[00:00] Boas-vindas ao treinamento de Node.js: API Rest com Express e MongoDB. Sou Jaqueline Oliveira e desejo à você boas-vindas ao nosso curso de API Rest.

[00:06] No curso nós vamos utilizar o Node que é um ambiente de execução propriamente para rodar JavaScript fora do navegador. Nós vamos utilizá-lo juntamente com o Express, que é um framework que vai nos auxiliar na construção da API seguindo o padrão Rest e vamos utilizar também o banco de dados MongoDB, que é um banco Nocycle (não relacional) e vamos partir de toda a configuração dele no ambiente Atlas, que é a versão dele que roda na nuvem.

[00:43] O objetivo vai ser construir uma API para a Livraria Alura. Nós vamos ter métodos para buscar, cadastrar, atualizar e excluir um livro específico. Não só livros, mas autores também, seguindo as boas práticas do Rest e fazendo isso de forma bem passo a passo para que você consiga entender qual é o objetivo de uma API, como ela funciona, qual é a melhor forma prática de utilizar, de programá-la.

[01:20] E também para você entender o conceito de requisição e resposta e como que o protocolo HTTP está envolvido nisso, porque fazemos uso dos verbos HTTP. Vamos trabalhar também com objetos no formato JSON, nossas requisições e respostas estarão trafegando JSON por isso vamos isso o tempo inteiro no curso.

[01:44] Para dar um spoiler da aplicação eu vou mostrar um pouco no VS Code, vamos usar o Mongoose também para conectar com o banco de dados Mongo e toda a API vai seguir as boas práticas de ter controlador, ter a separação do models, das rotas, a configuração com o banco toda separada nos arquivos corretamente e aí nós vamos utilizar também o Postman para obter os dados.

[02:13] Vamos fazer todas as requisições pelo Postman e ter acesso a todos os dados que cadastrarmos e fazer as atualizações, tudo pelo Postman. Exatamente como é feito quando você está usando um front-end, só que ao invés de programarmos o front-end para fazer requisições nós vamos usar o Postman como ferramenta para simular todas essas requisições, fazer o servidor local processar esses dados e devolver as respostas considerando os status adequados.

[02:45] Espero você, estou bem animada para dividir esse conteúdo com você e te vejo no próximo vídeo.

Nesse curso usaremos o Node.js para construir uma API REST utilizando o Express. Para isso, precisaremos fazer a instalação de algumas ferramentas. Se você já fez o(s) curso(s) que constam como pré-requisito para este, é muito provável que o ambiente já esteja pronto. 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 acessar 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 - caso retorne um número de versão, como por exemplo, v16.14.0, o NodeJS já está instalado.

Caso precise instalar, siga as instruções para cada sistema operacional que estão na página inicial do NodeJS. Este curso foi desenvolvido usando a versão 16.14.0.

O site do Node.js oferece duas opções para download, a LTS e a current (atual), conforme imagem abaixo. 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.

Print da tela do navegador na página inicial do NodeJS,com fundo branco e com opção de download da versão 16.14.0 LTS ou a 17.7.1 Atual, disponibilizadas em retângulos verdes clicáveis. Recomendamos sempre o download da versão LTS

Visual Studio Code

O Visual Studio Code é o editor que escolhemos utilizar para escrever nossos códigos durante este curso. Os links para baixar e instalar de acordo com o seu sistema operacional estão na página inicial do VSC.

Postman

Utilizaremos também o Postman para testar a nossa API.

Com essas informações, agora sim, podemos começar a usar o Node.js e iniciar nosso curso! Vamos lá?!

[00:00] Olá, sou a Jaqueline Oliveira, vou acompanhar você no treinamento de API com Express e MongoDB. Se você realizou o curso de NodeJS: criando sua primeira biblioteca que está como pré-requisito para esse curso, já está um pouco familiarizado com o Node e já sabe que o JavaScript não é uma linguagem exclusiva do front-end, ele é uma linguagem também usada para o back-end e é o que vamos trabalhar ao longo do curso.

[00:27] Eu vou rever alguns conceitos rapidamente nesse vídeo, só para ficar bem claro o que pretendemos entregar no final do curso, qual é o objetivo final, o que queremos atingir.

[00:40] Vamos começar falando um pouco sobre o back-end e o front-end, o que são esses conceitos que são tão falados. Vamos supor que quando você está no seu navegador, você está digitando um endereço, por exemplo, o endereço da Alura, você está trabalhando como se você fosse um cliente, você está digitando o endereço e em algum outro computador do outro lado do mundo da nuvem aonde quer que esteja, um servidor vai processar essas informações, esses dados e vamos devolver para que o seu navegador consiga exibir a página que você precisa, um conteúdo que está proposto.

[01:23] Esse processo acontece no servidor de buscar as informações e os clientes, a comunicação entre eles ocorrem através do protocolo HTTP, que também faz parte da formação, o curso de HTTP para ficar bem claro como acontece essa comunicação e quais são as características do protocolo.

[01:48] Toda a parte dos clientes, de você no seu navegador buscando as informações, o que está acontecendo no seu navegador você está fazendo uma requisição e o servidor está devolvendo uma resposta. Basicamente é assim, vamos trabalhar com requisições e respostas.

[02:10] Nós consideramos que tudo o que está acontecendo no servidor é responsabilidade do back-end, filtrar dados, processar as informações, ver as regras de negócio e o que é exibido ali na sua interface gráfica, o que você interage, os menus, os botões é o que consideramos aplicação front-end.

[02:33] É um resumo bem genérico, só para ficar mais claro o que vamos fazer e vamos falar de API. API é a sigla que significa Application Programming Interface, que seria Interface de Programação de Aplicação. Essas siglas sempre deixam a pessoa um pouco confusa.

[02:53] O que basicamente significa isso, vamos pensar nisso de uma forma bem simples. Você tem uma cafeteria e você senta na mesa, você e outras pessoas, você é o cliente e aí você vai chamar a pessoa, a garçonete e vai pedir o que você quer. "Eu quero um Frapuccino". A garçonete vai à cozinha, vai falar para o barista que a mesa 1 quer um Frapuccino e o barista vai preparar o pedido e quando estiver pronto ele vai avisar que está pronto, que conseguiu atender o que foi pedido.

[03:35] E vai passar de novo para a garçonete que vai devolver para a mesa 1, e aí o cliente vai gostar, vai agradecer e elogiar o pedido. Assim, de forma bem básica, temos três entidades, alguém está fazendo uma requisição, pediu de um jeito que a garçonete entendeu, não importava se o cliente era americano, japonês, mulher, homem, idoso, criança, falando o pedido da forma certa a garçonete entendeu, solicitou o pedido, o pedido foi preparado e ela devolveu isso como uma resposta.

[04:21] Quando estamos falando de API é basicamente a mesma coisa, você tem um sistema que pode ser Web ou mobile que vai fazer uma requisição, à API vai entender aquela requisição, vai processar os dados , coletar as informações, os dados, as funcionalidades, seja o que for que ela precise devolver e vai devolver como uma resposta para quem solicitou.

[04:48] O tempo inteiro vamos está trabalhando com requisições e respostas. E essas respostas geralmente vão acontecer em alguns formatos, os formatos mais comuns são JSON e XML. Para as APIs REST o formato mais utilizado hoje em dia é o JSON, nas antigas APIs SOAP que ainda tem, mas são menos utilizadas, trafegava muito XML.

[05:16] Já que falamos em API REST, o REST é um modelo de arquitetura que vai definir como que os sistemas distribuídos que tem essa arquitetura cliente/servidor que comentei exemplificando com a garçonete e o cliente sendo a API e o sistema, como que esses sistemas distribuídos vão se comunicar. O REST é um acrônimo para Representational State Transfer, que seria Transferência de Estado Representacional, outro nome complicado.

[05:57] Vamos falar um pouco sobre alguns aspectos, o primeiro que podemos citar é que tudo vai ter um conceito de recurso. Quando temos uma API ou uma aplicação de forma geral, estamos gerenciando recursos. Um e-commerce vai gerenciar produtos, fornecedores, clientes, forma de pagamento. No caso vamos fazer uma livraria, vamos estar trabalhando com recursos onde cada recurso vai ter uma URI, eu vou ter livros, autores, editoras. Esse é o primeiro aspecto do REST.

[06:37] O REST foi criado pelo mesmo criador do protocolo HTTP, por isso que ele é tão utilizado e simplificado, as APIs construídas com esse modelo são mais simples de utilizar. Vamos trabalhar o tempo inteiro com recursos e cada recurso vai ter a sua URI, o seu identificador único, a forma com que vamos acessar os recursos.

[07:02] Como estamos trabalhando com recursos, no caso os livros, os autores e as editores, é importante, uma base do REST, é que essa comunicação entre quem faz a requisição e quem está devolvendo a resposta seja stateless, não guardamos o estado. Quando eu devolvo uma resposta, por exemplo, de um recurso, um livro com aquele valor naquele momento, aquela representação é sem estado, é aquela representação do momento. Se dois segundos depois alguém alterar o valor do livro no servidor a representação que veio para você continua, não guarda estado.

[07:50] A comunicação é stateless para que a internet aguente as milhões de requisições e respostas o tempo inteiro, como acontece, uma das bases do REST é ter comunicação stateless.

