado);

} **catch** (erro) {

res.**status**(500).**json**({ message: "Erro interno no servidor" });

}

};COPIAR CÓDIGO

* listarLivroPorId

**static** listarLivroPorId = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**const** livroResultados = **await** livros.**findById**(id)

.**populate**("autor", "nome")

.**exec**();

res.**status**(200).**send**(livroResultados);

} **catch** (erro) {

res.**status**(400).**send**({message: `${erro.message} - Id do livro não localizado.`});

}

};COPIAR CÓDIGO

* cadastrarLivro

**static** cadastrarLivro = **async** (req, res) => {

**try** {

**let** livro = **new** **livros**(req.body);

**const** livroResultado = **await** livro.**save**();

res.**status**(201).**send**(livroResultado.**toJSON**());

} **catch** (erro) {

res.**status**(500).**send**({message: `${erro.message} - falha ao cadastrar livro.`});

}

};COPIAR CÓDIGO

* atualizarLivro

**static** atualizarLivro = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**await** livros.**findByIdAndUpdate**(id, {$set: req.body});

res.**status**(200).**send**({message: "Livro atualizado com sucesso"});

} **catch** (erro) {

res.**status**(500).**send**({message: erro.message});

}

};COPIAR CÓDIGO

* excluirLivro

**static** excluirLivro = **async** (req, res) => {

**try** {

**const** id = req.params.id;

**await** livros.**findByIdAndDelete**(id);

res.**status**(200).**send**({message: "Livro removido com sucesso"});

} **catch** (erro) {

res.**status**(500).**send**({message: erro.message});

}

};COPIAR CÓDIGO

* listarLivroPorEditora

**static** listarLivroPorEditora = **async** (req, res) => {

**try** {

**const** editora = req.query.editora;

**const** livrosResultado = **await** livros.**find**({"editora": editora});

res.**status**(200).**send**(livrosResultado);

} **catch** (erro) {

res.**status**(500).**json**({ message: "Erro interno no servidor" });

}

};

**Para saber mais: Responsabilidade do front-end ou do back-end?**

No vídeo anterior, fizemos um tratamento de erros mais detalhado em um dos métodos do controlador de autores.

Contudo, pode surgir a seguinte dúvida: o Front-end não poderia se encarregar de tratar esses erros e mostrar mensagens mais adequadas ao usuário final?

Para responder essa pergunta, já vou começar com o principal ponto dessa discussão: **devemos evitar de enviar para o cliente detalhes técnicos internos da nossa aplicação**, principalmente informações do banco de dados que possam ser sensíveis.

Mesmo que o front-end possa tratar a nossa requisição e mostrar uma mensagem de erro mais apropriada ao usuário final, ainda assim estamos expondo informações da API nas respostas. Nada garante que o front-end vai de fato tratar todas as mensagens. E, mesmo que trate, ainda assim seria possível uma pessoa mal intencionada fazer requisições diretamente para a nossa API e tentar obter e explorar informações sensíveis a partir das mensagens de erros.

Vamos considerar o seguinte erro como exemplo:

Cast to ObjectId failed **for** value \"6485e5aaad1084605f44a4f5H\" (type string) at path \"\_id\" **for** model \"Autor\"

COPIAR CÓDIGO

Esse erro é interno do mongoose, e contém informações como o nome do modelo e o nome do campo. Até mesmo o formato do erro pode denunciar que o banco utilizado é o MongoDB! Essas informações podem ser úteis para um atacante entender a estrutura do banco de dados ou explorar suas vulnerabilidades.

Essas mensagens de erros do Banco de Dados **servem apenas para as pessoas desenvolvedoras da API**, justamente para sabermos o que aconteceu e para que possamos tratar melhor esses erros em tempo de desenvolvimento.

## Mensagens de erro semânticas

Superada a motivação de proteger detalhes internos do sistema, ainda assim existem casos onde é interessante fornecer mensagens de erro mais adequadas para o cliente.

Por exemplo, se tiver havido um erro de requisição por parte do cliente, nós **não** devemos retornar um erro genérico de servidor com código de status 500. Se o usuário passar um ID inválido, então o erro deve informar apropriadamente que foi fornecida uma informação inválida e o código de status deve ser 400.

Ou seja, ao mesmo tempo que não devemos expor detalhes internos da API, devemos retornar informações semânticas que sejam úteis ao usuário final. Dessa forma, estamos protegendo o sistema e seguindo as boas práticas de construção de uma API!