[08:04] Outros aspectos que podemos citar é que ela usa do próprio protocolo HTTP para manipular esses recursos. Porque o que vamos precisar fazer é cadastrar um livro, vamos precisar obter uma lista de livros, atualizar, excluir. Eu vou usar o próprio protocolo HTTP e os seus verbos para fazer essa manipulação, vou usar aquela mesma URI, que é o identificador único do recurso, e só vou trocar o verbo, vou usar o: GET, POST, PUT, DELETE, o que quer que eu queira fazer.

[08:41] Essas características do REST também simplificam a comunicação e como os sistemas vão conversar. O formato de representação do recurso também, como eu citei antes, basicamente poderia ser em qualquer formato, o REST é aberto diferente do SOAP que basicamente é XML obrigatoriamente. No REST não, pode ser um HTML, pode ser o próprio XML, pode ser JSON que é o mais comum. Você vai representar aquele recurso e trafegá-lo nas suas requisições e respostas.

[09:22] Essas são algumas das poucas características que vamos trabalhar ao longo do curso e eu vou reforçar. E agora vamos começar, vamos fazer a API para que alguém faça uma requisição para processamos os dados e devolvemos o que precisamos devolver.

Para complementar seus estudos, separamos alguns conteúdos extras com mais exemplos sobre o que tratamos nesse primeiro vídeo. Esse [Alura+](https://cursos.alura.com.br/extra/alura-mais/qual-e-a-diferenca-entre-back-end-e-front-end--c174) destaca mais diferenças e exemplos de backend e frontend. Não deixe de conferir!

O termo API (Application Programming Interface ou, em português, Interface de Programação de Aplicações) é amplamente utilizado quando se trata de desenvolvimento de software. É muito importante que você conheça esses principais termos, entenda-os melhor, para que possa sempre estar antenado às melhorias e boas práticas que são utilizados na comunicação dos sistemas distribuídos. Para aprofundar seus conhecimentos e entender mais a fundo o significado, confira esse [Alura+](https://cursos.alura.com.br/extra/alura-mais/o-que-e-uma-api--c697). Bons estudos!

] Conforme mencionado, é muito comum a API ser definida como uma API SOAP ou REST. O REST é um acrônimo para *Representational State Transfer*. Para entender mais a fundo o significado desse termo, confira esse [artigo](https://www.alura.com.br/artigos/rest-conceito-e-fundamentos).

REST(*Representational State Transfer*, que significa Transferência Representacional de Estado) **é um modelo de arquitetura e não uma linguagem ou tecnologia de programação, que fornece diretrizes para que os sistemas distribuídos se comuniquem diretamente** usando os princípios e protocolos existentes da Web sem a necessidade de [SOAP](https://pt.wikipedia.org/wiki/SOAP) ou outro protocolo sofisticado.

**A origem do termo REST**

Os conceitos do **REST** foram submetidos à tese de doutorado de [**Roy Fielding**](https://en.wikipedia.org/wiki/Roy_Fielding) nos anos 2000, onde o princípio fundamental é usar o protocolo HTTP para comunicação de dados.

A arquitetura REST é simples e fornece acesso aos recursos para que o cliente REST acesse e renderize os recursos no lado do cliente. No estilo REST, URI ou IDs globais ajudam a identificar cada recurso.

Esta arquitetura usa várias representações de recursos para representar seu tipo, como **XML**, **JSON**, **Texto**, **Imagens** e assim por diante.

Vale ressaltar que o REST não se limita a solicitações e respostas de registros. Também é possível inserir um novo registro ou deletar um já existente.

**Responsabilidades no REST**

Existe no REST um princípio chamado STATELESSNESS (sem estado), onde o servidor não precisa saber em qual estado o cliente está e vice-versa. Mas o que é um servidor e um cliente?

**Cliente:** é o componente solicitante de um serviço e envia solicitações para vários tipos de serviços ao servidor.

**Servidor:** É o componente que é o provedor de serviços e fornece continuamente serviços ao cliente conforme as solicitações

Nesta arquitetura ou modelo, cliente-servidor ajuda na separação de responsabilidades entre a interface do usuário e o armazenamento de dados. Ou seja, quando uma solicitação REST é realizada, o servidor envia uma representação dos estados que foram requeridos.

Não há limite superior no número de clientes que podem ser atendidos por um único servidor. Também não é obrigatório que o cliente e o servidor residam em sistemas separados.

A comunicação entre cliente e servidor ocorre através da troca de mensagens usando um padrão de solicitação-resposta. O cliente basicamente envia uma solicitação de serviço e o servidor retorna uma resposta.

**Requisições e comunicações**

O REST precisa que um cliente faça uma requisição para o servidor para enviar ou modificar dados. Um requisição consiste em:

* **Um verbo ou método HTTP**, que define que tipo de operação o servidor vai realizar;
* **Um header**, com o cabeçalho da requisição que passa informações sobre a requisição;
* **Um path (caminho ou rota)** para o servidor, como por exemplo <https://www.alura.com.br/artigos/golang-trabalhando-com-datas>;
* **Informação no corpo da requisição**, sendo esta informação opcional.

**Métodos HTTP**

Em aplicação REST, os métodos mais utilizados são:

* O **método GET** é o método mais comum, geralmente é usado para solicitar que um servidor envie um recurso;
* O **método POST** foi projetado para enviar dados de entrada para o servidor. Na prática, é frequentemente usado para suportar formulários HTML;
* O **método PUT** edita e atualiza documentos em um servidor;
* O **método DELETE** que como o próprio nome já diz, deleta certo dado ou coleção do servidor.

[Neste link](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods) você encontrará a lista completa de todos os métodos.

**Códigos de Respostas**

Cada resposta que a aplicação REST retorna, é enviado um código definindo o status da requisição. Por exemplo:

* 200 (OK), requisição atendida com sucesso;
* 201 (CREATED), objeto ou recurso criado com sucesso;
* 204 (NO CONTENT), objeto ou recurso deletado com sucesso;
* 400 (BAD REQUEST), ocorreu algum erro na requisição (podem existir inumeras causas);
* 404 (NOT FOUND), rota ou coleção não encontrada;
* 500 (INTERNAL SERVER ERROR), ocorreu algum erro no servidor.

Estes são os principais, porém [neste link](https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status) você encontrará a lista completa do código de cada requisição.

**Conclusão**

Neste artigo, apresentamos a origem, conceitos e os principais fundamentos do REST. [**Os cursos de programação da Alura**](https://www.alura.com.br/cursos-online-programacao) podem ajudar você a começar, aprender e se desenvolver em tecnologia.

[00:00] Vamos começar com a parte que mais gostamos que é codar e ver as coisas acontecendo. Eu estou usando uma versão do *Node* no momento da gravação do vídeo que é a 16.4, se você estiver usando uma versão anterior o ideal seria que atualizasse para que não tenhamos problema de compatibilidade, de você não conseguir fazer alguma coisa que eu esteja fazendo no vídeo.

[00:26] Se você estiver com uma versão mais atualizada está ótimo porque na hora de instalar as bibliotecas eu também vou versionar e passar para você exatamente qual a versão que foi usada para cada item que eu vou explicar.

[00:43] Digitando "-v" no *prompt* de comando, podemos ver que a versão do Node está 16.4 e a do NPM nesse momento é a 8.3.1. **NPM** é o gerenciador de pacotes, se você fez o curso que é pré-requisito já conhece esses conceitos do Node, no NPM, de como criar, inicializar um projeto Node, mas caso não tenha feito eu vou deixar link para todos esses materiais que são fundamentais para você ter um bom aproveitamento no curso. Vai ter link para a Alura Mais, para artigo e recomendo que assista, ou leia o material indicado.

[01:22] Eu vou criar uma pasta com mkdir alura-node e vou entrar nessa pasta cd alura-node e inicializar o projeto Node. Inicializamos com npm init, e vou usar o "-y" para que ele já responda aquelas perguntas padrão sem eu ter que fazer nada. C:\alura-node>npm init -y. Porque podemos sempre mudar os *scripts* e alterar o nosso arquivo "package.json" Não preciso ficar respondendo aquelas perguntas passo a passo.

[02:01] Feito isso vou abrir o VS Code para começar a trabalhar no meu projeto. Pelo terminal eu posso digitar C:\alura-node> code . que ele já abre o VS Code para mim. Observe que ele já está na minha pasta e já tem o arquivo "package.json" do lado esquerdo que tem o nome do projeto, a versão, qual é o arquivo que ele espera que seja o arquivo principal, o "index.js" para ser o ponto de entrada.

Código em package.json:

{

"name":"alura-node",

"version": "1.0.0",

"description": "",

"main": "index.js"

"scripts": {

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

},

"keywords": [],

"author": "",

"license": "ISC"

}COPIAR CÓDIGO

[02:33] E aí eu vou criar aqui um novo arquivo e vou chamá-lo de "server.js", não vai ser o "index.js". Vamos criar um novo arquivo que eu vou chamar de "server.js". Nesse primeiro momento eu vou criar um servidor local com o HTTP que é um módulo nativo do Node, ainda não vou mostrar o *Express*.

[02:58] Nesse primeiro momento também vamos usar o módulo nativo usando a expressão *require* ao invés de *import*. Eu também vou explicar sobre isso, quando estamos programando é comum distribuirmos os métodos, as funcionalidade em vários arquivos e para eu acessar uma funcionalidade que está em outro arquivo eu preciso importar e exportar isso para que fique visível, que fique disponível para ser usado.

[03:29] Tem uma forma de fazer isso que o *require* para importar um modo, uma funcionalidade que está de acordo com o CommonJS e tem o *import* que é mais atual que está de acordo com o ECMAScript 6. Nesse primeiro momento vou usar o *require*, mas a partir do momento que começarmos a usar o *Express* todo o nosso projeto já vai ser utilizado com o *import* e vou voltar a falar sobre isso.

[03:55] Eu quero criar um servidor para poder escutar isso em uma porta e fazer as minhas requisições, igual mostrei no vídeo anterior, digitamos o endereço no navegador e em algum lugar tem um servidor remoto que vai receber essa requisição, processar e devolver uma resposta para nós.

[04:20] No caso que você é o programador e está desenvolvendo, você tem que ter um servidor local, ele tem que estar rodando na sua máquina para que você teste esse processo de requisição e resposta. Você não tem um servidor remoto no momento, tem que criar um servidor local que é o primeiro passo que vamos fazer. Vamos criar um servidor local.

[04:44] Eu vou criar em "server.js" uma constante que vou chamar de HTTP e vou usar o *require* para importar o módulo nativo HTTP, const http = require("http"). Essa primeira linha é isso, estou criando uma constante e estou informando que quero utilizar o módulo nativo HTTP.

const http = require("http")COPIAR CÓDIGO

[05:11] Eu também preciso definir uma porta do meu computador que vai escutar essa requisição, preciso dizer quando for passar, lembra que eu comentei da URI para cada recurso? Quando eu for fazer o meu projeto de livros vou precisar digitar o endereço e um barra livros para obter os meus livros. O meu servidor local que vai estar rodando localmente na minha máquina, eu preciso dizer em que porta ele vai escutar essas requisições.

[05:46] Então, vou criar uma constante porta e definir em que porta ele vai estar fazendo isso. Vou criar uma constante chamada const port = 3000;. Depois vamos fazer um ajuste que é comum, criando uma variável de ambiente, referenciando uma variável de ambiente que vai definir a porta quando estiver rodando isso na Web, mas aqui no seu servidor local você pode definir uma porta. 3000, 4000, 8080, são algumas portas que geralmente usamos como padrão para servidor local.

[06:27] Feito isso, vou criar uma terceira constante com o servidor, que é onde vou criar o servidor e definir o que ele vai escutar, como que vai acontecer essa comunicação. Eu vou criar um const server e usar o módulo HTTP e usar um método que o HTTP que tem que chama-se *create server*, vou usá-lo para que ele possa criar o servidor, const server = http.createServer().

[07:04] Esse *server* geralmente vai ter parâmetros, vai ter que ter requisição e resposta. Vamos criar como primeiro parâmetro um req para representação requisição e o res para representar a resposta. E vamos ter uma função de *callback* que vamos chamar com *arrow function* para executar o que quero fazer quando acontecer essa requisição, const server = http.creatServer((req, res) => {.

const http = require("http")

const port = 3000;

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

})COPIAR CÓDIGO

[07:37] Colocamos a parte inicial do nosso servidor e a partir disso eu vou definir o que quero devolver quando alguém chamar o meu servidor. A princípio vou devolver uma página com um texto, que a minha resposta vai ter um cabeçalho que vou chamar de *writeHead* e dizer para ele que vai devolver um *status*. Ainda vamos falar um pouco mais de *status* , mas isso tem no curso de HTTP falando sobre portas, sobre *status*, tudo o que estou citando tem melhor definido no curso.

[08:19] Vamos dizer que ele vai devolver um *status* 200, que é um *status* de "ok" e eu vou colocar qual é o tipo de conteúdo que estou mandando. 5res.writeHead(200, {}). Porque quando estamos devolvendo a representação de um recurso ela pode ser em JSON, pode ser texto, pode ser HTML, eu vou dizer que quero devolver um texto. O meu *content type* vai ser texto, vou colocar que res.writeHead(200, {'Content-Type': 'text/plain'})

const http = require("http")

const port = 3000;

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

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

})COPIAR CÓDIGO

[09:04] Quando estivermos usando o Postman vou mostrar melhor esses *content types* para ver o vários tipos e como configuramos, é só para termos um primeiro exemplo, basta você digitar dessa forma.

[09:18] Depois disso tenho que finalizar a minha resposta mandando o que quero mandar. Eu vou fazer um res.end mandando o texto que quero enviar para o navegador. Vou mandar, por exemplo, res.end('Curso de Node'), e está o servidor e o que ele vai fazer.

const http = require("http")

const port = 3000;

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

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

res.end('Curso de Node');

})COPIAR CÓDIGO

[09:44] Para finalizar a última coisa que precisamos é definir a porta que ele vai escutar, fazer esse link entre o servidor que criei e a porta que ele vai escutar a requisição e mandar a resposta. E posso dar uma mensagem, por exemplo, um *console* log dizendo que ele funcionou e que ele está operando em uma determinada porta.

[10:07] Vamos fazer isso com o server.listen para dizer que ele vai escutar a requisição e dizer em qual porta que ele vai escutar, essa informação está na minha variável port. Vou colocar server.listen(port, => { e um console.log para ele logar que está escutando requisição na porta 3000.

[10:37] Nesse *console* log irei usar um *string template* para podermos concatenar a *string* variável e vou digitar console.log(Servidor escutando em http://localhost: e inserir o cifrão e a variável port para ele pegar a porta que defini. console.log(Servidor escutando em [http://localhost:${port}`](http://localhost:$%7Bport%7D%60/))

const http = require("http")

const port = 3000;

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

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

res.end('Curso de Node');

})

server.listen(port, () => {

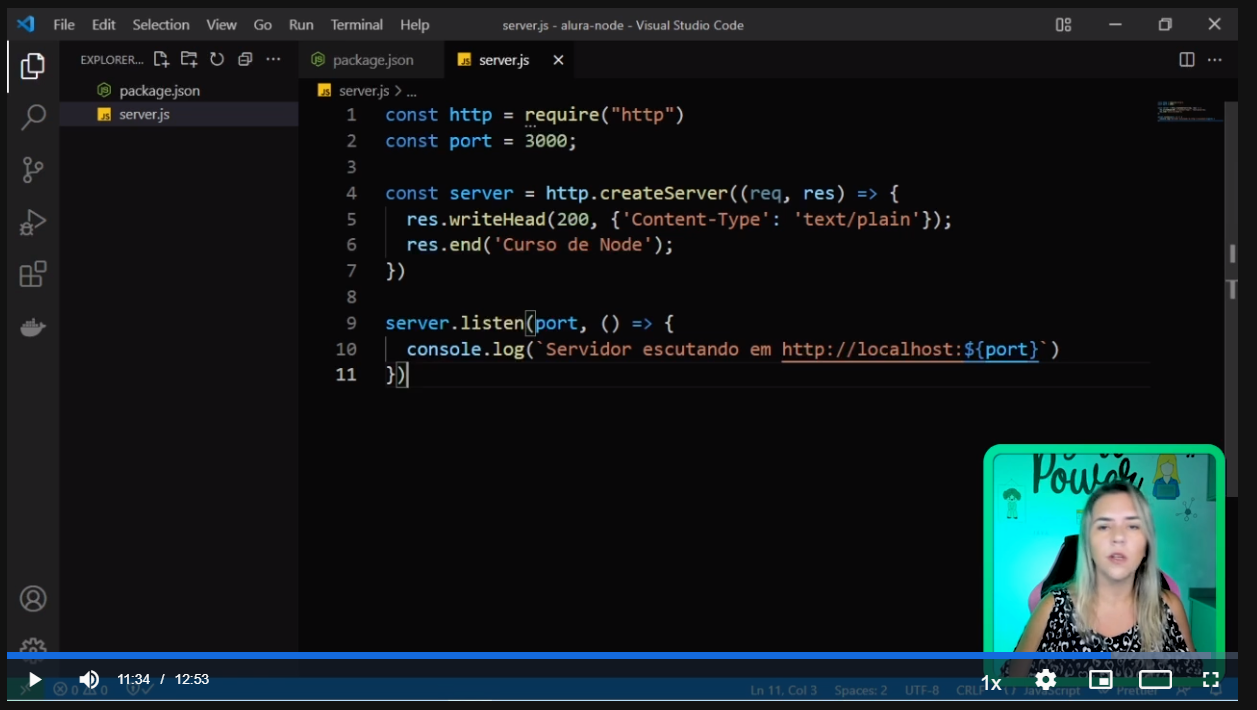
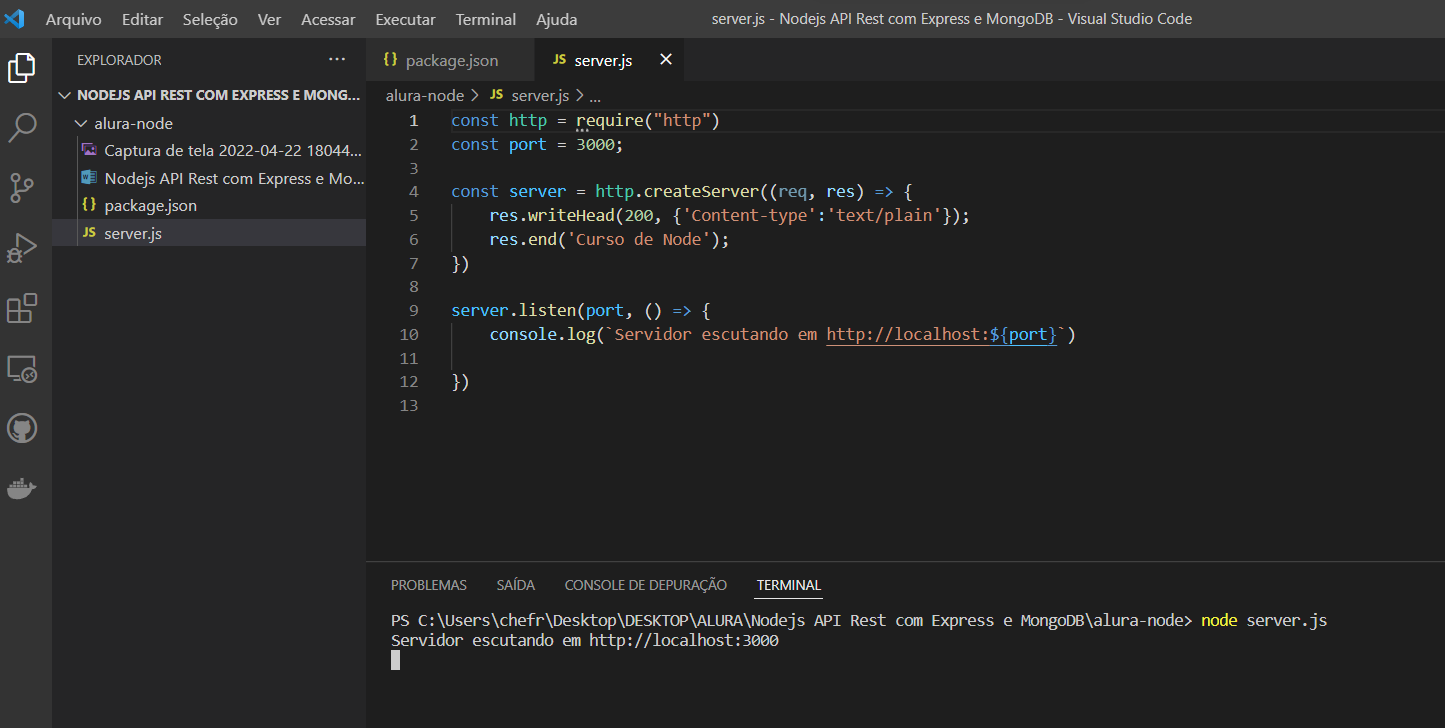
console.log(`Servidor escutando em http://localhost:${port}`)