Essa atividade surgiu por do seguinte tópico do forum, onde a Brenda trouxe o ótimo ponto de discussão: <https://cursos.alura.com.br/forum/topico-duvida-tratamento-de-erros-305676>

**Para saber mais: códigos de status HTTPPara saber mais:**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/node-js-buscas-filtros-paginacao-erros-api/task/124385/next)

Caso precise relembrar ou consultar os códigos HTTP que devem ser utilizados em cada situação, confira os seguintes links:

* [Documentação da MDN](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status);

# Códigos de status de respostas HTTP

Os códigos de status de resposta HTTP indicam se uma solicitação [HTTP](https://developer.mozilla.org/pt-BR/docs/Web/HTTP) específica foi concluída com êxito. As respostas são agrupadas em cinco classes:

1. [Respostas Informativas](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_informativas) (100 – 199)
2. [Respostas bem-sucedidas](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_bem-sucedidas) (200 – 299)
3. [Mensagens de redirecionamento](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#mensagens_de_redirecionamento) (300 – 399)
4. [Respostas de erro do cliente](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_de_erro_do_cliente) (400 – 499)
5. [Respostas de erro do servidor](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_de_erro_do_servidor) (500 – 599)

Os códigos de status listados abaixo são definidos por [RFC 9110](https://httpwg.org/specs/rfc9110.html#overview.of.status.codes).

**Nota:** Se você receber uma resposta que não esteja [nesta lista](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_informativas), é uma resposta não padrão, possivelmente personalizada para o software do servidor.

## [Respostas informativas](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_informativas)

[100 Continue](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/100)

Essa resposta provisória indica que o cliente deve continuar a solicitação ou ignorar a resposta se a solicitação já estiver concluída.

[101 Switching Protocols](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/101)

Esse código é enviado em resposta a um cabeçalho de solicitação [Upgrade (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) do cliente e indica o protocolo para o qual o servidor está mudando.

[102 Processing (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/102) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

Este código indica que o servidor recebeu e está processando a requisição, mas nenhuma resposta está disponível ainda.

[103 Early Hints](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/103) Experimental

Este código de status destina-se principalmente a ser usado com o cabeçalho [Link](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers/Link), permitindo que o agente do usuário inicie o [pré-carregamento (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/rel/preload) recursos enquanto o servidor prepara uma resposta.

## [Respostas bem-sucedidas](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_bem-sucedidas)

[200 OK](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/200)

A solicitação foi bem-sucedida. O significado do resultado de "sucesso" depende do método HTTP:

* GET: O recurso foi obtido e transmitido no corpo da mensagem.
* HEAD: Os cabeçalhos de representação são incluídos na resposta sem nenhum corpo de mensagem.
* PUT ou POST: O recurso que descreve o resultado da ação é transmitido no corpo da mensagem.
* TRACE: O corpo da mensagem contém a mensagem de requisição recebida pelo servidor.

[201 Created](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/201)

A requisição foi bem sucedida e um novo recurso foi criado como resultado. Esta é normalmente a resposta enviada após as solicitações POST ou algumas solicitações PUT.

[202 Accepted](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/202)

A solicitação foi recebida, mas ainda não foi atendida. É sem compromisso, pois não há como no HTTP enviar posteriormente uma resposta assíncrona indicando o resultado da solicitação. Destina-se a casos em que outro processo ou servidor manipula a solicitação ou processamento em lote.

[203 Non-Authoritative Information](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/203)

Esse código de resposta significa que os metadados retornados não são exatamente os mesmos que estão disponíveis no servidor de origem, mas são coletados de uma cópia local ou de terceiros. Isso é usado principalmente para espelhos ou backups de outro recurso. Exceto para esse caso específico, a resposta 200 OK é preferida a este status.

[204 No Content](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/204)

Não há conteúdo para enviar para esta solicitação, mas os cabeçalhos podem ser úteis. O agente do usuário pode atualizar seus cabeçalhos em cache para este recurso com os novos.

[205 Reset Content](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/205)

Diz ao agente do usuário para redefinir o documento que enviou esta solicitação.

[206 Partial Content](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/206)

Este código de resposta é usado quando o cabeçalho [Range](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Headers/Range) é enviado do cliente para solicitar apenas parte de um recurso.

[207 Multi-Status (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/207) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

Transmite informações sobre vários recursos, para situações em que vários códigos de status podem ser apropriados.

[208 Already Reported (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/208) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

Usado dentro de um elemento de resposta <dav:propstat> para evitar enumerar repetidamente os membros internos de várias ligações para a mesma coleção.

[226 IM Used (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/226) ([HTTP Delta encoding](https://datatracker.ietf.org/doc/html/rfc3229))

O servidor atendeu a uma solicitação GET para o recurso e a resposta é uma representação do resultado de uma ou mais manipulações de instância aplicadas à instância atual.

## [Mensagens de redirecionamento](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#mensagens_de_redirecionamento)

[300 Multiple Choices](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/300)

A solicitação tem mais de uma resposta possível. O agente do usuário ou usuário deve escolher um deles. (Não há uma maneira padronizada de escolher uma das respostas, mas links HTML para as possibilidades são recomendados para que o usuário possa escolher).

[301 Moved Permanently](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/301)

A URL do recurso solicitado foi alterada permanentemente. A nova URL é fornecida na resposta.