})COPIAR CÓDIGO

[11:17] Basicamente esse código está pronto, com isso já consigo visualizar no navegador uma requisição, quando eu digitar localhost:3000 ele já tem que me responder com o texto escrito Curso de Node. Só falta colocarmos isso para funcionar, vou abrir o próprio terminal do VS Code, "Terminal > New Terminal" e para executarmos e ver esse servidor rodando vamos digitar: node server.js e ele já deu a mensagem: "Servidor escutando em htp://localhost:3000”. Ele tem, inclusive, um link, se passarmos a mão tem o link, eu vou seguir o link e ele já vai mostrar para mim: "Curso de Node".

[12:12] Basicamente é o que aconteceu, definimos uma porta, criamos um servidor local, fizemos uma requisição quando chamamos o “localhost:3000” e ele devolve uma resposta. Tudo se consolidando com o que foi visto no vídeo anterior, requisição, resposta, servidor local. Adiante vamos só evoluindo passo a passo para ter a API robusta com *Express*, com Mongo cada vez mais sofisticada e de acordo com o mercado.

[12:48] Eu espero vocês na próxima aula para continuarmos.



Ao instalar o Node, já temos à nossa disposição alguns módulos embarcados para começar a desenvolver nossas bibliotecas e aplicações. Para conhecer mais sobre esses recursos nativos, acesse a documentação oficial referente às APIs nesse [link](https://nodejs.org/api/).

[00:00] Vamos dar continuidade. Anteriormente fizemos somente uma mensagem para ser exibida como resposta no navegador quando fazíamos uma requisição para “localhost:3000” só que agora eu quero realmente começar a colocar o conceito de rotas, ter um “rota/livros”, outra “rota/autores”, vamos implementar isso no VS Code.

[00:25] Vou parar o serviço em server.js e começar a alterar, fazer o que preciso para mostrar a rota do jeito que queremos. Minimamente, também só como exemplo, vamos evoluir essas rotas depois. Eu vou criar um objeto chamado "rotas" e o objeto tem a característica de ser chave valor, vou ter, por exemplo, se ele chamar a rota básica sem nada vou exibir aquela mensagem, const rota = { '/': 'Curso de Node'. Se ele chamar, por exemplo, vou digitar uma vírgula e uma segunda rota. Se ele chamar '/livros': 'Entrei na pag de livros', E se chamar '/autores': 'Listagem de autores'

const http = require("http")

const port = 3000;

const rotas = {

'/': 'Curso de Node',

'/livros': 'Entrei na pag de livros',

'/autores': 'Listagem de autores',

}COPIAR CÓDIGO

[01:40] Define três tipos de rotas e a minha resposta a partir de agora não pode ser fixa, esse texto em res.end é "Curso de Node", eu preciso trocar isso para que ele consiga obter o valor que está no meu objeto rotas. Para isso vou substituir isso por rotas e entre colchetes vou passar res.end(rotas[req.url]);. O que vier no endereço, se no endereço vier “/livros” ele sabe que é para devolver a mensagem "Entrei na pag de livros". Se vier “\autores” ele tem que identificar que é para devolver a mensagem "Listagem de autores".

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

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

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

})COPIAR CÓDIGO

[02:28] Com esse ajuste possivelmente já vamos ver isso funcionando. Eu vou subir o servidor novamente com o comando node server.js no terminal e vamos voltar no navegador e ver sem nada e voltou "Curso de Node" com a URL "localhost.3000". Vamos incluir no final da URL “/livros”, "Entrei na pag de livros", funcionou. Agora vou o “/autores” e entrou na página "Listagem de autores".

[03:04] Tudo fluindo, por enquanto. Só que observe que se eu quisesse uma nova rota chamada "Editora", se eu fizer isso agora em server.js e nesse momento alterar o objeto “/editora” e digitar '/editora': 'Pag de editora' eu não consigo visualizar essa alteração no momento no navegador, se eu digitar no final da URL do site /editora não vai funcionar.

[03:42] Então, nós vamos instalar uma biblioteca chamada Nodemon que vai fazer um live reload. Cada vez que fizermos uma alteração e salvar no arquivo ele já vai identificar que é para fazer o reload e vamos conseguir visualizar isso no navegador. Eu vou parar o servidor e fazer essa instalação juntos.

[04:09] Eu vou no browser e fechar essa página, vou parar o servidor com o "Ctrl + C" e para fazer essa instalação já vou versionar com a última versão do Node que é a 2.0.15. Vou digitar no terminal npm install nodemon@2.0.15 e um -D porque essa é uma dependência de desenvolvimento. Essa dependência não precisa ir para a produção, é uma dependência que facilita para nós quando estamos desenvolvendo, não precisamos que ela vá para a produção por isso coloquei esse "-D" maiúsculo para indicar que é uma dependência de desenvolvimento.

[05:05] Vamos fazer a instalação, vai começar a fazer o download, baixar no projeto. E vamos observar que vai ter só uma pequena mudança, a pasta "node\_modules" já foi criada no lado esquerdo da tela. Se você também já tem uma familiaridade, sabe que cada nova biblioteca que instalamos no terminal cria a pasta "node\_modules" que vai ficando ao longo do tempo muito grande porque uma dependência usa outras dependências que usam outras dependências e quando fazemos a instalação vem todo mundo para a pasta.

[05:45] Se procurarmos o Nodemon ele vai ter também o "package.json" dele cheio de dependências que ele precisa. Tudo o que essa dependência precisa vai ser baixada também, aos poucos essa pasta vai ficando muito grande e é por isso que sempre orientamos quando você está fazendo o projeto e vai subir esse projeto para o Git, criar o arquivo gitignore que eu vou fazer daqui a pouco.

[06:14] Para podermos usar o Nodemon vamos alterar o "package.json". No "package.json" onde temos script vou criar um novo script para que quando eu rode esse script já tenha essa facilidade dele rodar o comando "Nodemon Server.js". Vamos fazer isso. Já tem um script padrão como teste que não estamos usando, vou criar um novo chamado "dev" e nesse script dev eu vou colocar "dev": "nodemon server.js", e uma vírgula para que consigamos executar esse script.

Código em package.json:

{

"name": "alura-node",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"dev": "nodemon server.js",

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

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"nodemon": "^2.0.15"

}

}COPIAR CÓDIGO

[06:59] No terminal ao invés de node server.js como vínhamos fazendo, vou digitar npm run que é para rodar o script e referenciar esse script que acabei de criar, o dev: npm run dev. Vai executar, observe que já iniciou o Nodemon e colocou que ele está visualizando os arquivos com as extensões .js, .mjse .json. Cada vez que alterar algum arquivo dessas extensões e salvar ele vai reiniciar o servidor.

[07:48] Vamos abrir o "Local Server" e vou ter a minha página, vou ter a “/autores”, por exemplo, e se eu incluir uma nova página agora no server.js, por exemplo, '/sobre': 'Info sobre o projeto', a partir do momento que salvar quando voltarmos no navegador já consigo entrar no “/sobre” que ele já vai ter recarregado, o "Info sobre o projeto" já apareceu para nós.

Código em server.js

const http = require("http")

const port = 3000;

const rotas = {

'/': 'Curso de Node',

'/livros': 'Entrei na pag de livros',

'/autores': 'Listagem de autores',

'/editora': 'Pag de editora',

'/sobre': 'Info sobre o projeto'

}

//código omitidoCOPIAR CÓDIGO

[08:35] O Nodemon facilita muito a vida quando estamos desenvolvendo, não precisa ficar toda hora parando e subindo o servidor novamente. Vale muito a pena usar essa biblioteca.