[302 Found](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/302)

Este código de resposta significa que o URI do recurso solicitado foi alterado temporariamente. Outras alterações no URI podem ser feitas no futuro. Portanto, esta mesma URI deve ser utilizada pelo cliente em requisições futuras.

[303 See Other](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/303)

O servidor enviou esta resposta para direcionar o cliente a obter o recurso solicitado em outro URI com uma solicitação GET.

[304 Not Modified](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/304)

É usado para fins de cache. Ele informa ao cliente que a resposta não foi modificada, portanto, o cliente pode continuar a usar a mesma versão em cache da resposta.

[305 Use Proxy](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#305_use_proxy) Deprecated

Definido em uma versão anterior da especificação HTTP para indicar que uma resposta solicitada deve ser acessada por um proxy. Foi descontinuado devido a questões de segurança em relação à configuração em banda de um proxy.

[306 unused](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#306_unused) Deprecated

Esse código de resposta não é mais usado, é apenas reservado. Foi usado em uma versão anterior da especificação HTTP/1.1.

[307 Temporary Redirect](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/307)

O servidor envia esta resposta para direcionar o cliente a obter o recurso solicitado em outra URI com o mesmo método usado na solicitação anterior. Tem a mesma semântica do código de resposta HTTP 302 Found, com a exceção de que o agente do usuário não deve alterar o método HTTP usado: se um POST foi usado na primeira solicitação, um POST deve ser usado no segundo pedido.

[308 Permanent Redirect](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/308)

Isso significa que o recurso agora está permanentemente localizado em outro URI, especificado pelo cabeçalho de resposta HTTP Location:. Isso tem a mesma semântica que o código de resposta HTTP 301 Moved Permanently, com a exceção de que o agente do usuário não deve alterar o método HTTP usado: se um POST foi usado na primeira solicitação, um POST deve ser usado no segundo pedido.

## [Respostas de erro do cliente](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_de_erro_do_cliente)

[400 Bad Request](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/400)

O servidor não pode ou não irá processar a solicitação devido a algo que é percebido como um erro do cliente (por exemplo, sintaxe de solicitação malformada, enquadramento de mensagem de solicitação inválida ou roteamento de solicitação enganosa).

[401 Unauthorized](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/401)

Embora o padrão HTTP especifique "unauthorized", semanticamente, essa resposta significa "unauthenticated". Ou seja, o cliente deve se autenticar para obter a resposta solicitada.

[402 Payment Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/402) Experimental

Este código de resposta está reservado para uso futuro. O objetivo inicial da criação deste código era usá-lo para sistemas digitais de pagamento, no entanto, este código de status é usado raramente e não existe nenhuma convenção padrão.

[403 Forbidden](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/403)

O cliente não tem direitos de acesso ao conteúdo; ou seja, não é autorizado, portanto o servidor está se recusando a fornecer o recurso solicitado. Ao contrário do 401 Unauthorized, a identidade do cliente é conhecida pelo servidor.

[404 Not Found](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/404)

O servidor não pode encontrar o recurso solicitado. No navegador, isso significa que o URL não é reconhecido. Em uma API, isso também pode significar que o endpoint é válido, mas o próprio recurso não existe. Os servidores também podem enviar esta resposta em vez de 403 Forbidden para ocultar a existência de um recurso de um cliente não autorizado. Este código de resposta é provavelmente o mais conhecido devido à sua ocorrência frequente na web.

[405 Method Not Allowed](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/405)

O método de solicitação é conhecido pelo servidor, mas não é suportado pelo recurso de destino. Por exemplo, uma API pode não permitir chamar DELETE para remover um recurso.

[406 Not Acceptable](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/406)

Esta resposta é enviada quando o servidor web, após realizar [negociação de conteúdo orientada pelo servidor](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Content_negotiation#negocia%C3%A7%C3%A3o_baseada_no_servidor), não encontra nenhum conteúdo que esteja em conformidade com os critérios fornecidos por o agente do usuário.

[407 Proxy Authentication Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/407)

É semelhante a 401 Unauthorized, mas a autenticação precisa ser feita por um proxy.

[408 Request Timeout](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/408)

Esta resposta é enviada por alguns servidores em uma conexão ociosa, mesmo sem qualquer requisição prévia pelo cliente. Isso significa que o servidor gostaria de desligar esta conexão não utilizada. Essa resposta é muito mais usada, pois alguns navegadores, como Chrome, Firefox 27+ ou IE9, usam mecanismos de pré-conexão HTTP para acelerar a navegação. Observe também que alguns servidores simplesmente encerram a conexão sem enviar esta mensagem.

[409 Conflict](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/409)

Esta resposta será enviada quando uma requisição conflitar com o estado atual do servidor.

[410 Gone](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/410)

Esta resposta é enviada quando o conteúdo solicitado foi excluído permanentemente do servidor, sem endereço de encaminhamento. Espera-se que os clientes removam seus caches e links para o recurso. A especificação HTTP pretende que esse código de status seja usado para "serviços promocionais por tempo limitado". As APIs não devem se sentir compelidas a indicar recursos que foram excluídos com esse código de status.

[411 Length Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/411)

O servidor rejeitou a solicitação porque o campo de cabeçalho Content-Length não está definido e o servidor o exige.

[412 Precondition Failed](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/412)

O cliente indicou nos seus cabeçalhos pré-condições que o servidor não atende.

[413 Payload Too Large](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/413)

A entidade requisição é maior do que os limites definidos pelo servidor. O servidor pode fechar a conexão ou retornar um campo de cabeçalho Retry-After.

[414 URI Too Long](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/414)

O URI solicitado pelo cliente é mais longo do que o servidor está disposto a interpretar.

[415 Unsupported Media Type](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/415)

O formato de mídia dos dados requisitados não é suportado pelo servidor, portanto, o servidor está rejeitando a requisição.

[416 Range Not Satisfiable](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/416)

O intervalo especificado pelo campo de cabeçalho Range na solicitação não pode ser atendido. É possível que o intervalo esteja fora do tamanho dos dados do URI de destino.

[417 Expectation Failed](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/417)

Este código de resposta significa que a expectativa indicada pelo campo de cabeçalho de solicitação Expect não pode ser atendida pelo servidor.

[418 I'm a teapot](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/418)

O servidor recusa a tentativa de coar café num bule de chá.

[421 Misdirected Request (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/421)

A requisição foi direcionada a um servidor inapto a produzir a resposta. Pode ser enviado por um servidor que não está configurado para produzir respostas para a combinação de esquema e autoridade inclusas na URI da requisição.

[422 Unprocessable Content](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/422) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

A solicitação foi bem formada, mas não pôde ser atendida devido a erros semânticos.

[423 Locked (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/423) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

O recurso que está sendo acessado está bloqueado.

[424 Failed Dependency (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/424) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

A solicitação falhou devido à falha de uma solicitação anterior.

[425 Too Early](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/425) Experimental

Indica que o servidor não está disposto a correr o risco de processar uma solicitação que pode ser repetida.

[426 Upgrade Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/426)

O servidor se recusa a executar a solicitação usando o protocolo atual, mas pode estar disposto a fazê-lo depois que o cliente atualizar para um protocolo diferente. O servidor envia um cabeçalho [Upgrade (en-US)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Upgrade) em uma resposta 426 para indicar os protocolos necessários.

[428 Precondition Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/428)

O servidor de origem exige que a solicitação seja condicional. Esta resposta destina-se a prevenir o problema de 'atualização perdida', onde um cliente pega (GET) o estado de um recurso, o modifica e o coloca (PUT) de volta no servidor, quando entretanto um terceiro modificou o estado no servidor, levando a um conflito.

[429 Too Many Requests](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/429)

O usuário enviou muitas requisições num dado tempo ("limitação de frequência").

[431 Request Header Fields Too Large](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/431)

O servidor não está disposto a processar a solicitação porque seus campos de cabeçalho são muito grandes. A solicitação pode ser reenviada após reduzir o tamanho dos campos do cabeçalho da solicitação.

[451 Unavailable For Legal Reasons](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/451)

O agente do usuário solicitou um recurso que não pode ser fornecido legalmente, como uma página da Web censurada por um governo.

## [Respostas de erro do servidor](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status#respostas_de_erro_do_servidor)

[500 Internal Server Error](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/500)

O servidor encontrou uma situação com a qual não sabe lidar.

[501 Not Implemented](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/501)

O método da requisição não é suportado pelo servidor e não pode ser manipulado. Os únicos métodos que servidores devem suportar (e portanto não devem retornar este código) são GET e HEAD.

[502 Bad Gateway](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/502)

Essa resposta de erro significa que o servidor, enquanto trabalhava como um gateway para obter uma resposta necessária para lidar com a solicitação, obteve uma resposta inválida.

[503 Service Unavailable](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/503)

O servidor não está pronto para manipular a requisição Causas comuns são um servidor em manutenção ou sobrecarregado. Note que junto a esta resposta, uma página amigável explicando o problema deveria ser enviada. Esta resposta deve ser usada para condições temporárias e o cabeçalho HTTP Retry-After deverá, se possível, conter o tempo estimado para recuperação do serviço. O webmaster deve também tomar cuidado com os cabeçalhos relacionados com o cache que são enviados com esta resposta, já que estas respostas de condições temporárias normalmente não deveriam ser postas em cache.

[504 Gateway Timeout](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/504)

Essa resposta de erro é fornecida quando o servidor está atuando como um gateway e não consegue obter uma resposta a tempo.

[505 HTTP Version Not Supported](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/505)

A versão HTTP usada na requisição não é suportada pelo servidor.