[08:48] Para finalizar porque na próxima aula já vamos começar com o Express, a minha recomendação é criar o “gitignore” de uma vez, eu vou criar o novo arquivo .gitignore e toda vez que fizer a integração com o meu GitHub, subir o meu projeto, vou subir desconsiderando a pasta "node\_modules". É muito importante salvar, ter essa opção de “gitignore” porque quando você versionar e subir o seu código não vai subir com todas essas dependências.

Código em gitignore:

node\_modulesCOPIAR CÓDIGO

[09:28] Espero vocês na próxima aula e já vamos começar com o Express.

**11 Para saber mais: o arquivo package.json**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103618/next)

Como vimos ao longo dessa aula, os projetos em Node.js geralmente são chamados de módulos. Esse termo surgiu do conceito de que o JavaScript trabalha com uma arquitetura modular. E quando criamos um projeto, ou seja, um novo módulo, é comum termos associado a ele um arquivo de configuração, que irá descrever os atributos-chaves, dependências, nome, versão, dentre outras informações fundamentais ao projeto. Estamos falando do arquivo package.json cuja estrutura e conteúdo é similar ao bloco de código abaixo:

{

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

"version": "1.0.0",

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

"main": "primeiro-projeto.js",

"scripts": {

"start": "node primeiro-projeto.js",

"test": "jest"

},

"author": "Jacqueline Oliveira",

"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 você define o nome pelo qual seu módulo será chamado;
* *version*: cada vez que uma atualização do módulo é lançada é atribuído um conjunto de 3 números. os módulos trabalham com três níveis de versionamento seguindo um padrão chamado SemVer (*Semantic Versioning*), ou seja, versionamento semântico, onde 3 números separados por ponto correspondem, respectivamente, aos atributos *Major*, *Minor* e *Patch*. *Patch* está relacionado a uma alteração que não quebra uma funcionalidade anterior e nem adiciona novas. Geralmente é usado para liberar alguma correção de bug. *Minor* é quando adicionamos uma nova funcionalidade, sem quebrar funcionalidades anteriores. *Major* é quando pode ocorrer uma quebra de compatibilidade. Por isso, é importante indicar a versão de forma adequada. Você pode ler mais sobre versionamento semântico [aqui](https://semver.org/lang/pt-BR/);
* *description*: define o que será este módulo. 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. Nesse [link](https://docs.npmjs.com/cli/v8/using-npm/scripts) é possível ter acesso a algumas informações sobre os mesmos;
* *keywords*: é um array de palavras-chave sobre o projeto;
* *author*: são dados de autoria, pode conter nome, e-mail e site;
* *license*: é a licença escolhida para o projeto;
* *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 num 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](https://docs.npmjs.com/cli/v8/configuring-npm/package-json) do NPM.

**Para saber mais: gerenciadores de pacote**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103619/next)

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 nada mais são do que 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. E um pacote (ou módulo) - aqui podemos pensar em um pacote mesmo - que é como chamamos o conjunto do código que determinada lib ou framework utiliza para executar.

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, isso significa que 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 e ficando disponíveis para todos os projetos em seu computador, 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.

**Para saber mais: atalhos do NPM**

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103620/next)

Ao trabalhar com o Node.js, frequentemente iremos instalar e utilizar novas bibliotecas ou frameworks, bem como publicar as nossas próprias. Com isso, utilizamos algum gerenciador de pacotes, como o NPM ou o YARN. Visto que o NPM já é instalado juntamente com o Node.js, destacamos aqui alguns dos principais comandos e atalhos para facilitar o uso do mesmo:

| **Comando** | **Atalho** | **Uso** |
| --- | --- | --- |
| install | i | Instala um novo módulo |
| list | ls | Lista os módulos instalados |
| test | t | Executa o script de teste especificado no package.json |
| --global | -g | Usado como um sinalizador para instalar um pacote globalmente (não só na pasta do projeto |
| --save | -S | Salva o módulo ao instalar |
| --save-exact | -E | Salva o módulo ao instalar, porém na versão exata mencionada |
| --save-dev | -D | Salva o módulo ao instalar, porém como uma dependência de desenvolvimento |

Outros comandos muito comuns:

| **Comando** | **Descrição** |
| --- | --- |
| npm home | Abre a página do projeto |
| npm repo | Abre o repositório de código do projeto |
| npm audit | Executa uma auditoria no projeto para identificar se existe alguma dependência com vulnerabilidade conhecida |
| npm update | Atualiza projetos para a última versão respeitando o package.json |
| npm outdated | Retorna a lista de dependências desatualizadas mostrando a versão mais recente |
| npm adduser | Adiciona um novo usuario no [https://registry.npmjs.org](https://registry.npmjs.org/) para permitir a publicação de pacotes |
| npm publish | Faz a publicação do pacote no NPM |

Lembrando que não é necessário decorar todos estes comandos, sempre é possível utilizar o comando:

npm help <nome do comando>COPIAR CÓDIGO

para obter mais informações e detalhes dentro do próprio terminal.

**Nessa aula, você aprendeu:**

* Como o backend entrega os dados para o frontend;
* O que é uma API REST, que é uma interface que vai fornecer dados e recursos baseados em requisições e respostas HTTP;
* Iniciar um novo projeto Node através do comando npm init;
* Personalizar o arquivo de configuração package.json com scripts e as informações individualizadas do projeto;
* Utilizar um gerenciador de pacotes, como o NPM, para instalar novos módulos que serão utilizados no projeto;
* Como está estruturada a pasta node\_modules e a importância de incluí-la no arquivo .gitignore;
* Construir um servidor local utilizando o módulo http do Node.js;
* Realizar a criação de novas rotas e qual o objetivo das mesmas;
* Instalação e utilização do nodemon para não ter que reiniciar o servidor a cada alteração.

## Transcrição

[00:00] Na aula anterior nós criamos um servidor local utilizando um módulo nativo no Node que é o HTTP. E foi uma coisa bem simples só para demonstrar só que para construir uma aplicação Web mesmo robusta em que vamos fazer uso dos verbos HTTP e para chamar um recurso eu vou usar o Get, o Postgre, isso vai ter um comportamento diferenciado, fica ruim fazer com esse padrão do HTTP. Não tem porque reinventarmos rodas se eu tenho um framework que já facilita o meu trabalho. É justamente aí que vai entrar o Express.

[00:38] O Express é um framework que é considerado rápido, minimalista e não opinativo. Esse não opinativo, geralmente um framework já tem um padrão de código, de escrita, uma forma que se considera o apropriado para você desenvolver a aplicação e com o Express isso não acontece, ele deixa você criar e usar seus mideals, usar padrão MVR de Model View Roots ou MVC de Model View Controler, do jeito que você quiser. Ele dá essa liberdade para o desenvolvedor e não amarra você a escrever sempre em um determinado padrão.

[01:20] Vamos usar um padrão, mas pode ser que você veja alguém usando diferente em outro lugar porque ele dá essa liberdade. E ele é a base para vários outros frameworks, é um dos mais utilizados no mercado, está sempre liberando versão, sempre com a documentação atualizada. Vamos usar essa última versão lançada que é 4.17.3, é a última versão no momento de gravação do vídeo.

[01:50] No site do [NPM neste link](https://www.npmjs.com/) temos a possibilidade de consultar também a última versão basta você digitar em "search" o que você vai utilizar, o que você vai instalar no seu VSCode, vou digitar "Express" e ele já mostra para a versão 4.17.3, que é a que vamos usar.

[02:12] Da mesma forma se digitarmos Nodemon que usamos no outro vídeo, está a versão 2.0.15 que é a última, justamente a que instalamos. Nesse site você tem a documentação oficial do Express que vale a pena começar em "Getting started", tem como que é feita a instalação, como pode fazer pode fazer um reload tem um exemplo básico, mas vamos fazer passo a passo nesse vídeo.

[02:38] Vamos lá no VSCode fazer a instalação do Express e começar a utilizar. No VSCode eu vou fazer a instalação do mesmo padrão que eu fiz o do Node, indicando a versão. Vou digitar no terminal do VS Code npm install express@4.17.3, essa é a última versão. E ele vai fazer a instalação e enquanto isso vamos preparando o nosso projeto para as boas práticas, para o padrão de pasta que utilizamos vamos criar uma pasta chamada "SRC", a pasta source onde dentro vamos criar um novo arquivo.

[03:27] Primeiro a nova pasta chamada "src" e eu vou criar um novo arquivo dentro desta pasta chamado "app.js". Vamos dar início a desenvolver a aplicação e fazer a API agora usando o Express. Para utilizar o Express a primeira coisa que fazemos é importar, eu preciso importar, vou fazer o import express from "express", ele já está, já preencheu para nós.