[506 Variant Also Negotiates](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/506)

O servidor tem um erro de configuração interna: o recurso variante escolhido está configurado para se envolver em negociação de conteúdo transparente e, portanto, não é um ponto final adequado no processo de negociação.

[507 Insufficient Storage](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/507) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

O método não pôde ser executado no recurso porque o servidor não pode armazenar a representação necessária para concluir a solicitação com êxito.

[508 Loop Detected](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/508) ([WebDAV (en-US)](https://developer.mozilla.org/en-US/docs/Glossary/WebDAV))

O servidor detectou um loop infinito ao processar a solicitação.

[510 Not Extended](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/510)

Extensões adicionais à solicitação são necessárias para que o servidor a atenda.

[511 Network Authentication Required](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/511)

Indica que o cliente precisa se autenticar para obter acesso à rede.

* [HTTP Status Dog](https://httpstatusdogs.com/);

https://httpstatusdogs.com/

* [HTTP Cats](https://http.cat/).

<https://http.cat/>

**Para saber mais: erros do Mongoose e o ObjectId**

No vídeo passado, aprendemos a identificar se o erro capturado no método listarAutorPorId era uma instância de mongoose.Error.CastError, que é um tipo de erro lançado pelo Mongoose quando ele falha em converter um valor (“cast error” significa “erro de conversão”, do inglês). Mas por que esse erro em específico foi lançado?

Quando obtemos um valor por parâmetro de rota da URL, ele sempre é tratado como uma string no código. Ou seja, no seguinte código do método listarAutorPorId, a constante id é uma string:

**const** id = req.params.id;COPIAR CÓDIGO

Na sequência, nós tentamos encontrar um autor no banco de dados utilizando o seguinte código:

**const** autorResultado = **await** autores.**findById**(id);COPIAR CÓDIGO

O método findById receberá uma string, e o Mongoose tentará convertê-la para ObjectId, um tipo reservado dessa biblioteca. O ObjectId abstrai o tipo de mesmo nome do MongoDB. Ou seja, o MongoDB também armazena dados do tipo ObjectId, que são exatamente os IDs únicos gerados para cada documento armazenado no banco de dados.

Tanto no MongoDB quanto no Mongoose, um ObjectId deve ser uma sequência de 12 ou 24 caracteres hexadecimais. Se o método findById receber uma string que não cumpre esses requisitos, ele falhará em converter a string para o tipo ObjectId e lançará um CastError.

Você pode ler mais sobre os erros internos do Mongoose na página [Error](https://mongoosejs.com/docs/api/error.html) da API.

**Como tratar cada erro?**

Nesta aula, você conheceu alguns casos de erro que podemos tratar em uma rota que recebe um dado pela URL. Lidamos, especificamente, com a rota que busca um autor pelo seu ID e recebemos esse ID como um parâmetro de rota. Sabemos que o método findById do Mongoose espera receber um ID do tipo ObjectId, que pode ser fornecido a partir de uma string de 12 ou 24 caracteres hexadecimais.

Com base nessas informações, marque as alternativas corretas sobre como cada caso de erro deve ser tratado nessa rota:

Top of Form

* Alternativa correta

Se o ID fornecido na URL estiver no formato esperado pelo Mongoose, mas não corresponder a nenhum autor cadastrado no banco de dados, devemos retornar um erro com código de status 400.

Na verdade, o apropriado para esse caso é um código de status 404 “Not Found” (“Não Encontrado”, do inglês) que informa que o recurso solicitado não foi encontrado. Ou seja, os dados fornecidos até estavam no formato esperado, mas não havia nenhuma correspondência no banco de dados.

* Alternativa correta

Se o ID fornecido na URL estiver no formato esperado pelo Mongoose, mas ainda assim um erro for lançado pelo código, podemos retornar um erro com código de status 500, informando que houve um problema interno do servidor.

* Alternativa correta

Caso o ID fornecido na URL não esteja no formato esperado pelo Mongoose, devemos retornar um erro com código de status 400.

O código de status 400 informa que houve um “Bad Request” (“Requisição Incorreta”, do inglês). Logo, se o dado fornecido não estiver no formato esperado pelo servidor, devemos gerar um erro de Requisição Incorreta. É possível identificar esse erro ao verificar se ele é uma instância de mongoose.Error.CastError.

Bottom of Form

oBS: middlewares são funcões que são interceptadores de requisições, esles interceptam uma requição ou ação e faz algo, no caso usado no programa foi um middlewares de manipulkação de erros.

**Para saber mais: controladores são middlewares?**

No vídeo passado, você aprendeu a criar um **middleware de manipulador de erros**, um middleware especial do Express caracterizado por receber quatro parâmetros que chamamos de: erro, req, res e next.

E talvez você tenha notado uma semelhança entre a estrutura dessa função e os métodos dos controladores da API, como o listarAutorPorId. Afinal, os métodos de controladores podem receber três dos quatro parâmetros que um middleware de erro pode receber, e que convencionalmente chamamos de req, res e next.

Isso porque **os métodos de controladores também são middlewares do Express**! Eles são o tipo mais convencional de middleware, que recebem até três parâmetros. Como dito na aula, um middleware do Express é uma função que é executada em toda requisição para a API ou em determinadas requisições.

Se pararmos para pensar, é exatamente assim que os métodos dos controladores se comportam: quando uma requisição é feita para uma determinada rota da API, esses métodos executam um determinado código (recebem parâmetros da requisição, acessam o banco de dados, etc) e devolvem uma resposta para o cliente.

Na verdade, a própria documentação diz que uma aplicação Express é essencialmente composta pela execução de várias funções middlewares em resposta às requisições!

Para entender como os métodos de controladores também são middlewares, vamos primeiro entender como registrar middlewares da forma mais básica.

## Registrando middlewares na aplicação Express

Para registrar um middleware que é executado em todas as requisições para a API, independente da rota ou do método HTTP, utilizamos o método app.use. Para verificar esse método funcionando, abra o arquivo app.js e encontre o seguinte código:

**const** app = **express**();

app.**use**(express.**json**());

**routes**(app);COPIAR CÓDIGO

Antes de routes(app), adicione o seguinte código:

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

res.**status**(200).**send**({ mensagem: "Resposta do novo middleware" });

});COPIAR CÓDIGO

A função acima tem acesso aos objetos de requisição e de resposta, pois eles podem ser acessados por qualquer middleware. Se você fizer qualquer requisição para a API no Postman, obterá como resposta o objeto { mensagem: "Resposta do novo middleware" }. Ou seja, o código desse middleware foi executado antes do código de qualquer um dos métodos dos controladores.

Isso porque **a ordem em que os middlewares são registrados na aplicação é importante**. Como o middleware acima foi registrado antes dos métodos dos controladores, seu código será executado primeiro para qualquer requisição. E se um middleware enviar uma resposta para o cliente (nesse caso, com o método send), o fluxo da requisição encerra nessa resposta, e quaisquer middlewares registrados depois desse não serão executados. Afinal, apenas uma resposta pode ser enviada para cada requisição.

Então, como fazer os próximos middlewares registrados serem executados? Para isso, podemos receber a função next como terceiro parâmetro (“next” significa “próximo”, do inglês). Altere o código do novo middleware para o seguinte:

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

console.**log**("Código de um novo middleware");

**next**();

});COPIAR CÓDIGO

E agora, se você fizer qualquer requisição para a API, você obterá o texto "Código de um novo middleware" no terminal, e os métodos dos controladores voltarão a processar corretamente as respostas para o cliente. Isso porque a função next foi executada, assim, será executado o próximo middleware registrado para a rota e o método HTTP solicitados.

Dentro de um middleware, se você não executar a função next e nem enviar uma resposta para o cliente, a requisição ficará em processamento até ser encerrada por causa do longo tempo de espera de resposta do servidor. Você pode testar isso no Postman!

## Registrando middlewares para rotas e métodos HTTP específicos

Com o método app.use, conferimos como registrar um middleware que é executado em toda requisição para a API. Mas como fazer para registrar um middleware que é executado apenas em uma requisição GET para a rota /livros, por exemplo?

Para isso, você pode alterar o código do novo middleware para o seguinte:

app.**get**("/livros", (req, res, next) => {

console.**log**("Middleware registrado no GET da rota /livros");

**next**();

});COPIAR CÓDIGO

Alteramos o método use para o método get. Agora, seu primeiro parâmetro é a string "/livros" e o segundo parâmetro é a função middleware. Se você fizer uma requisição GET para essa rota no Postman, obterá a frase "Middleware registrado no GET da rota /livros" no terminal, além da lista de livros cadastrados retornada pelo Postman.

Além do get, a constante app do Express também possui métodos como post, put e delete, que seguem a mesma estrutura do código acima e registram middlewares para requisições desses outros métodos HTTP.

Nesse momento, você deve estar achando o código acima muito semelhante à estrutura dos métodos dos controladores! A diferença é que a estrutura da nossa API está aplicando mais algumas camadas de abstração do Express, como roteadores e classes para os controladores. Podemos até dar uma conferida em como isso é feito no arquivo index.js da pasta routes. Atualmente, o arquivo possui o seguinte código:

**import** express **from** "express";

**import** livros **from** "./livrosRoutes.js";

**import** autores **from** "./autoresRoutes.js";

**const** **routes** = (app) => {

app.**route**("/").**get**((req, res) => {

res.**status**(200).**send**({titulo: "Curso de node"});

});

app.**use**(

express.**json**(),

livros,

autores

);

};

**export** **default** routes;COPIAR CÓDIGO

Nesse código, o método app.use está sendo utilizado para registrar três middlewares:

* express.json(): esse é um middleware nativo do Express que converte os dados de uma requisição para o formato JSON;
* livros e autores: são roteadores criados no curso anterior, nos arquivos livrosRoutes.js e autoresRoutes.js, e que podem ser utilizados como middlewares.