Código em app.js:

import express from "express";COPIAR CÓDIGO

[04:08] E vou criar uma instância do Express para eu poder utilizar e fazer o mesmo procedimento que eu fiz para o HTTP, subir um servidor e começar a ver as rotas acontecendo. Vamos fazer um const app = e vou dizer que esse const app recebe uma instância do Express `const app = express();´

import express from "express";

const app = express();COPIAR CÓDIGO

[04:29] E para podermos simular o projeto de livraria eu vou criar uma constante de livros, para que possamos visualizar alguns livros. Vou criar uma constante livros que vai ser um array com dois livros, por exemplo, const livros = [ e o livro vai ter um "Id1" e o título, por exemplo, vai ser "Senhor dos Anéis" {id: 1, "titulo": "Senhor dos Aneis"} e um segundo livro com "Id2" cujo o título vai ser "O Hobbit" id: 2, "titulo": "O Hobbit"}.

import express from "express";

const app = express();

const livros = [

{id: 1, "titulo": "Senhor dos Aneis"},

{id: 2, "titulo": "O Hobbit"}

]COPIAR CÓDIGO

[05:19] Só para começarmos a trabalhar com a rota visualizando como que é a rota base e como que é uma rota "/livros", por exemplo. Diferente do que foi feito usando o módulo nativo HTTP. Vamos começar a programar o get porque quando eu vou no browser e digite o endereço ele está fazendo uma requisição get por trás. É o que vamos fazer agora.

[05:51] Vou fazer um app.get para eu passar os atributos, o que eu quero que aconteça quando eu fizer uma requisição para uma determinada rota. A primeira que eu vou fazer é da rota vazia, eu não passar nada só passar aquele “localhost:3000” que é a porta que o nosso servidor escuta. Se isso acontecer eu vou ter o req e o res que são os parâmetros de requisição e resposta e vou abrir aquela arrow function para programar o que vai acontecer se a rota for somente “localhost:3000”, app.get('/', (req, res) => {

[06:36] Como vamos devolver uma resposta? Com o HTTP usávamos o res.end e nesse caso vamos fazer o res.status e para simplificar já vou mandando 200 de um status ok e vou enviar um send com a mensagem que eu quero que ele devolva para o navegador res.status(200).send('Curso de Node');.

import express from "express";

const app = express();

const livros = [

{id: 1, "titulo": "Senhor dos Aneis"},

{id: 2, "titulo": "O Hobbit"}

]

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

res.status(200).send('Curso de Node');

}) COPIAR CÓDIGO

[07:05] E eu já quero criar também a rota para os livros, já que eu criei uma constante com livros para me devolver livros, então quero devolver também a rota dos livros. Vou incluir mais um app.get('/livros') que vai acontecer a mesma coisa, vou ter o conceito de requisição, de resposta, vou abrir uma arrow function ('=> {}') e dentro vou pedir para ele me devolver um JSON com esse meu conteúdo app.get('/livros', (req, res) => { res.status(200).json(livros)})

import express from "express";

const app = express();

const livros = [

{id: 1, "titulo": "Senhor dos Aneis"},

{id: 2, "titulo": "O Hobbit"}

]

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

res.status(200).send('Curso de Node');

})

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

res.status(200).json(livros)

})COPIAR CÓDIGO

[08:10] Como eu falei, para que outro arquivo utilize esse arquivo eu tenho que importar e exportar,nesse caso eu importei o Express para fazer o uso do Express e aí preciso exportar esse arquivo para que eu possa fazer uso dele no Server, porque o Server é que vai poder fazer, vai ter o listen e vai ficar ouvindo a requisição.

[08:40] Eu vou fazer um export default para ele exportar tudo o que eu tenho nesse arquivo para que outro arquivo possa fazer uso, export default app. A princípio o arquivo app vai estar desse jeito, importando Express, criando uma instância do Express, vai ter uma constante de livros e dois métodos Get, um sem nada e outro com "/livros" para podermos ver esse livros aparecendo.

import express from "express";

const app = express();

const livros = [

{id: 1, "titulo": "Senhor dos Aneis"},

{id: 2, "titulo": "O Hobbit"}

]

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

res.status(200).send('Curso de Node');

})

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

res.status(200).json(livros)

})

export default appCOPIAR CÓDIGO

[09:25] Agora para ver isso funcionando igual vimos lá com HTTP eu vou alterar o arquivo server.js, no arquivo Server vou apagar essas rotas, vou apagar esse Server todo, só vou deixar a porta e esse listen que a implementação vai ficar igual. Agora não vou usar HTTP, vou usar o Express, só que eu já declarei o meu Express naquele arquivo app.js.

[09:58] Basta eu importar o app.js e começar a utilizá-lo. Vou fazer um import app from e indicar o caminho, ele vai está no './src/app.js'. Quando usamos o import temos que colocar a extensão, às vezes mesmo você clicando na extensão ele tira por isso você tem que colocar meio manualmente porque senão corre o risco de não funcionar, ele não localizar e ficar no console dando mensagem de que não achou o app.js.

Código em server.js:

import app from './src/app.js'

const port = 3000;

server.listen(port, () => {

console.log(`Servidor escutando em http://localhost:${port}`)

})COPIAR CÓDIGO

[10:37] Na porta nós também vamos uma alteração que eu tinha dito porque muitas vezes nós vamos subir essa aplicação para Heroku ou para um outro servidor que não é um servidor local e a porta que ele vai está escutando vai ser diferente, nós vamos fazer um condicional.

[10:58] Ou a porta vai ser do cons port = process.env.port; que é a constante que vai identificar a porta que está sendo escutada a requisição que o servidor subiu ou na 3000, cons port = process.end.PORT | | 3000; Já vamos deixar com essa boa prática de diferenciar ou a porta no ambiente de produção ou a porta fixa que é a porta 3000.

import app from './src/app.js'

const port = process.env.PORT || 3000;

server.listen(port, () => {

console.log(`Servidor escutando em http://localhost:${port}`)

})COPIAR CÓDIGO

[11:28] Feito isso eu vou alterar agora ao invés de server.listen vai ser o app.listen(port, () => {. Com essa pequena alteração no Server já tem que funcionar aquela rota que programamos no app.js. Eu vou salvar, vou abrir o meu terminal e vou rodar a aplicação para vermos.

import app from './src/app.js'

const port = process.env.PORT || 3000;

app.listen(port, () => {

console.log(`Servidor escutando em http://localhost:${port}`)

})COPIAR CÓDIGO

[11:50] Seguindo a aula anterior, npm run dev que ele vai subir com o Nodeman, se alterarmos ele vai dizer o que houve. Ele está informando: "não consigo importar do server.app.js", vamos ver o que aconteceu. Tem um detalhe que ele nos mostra, quando você vai trabalhar com o import, o export default lá no package JSON temos que incluir uma linha chamada type e module, para dizer que você está trabalhando com importação e exportação module.

[12:30] Eu vou copiar justamente esse trecho do erro e vou inserir no meu package.json. Pode ser em qualquer lugar, depois desse main, por exemplo, "main": "index.js", "type": "module". Eu vou selecionar "Enter", colar e salvar, provavelmente já vai subir o meu servidor.

Código em package.json:

{

"name": "alura-node",

"version": "1.0.0",

"description": "",

"main": "index.js",

"type": "module",

"scripts": {

"dev": "nodemon server.js",

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

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"nodemon": "^2.0.15"

},

"dependencies": {

"express": "^4.17.3"

}

}COPIAR CÓDIGO

[13:10] Agora foi, funcionou e está rodando no “[http://localhost:3000”](http://localhost:3000%E2%80%9D/) onde estava antes. Se analisarmos no navegador, o localhost:3000 mostra o que já mostrava "Curso de Node", se eu mostrar agora “[http://localhost:3000/livros”](http://localhost:3000/livros%E2%80%9D) ele vai mostrar em formato de JSON, ele já mostrou um array com os dois livros que eu cadastrei.

[13:48] Ele fica bem mais enxuto, se você observar no server.js fica com uma funcionalidade definida só para iniciar o servidor e ficar escutando requisição. Já a app tem as rotas e com o tempo vamos trabalhar com as rotas, com as requisições, fazendo o nosso crud, mas conforme a evolução vamos ver que esses arquivos ficarão mais limpos ainda, com o código mais limpo ainda, separando mesmo a responsabilidade em cada pacote, em cada arquivo. Vamos evoluir a nossa API a cada aula para ela ficar cada vez mais com uma cara profissional.

**Para saber mais: o que é framework?**

Nessa aula começamos a ver a importância de se utilizar um framework para poder ganhar agilidade no desenvolvimento e evitar retrabalho escrevendo código ou fazendo melhorias já implementadas. Para saber mais sobre frameworks em geral, confira esse [artigo](https://www.alura.com.br/artigos/framework-o-que-e-pra-que-serve-essa-ferramenta) que explica em detalhes os prós e contras e exemplifica alguns frameworks por linguagem.

[00:00] Vamos trabalhar com o conceito de livraria, vamos supor que queremos fazer uma API para a livraria da Alura. Seguindo o que foi falado do *REST*, de construir uma API com boas práticas e seguindo o padrão *REST* vamos ter uma URI, o identificador único para livros, vai rodar em um servidor local que é o *localhost* na porta 3000 é preciso usar os métodos HTTP para que eu possa fazer as operações que quero.

[00:37] Geralmente precisamos fazer o *CRUD* (*Create, Read, Update, Delete*). O *create* é a parte de criar um recurso, fazemos isso através do método *post*, você vai fazer um *post* para API de livros e vai conseguir cadastrar um novo livro. O segundo que é o *read* é o que vamos fazer com o método *Get*, posso ou buscar todos os livros cadastrados ou buscar um livro específico passando ainda um “/id” indicando o Id desse livro.

[01:15] O *update* que vamos fazer com o *put* para atualizar um livro e, por fim, *delete* que vamos usar para excluir um livro. É isso que vamos fazer. No vídeo anterior iniciamos com o *get*, já buscamos os livros, só que ainda não implementamos a busca por Id e nem fomos tão específicos. Vamos começar agora e vamos continuar a fazer os métodos.