E para entender melhor os roteadores registrados, vamos conferir, por exemplo, o arquivo livrosRoutes.js. Nele, temos o seguinte código:

**import** express **from** "express";

**import** **LivroController** **from** "../controllers/livrosController.js";

**const** router = express.**Router**();

router

.**get**("/livros", **LivroController**.listarLivros)

.**get**("/livros/busca", **LivroController**.listarLivroPorEditora)

.**get**("/livros/:id", **LivroController**.listarLivroPorId)

.**post**("/livros", **LivroController**.cadastrarLivro)

.**put**("/livros/:id", **LivroController**.atualizarLivro)

.**delete**("/livros/:id", **LivroController**.excluirLivro);

**export** **default** router; COPIAR CÓDIGO

Nesse arquivo, foi criado um roteador com o código const router = express.Router(). Um roteador do Express basicamente é capaz de registrar um grupo de middlewares para determinadas rotas e métodos HTTP, para proporcionar maior organização na aplicação. Perceba que ele segue uma sintaxe muito semelhante à que acabamos de conhecer: a de registrar um middleware diretamente em app. No roteador, para cada método get, post, put ou delete, foi registrado um método de controlador, que está sendo utilizado como middleware.

Os métodos dos controladores, por sua vez, possuem acesso aos parâmetros req, res e next. Ou seja, esses métodos podem:

* Obter dados de uma requisição com o parâmetro req;
* Executar qualquer código, como operações no banco de dados;
* E devolver uma resposta para o cliente com o parâmetro res ou executar o próximo middleware registrado na aplicação com o parâmetro next.

Com isso, apuramos em detalhes por que os métodos dos controladores da API também são middlewares! Se você quiser ler mais sobre esses recursos, você pode conferir as seguintes páginas da documentação do Express:

* [Writing middleware](https://expressjs.com/en/guide/writing-middleware.html) => <https://expressjs.com/en/guide/writing-middleware.html>
* [Using middleware](https://expressjs.com/en/guide/using-middleware.html) => <https://expressjs.com/en/guide/using-middleware.html>
* [Router](https://expressjs.com/en/api.html#router) => <https://expressjs.com/en/api.html#router>

**Para saber mais: logando os erros**

Atualmente, o código do middleware de erros (manipuladorDeErros.js) está assim:

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

**if** (erro **instanceof** mongoose.Error.CastError) {

res.**status**(400).**send**({message: "Um ou mais dados fornecidos estão incorretos."})

} **else** {

res.**status**(500).**send**({message: "Erro interno de servidor."})

}

});COPIAR CÓDIGO

E, sabendo que todos os códigos dos métodos de controladores estão envolvidos por blocos try/catch, até mesmo um erro de código gerado pela própria pessoa desenvolvedora será capturado e encaminhado para esse manipulador de erros (até mesmo o famoso erro de tentar acessar um propriedade de null ou undefined).

Em outras palavras, os erros lançados (throw) pela aplicação não aparecem mais no terminal como normalmente apareceriam. Isso porque, quando um erro é lançado e não é tratado, ele automaticamente é imprimido no terminal, além de derrubar o nosso servidor.

Porém, como estamos tratando praticamente todos os erros da aplicação (não só os gerados por uma requisição, mas também até erros de código escritos pela pessoa desenvolvedora), os erros não aparecerão mais automaticamente no terminal.

Com isso, pode ser que aconteça de você escrever algum código errado e o Postman retorne "Erro interno de servidor" quando você fizer uma requisição, mas você não tem ideia de onde o erro surgiu. Então, para te ajudar a entender qualquer erro que possa ser gerado pela aplicação, você pode simplesmente imprimi-lo no manipulador de erros com um console.log() ou console.error(). O código do manipulador de erros ficaria agora assim:

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

console.**log**(erro); // imprime o erro para a pessoa desenvolvedora

**if** (erro **instanceof** mongoose.Error.CastError) {

res.**status**(400).**send**({message: "Um ou mais dados fornecidos estão incorretos."})

} **else** {

res.**status**(500).**send**({message: "Erro interno de servidor."})

}

});COPIAR CÓDIGO

Espero que isso te ajude a identificar os erros mais facilmente!

## Tipos de erros

Existe até uma classificação que define os erros de uma aplicação em dois tipos: **erros operacionais** e **erros de programação**.

Os erros operacionais são os que viemos tratando até o momento, como um dado incorreto vindo do front-end, ou mesmo um problema interno do Banco de Dados.