[01:47] Vamos começar com a criação, porque para testarmos a visualização, a exclusão acho que o ponto mais importante é partirmos da criação, de conseguir pegar aquele array com dois livros e acrescentar um livro. Vamos para o nosso VS Code e vamos fazer esse passo a passo.

[02:05] Vamos abrir o VS Code, em app.js no momento ele já está rodando e tenho só um *get* livros que já me devolveram esse livro. O que queremos fazer agora é pegar esse array com dois livros e incluir mais um. Embaixo desse *get* vou fazer mais um método que vai ser o app.post() e no *post* eu também preciso dizer a URI, eu vou repeti-la passando livros e vou ter o mesmo conceito de requisição e resposta e o que eu quero fazer com essa requisição e resposta, app.post('/livros', (req, res) => {

[02:53] Para eu fazer um *post* preciso mandar uma representação desse recurso. No navegador quando digitamos a rota só conseguimos listar, eu não consigo mandar alguma coisa. Vamos usar o *Postman* e vamos através dele conseguir mandar uma representação do livro em formato JSON.

[03:17] Antes de eu abrir o *Postman* eu vou fazer o código para entendermos e depois vamos lá para testar. O que vier no corpo da minha requisição preciso acrescentar na minha constante livros, o que eu vou usar para fazer isso é um método chamado *push* que é para eu incluir um novo item em um array.

[03:43] Eu vou colocar livros.push e no meu array eu vou querer colocar o que vier no corpo da requisição, vamos colocar entre parênteses o livros.post(req.body) que é o que está vindo no corpo da requisição, é o que vamos fazer no *Postman*.

[04:04] E depois que eu fizer isso vou devolver uma resposta para confirmar que o livro foi cadastrado com sucesso. Vou fazer um res.status(201), para criação geralmente o *status* é 201 que dizemos que está *"ok"*, que foi criado, vamos colocar um link também para o material complementar falando de *status*, o que significa cada *status*, os intervalos de resposta, "", o que é sucesso, o que é falha, o que é erro do servidor, o que é erro de requisição. Eu vou deixar esse material complementar.

[04:43] Só que eu vou digitar um 201 e vou mandar uma mensagem dizendo que o livro foi cadastrado com sucesso, res.status(201).send('Livro foi cadastrado com sucesso') })´. A princípio isso já faria a minha rotina funcionar, meu método funcionar. Eu preciso testar isso, para testar isso vamos usar o *Postman*, vamos abrir o *Postman* e começar a fazer os testes.

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

livros.push(req.body);

res.status(201).send('Livro foi cadastrado com sucesso')

})COPIAR CÓDIGO

[05:13] Existe a versão do *Postman* para web, mas estou usando uma versão *Desktop*, se você quiser tenho a indicação também de onde fazer o download e como utilizar o *Postman*, o link complementar.

[05:28] Vou clicar neste sinal de mais do lado esquerdo para fazer uma nova requisição, a primeira requisição que vou fazer é aquele próprio *get* para garantir que está tudo funcionando. Tem outras ferramentas também que você pode usar para fazer esse teste, algumas pessoas preferem o *Insomnia*. No curso vou usar o *Postman* que é o mais utilizado geralmente, vou inserir http://localhost:3000 no campo.

[06:01] Quando faço essa requisição ele já me volta "Curso de Node" e mostra o *status*, voltou com o *status* "ok" porque foi justamente o *status* que nós programamos. Se incluir um /livros na minha requisição, no meu *endpoint*, na minha URI ele vai voltar já em formato JSON com os dois livros que eu deixei previamente cadastrado, "O senhor dos anéis" e "O Hobbit". E também voltou aqui o *status* 200.

[06:36] Agora precisamos incluir um novo livro, para isso vou usar o *post*, então no campo ao invés de *get* vou selecionar o *post*. O *endpoint* é o mesmo e vou precisar passar o corpo da requisição, eu vou ir em “Body” selecionar a opção “raw” e marcar que vou mandar um JSON, estou mandando um JSON para que você acrescente no array e digitar como quero mandar o JSON.

[07:15] Eu vou mandar exatamente com os mesmos campos que eu tenho no meu objeto, "Id" e "título". Vou fazer o "id":e vou colocar, por exemplo, ele é um número inteiro vou colocar o "id": 3, e o título do livro vai ser "titulo": "O Silmarilion" e vou clicar no botão “Send” do lado direito da tela para enviar esse livro para que seja cadastrado. E ele me retornou a mensagem: "Livro foi cadastrado com sucesso".

{

"id": 3,

"título": "O Silmarillion"

}COPIAR CÓDIGO

[07:52] E para conferir se, de fato, cadastrou e incluiu o livro no array: vou alterar para *get* novamente e confirmar se tenho o "Id3" cadastrado, ficou *null*. Ficou *null* porque alguma coisa ele não conseguiu interpretar, fiz a requisição, mas ele não conseguiu compreender. Provavelmente no formato do JSON ele não está preparado para ler, vamos precisar usar um recurso para identificar e corrigir esse problema.

[08:33] Vamos voltar para o VS Code para vermos isso acontecendo. Ao abrir o VS Code em app.js para fazer com que o *App* reconheça o que enviamos pelo *Postman*, preciso fazer logo abaixo da minha instância do Express um app.use e dizer que é para usar o app.use(Express.json()) que é um recurso do *Express* que vai conseguir fazer interpretar o que está chegando via *post* ou via *put* e transformar aquilo em um objeto para eu poder armazenar, visualizar e manipular.

import express from "express";

const app = express();

app.use(express.json())

//código omitidoCOPIAR CÓDIGO

[09:25] Vamos voltar agora no *Postman* e confirmar se com essa linha incluída se ele vai passar a reconhecer o livro que cadastramos. Vamos voltar aqui no *Postman*, vou fazer o *get*, ele vai ter só os dois livros porque estamos trabalhando com o *live reload*. A partir do momento que fui lá e inclui aquela linha, o *Nodemon* desligou o servidor e ativou de novo e eu perdi aquele *post* que tinha feito porque está em memória, está em variável ainda não estamos conectando com o banco de dados.

[10:01] Novamente vou fazer o *post* com a requisição com esse corpo de "Id3" e "título". Quando selecionar o botão *send* retorna uma mensagem que o livro foi cadastrado com sucesso, isso é uma outra coisa que vamos evoluir mais adiante porque da primeira vez ele não tinha sido cadastrado com sucesso e a mensagem foi de que ele cadastrou com sucesso, esse tipo de tratamento nós também vamos ver adiante.