Já os erros de programação, que são os famosos bugs, envolvem um erro de escrita do próprio código, como um erro de sintaxe ou o erro de tentar acessar um propriedade de null ou undefined. Os erros de programação são até lançados como classes de erros específicas do JavaScript, que você pode ler melhor no artigo [Lidando com erros no Node.js](https://www.alura.com.br/artigos/lidando-com-erros-node-js) => [https://www.alura.com.br/artigos/lidando-com-erros-node-js?\_gl=1\*1kheh7c\*\_ga\*MTk3MzI1MzA4NS4xNzA4NjE2OTUx\*\_ga\_1EPWSW3PCS\*MTcwODgxNDkzOC4xMi4xLjE3MDg4MTQ5MzkuMC4wLjA.\*\_fplc\*U2FXVkcya1l2bVR6cXN3dmtlb1MzRiUyQnM3S0M0dVo3MGFJOWRQRWdKNG5LSXlmTzBsYXhiQ0w3WXZIMk1KTUIxbyUyQjVvbkc4R2RqMXAwSTR3SHhBd2dPdnNrMDFXaXZrMEtGbCUyRjhuUmQ2RmNkbVdVSTE5em9zcDl1UHdwUFpnJTNEJTNE](https://www.alura.com.br/artigos/lidando-com-erros-node-js?_gl=1*1kheh7c*_ga*MTk3MzI1MzA4NS4xNzA4NjE2OTUx*_ga_1EPWSW3PCS*MTcwODgxNDkzOC4xMi4xLjE3MDg4MTQ5MzkuMC4wLjA.*_fplc*U2FXVkcya1l2bVR6cXN3dmtlb1MzRiUyQnM3S0M0dVo3MGFJOWRQRWdKNG5LSXlmTzBsYXhiQ0w3WXZIMk1KTUIxbyUyQjVvbkc4R2RqMXAwSTR3SHhBd2dPdnNrMDFXaXZrMEtGbCUyRjhuUmQ2RmNkbVdVSTE5em9zcDl1UHdwUFpnJTNEJTNE)

**Faça como eu fiz: tratando mais recursos inexistentes**

No vídeo passado, atualizamos o método listarAutorPorId para encaminhar o erro NaoEncontrado caso o autor não fosse encontrado pelo ID. Atualmente o método está assim:

**static** listarAutorPorId = **async** (req, res, next) => {

**try** {

**const** id = req.params.id;

**const** autorResultado = **await** autores.**findById**(id);

**if** (autorResultado !== **null**) {

res.**status**(200).**send**(autorResultado);

} **else** {

**next**(**new** **NaoEncontrado**("Id do Autor não localizado."));

}

} **catch** (erro) {

**next**(erro);

}

};COPIAR CÓDIGO

Agora é a sua vez de fazer essa tratativa de ID não encontrado nos outros métodos! Os métodos findById, findByIdAndUpdate, findByIdAndDelete também retornam null caso o ID não seja encontrado. Modifique os métodos restantes do controlador de autores e também os métodos do controlador de livros.

**Encaminhando erros**

Nesta aulas, você aprendeu como tratar diferentes tipos de erros na aplicação, como erros do Mongoose, página 404 ou erros internos do servidor. O tratamento desses erros foi centralizado no seguinte middleware de manipulador de erros:

**function** **manipuladorDeErros** (erro, req, res, next) {

**if** (erro **instanceof** mongoose.Error.CastError) {

**new** **RequisicaoIncorreta**().**enviarResposta**(res);

} **else** **if** (erro **instanceof** mongoose.Error.ValidationError) {

**new** **ErroValidacao**(erro).**enviarResposta**(res);

} **else** **if** (erro **instanceof** **NaoEncontrado**) {

erro.**enviarResposta**(res);

} **else** {

**new** **ErroBase**().**enviarResposta**(res);

}

}COPIAR CÓDIGO

Mas para que a tratativa de erros seja realizada corretamente, qual a forma correta de encaminhar um erro para esse middleware?

Top of Form

* Alternativa correta

Dentro de um método de controlador ou de outro middleware, precisamos receber um terceiro parâmetro, que é uma função convencionalmente chamada de next e executá-la sem parâmetros.

* Alternativa correta

Os erros lançados por um método de controlador ou outro middleware sempre são automaticamente enviados para o manipulador de erros, e não é necessário realizar nenhuma tratativa adicional no código.

* Alternativa correta

Dentro de um método de controlador ou de outro middleware, precisamos receber um terceiro parâmetro, que é uma função convencionalmente chamada de next e executá-la passando o erro como parâmetro.

Todo middleware tem acesso à função next, que pode ser utilizada para executar o próximo middleware registrado na aplicação ou para executar diretamente o manipulador de erros quando recebe um erro como parâmetro.

Bottom of Form

**Para saber mais: documentação do Mongoose**

Nos últimos vídeos, você aprendeu como realizar validações nativas do Mongoose e validações personalizadas, que possibilitam verificações mais complexas. Caso você fique em dúvida em relação à sintaxe desses recursos, consulte a página [Validation](https://mongoosejs.com/docs/validation.html) da documentação!

<https://mongoosejs.com/docs/validation.html>