{

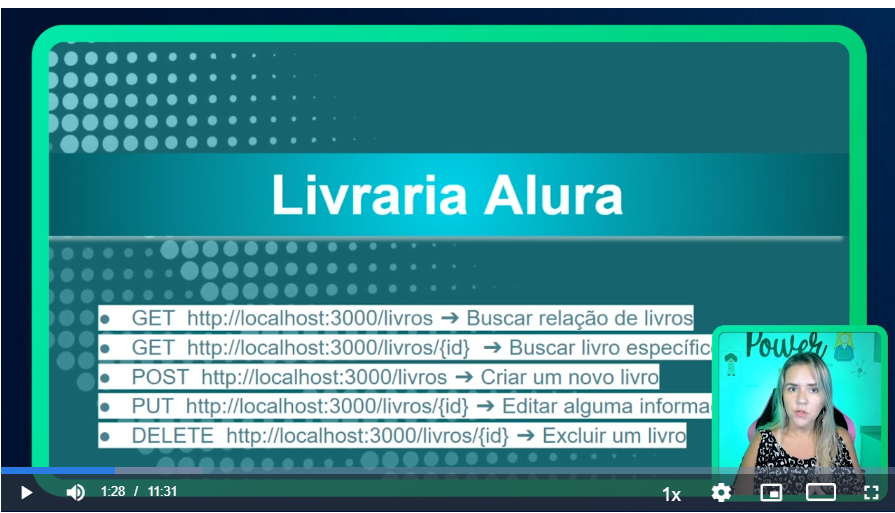
"id": 3,

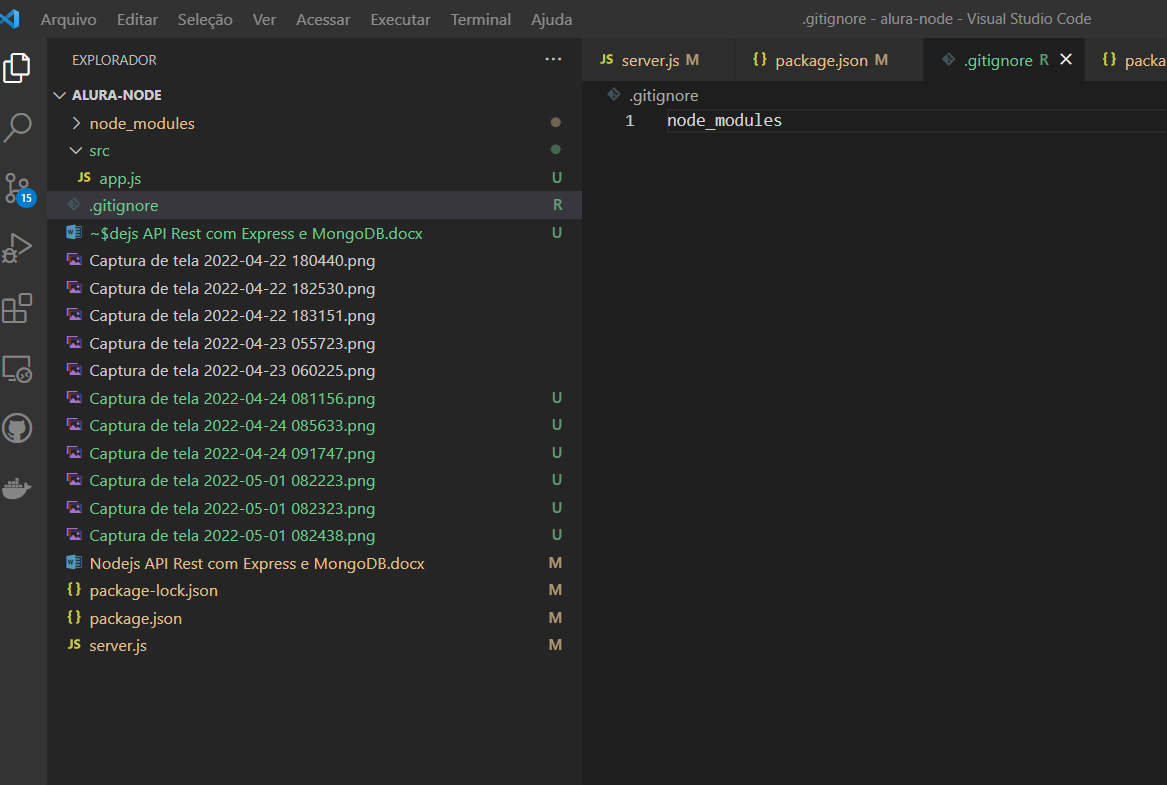
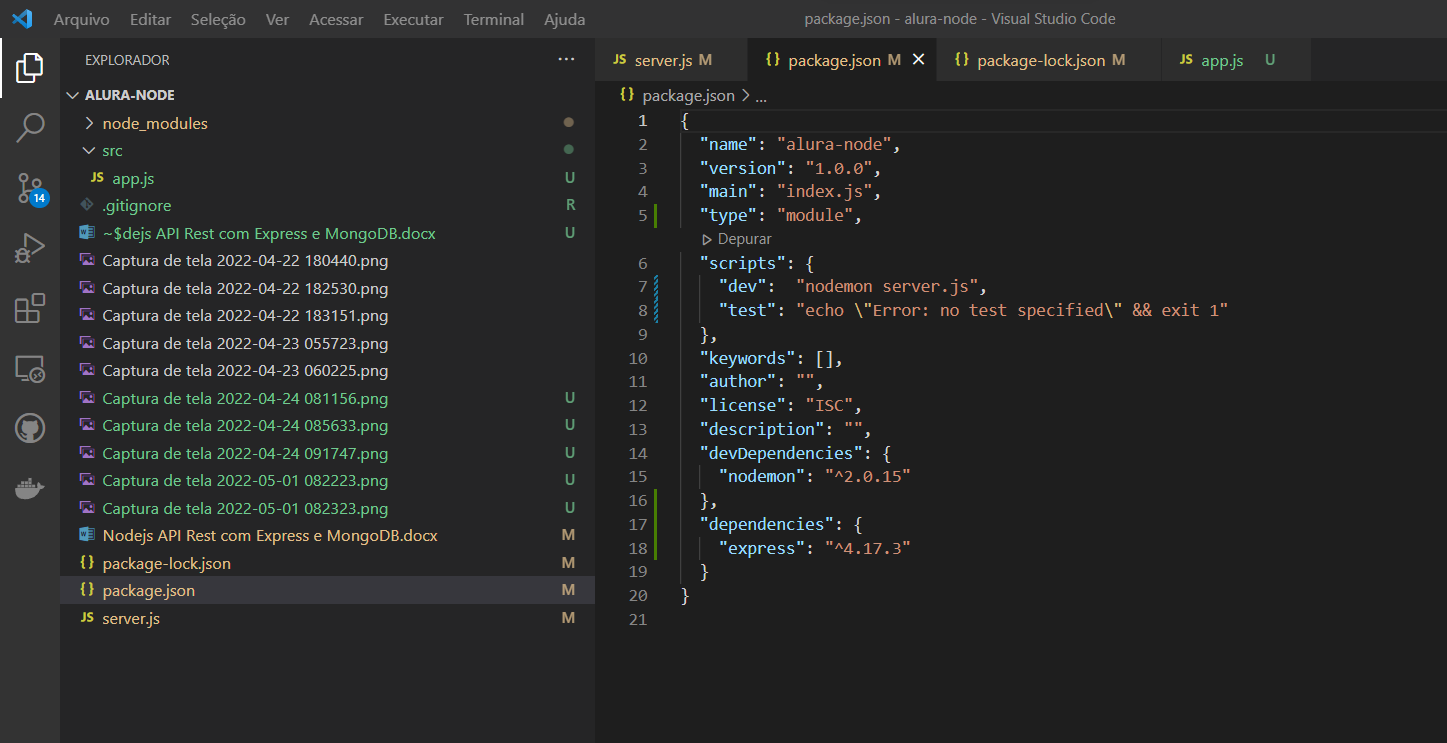
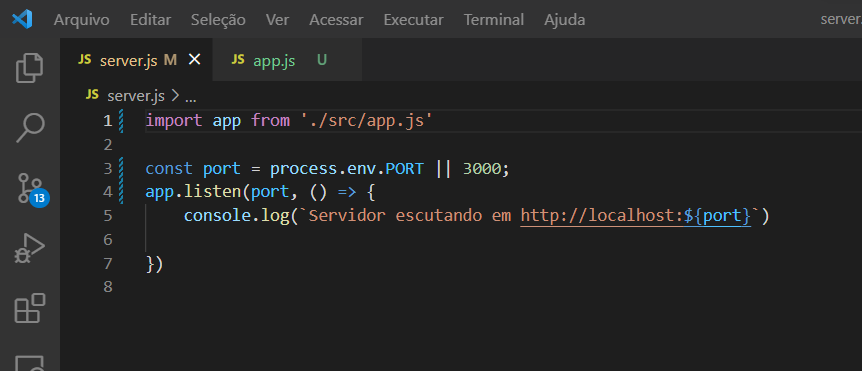
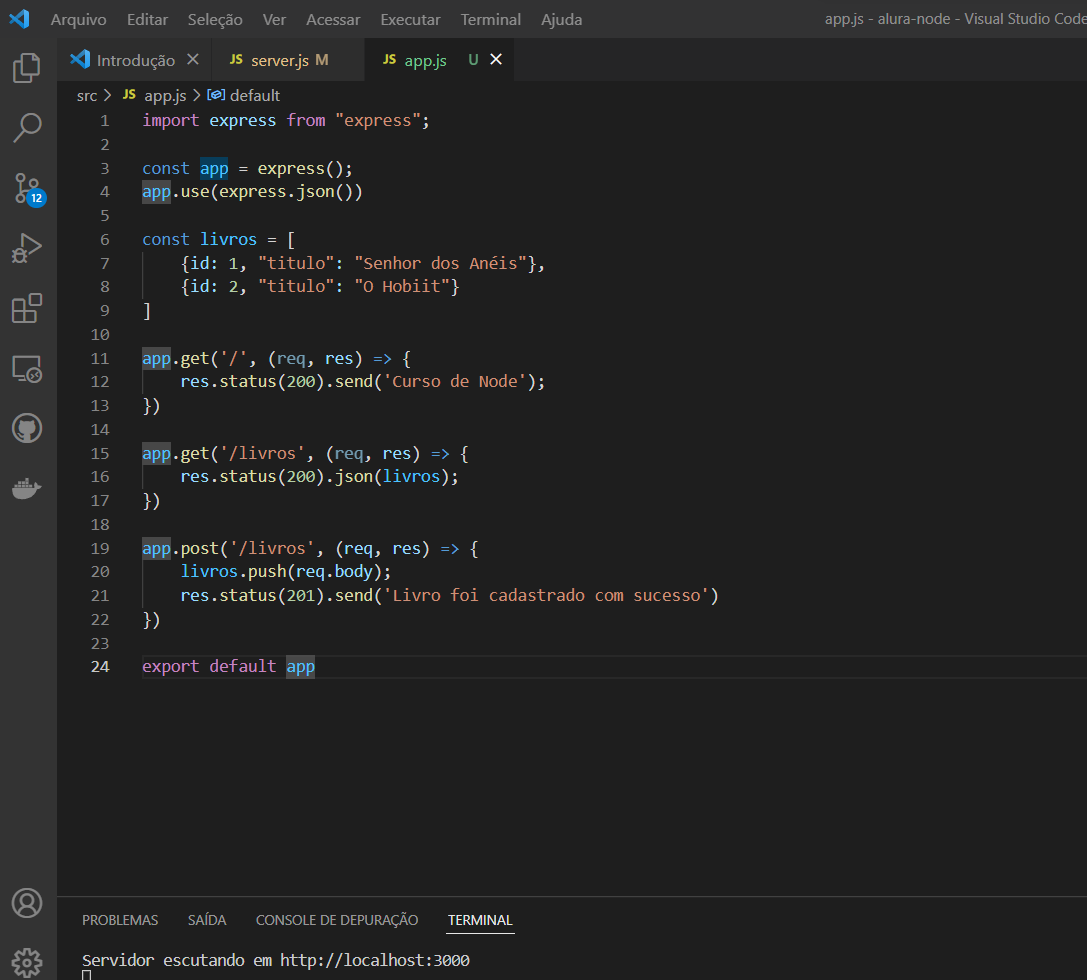
"título": "O Silmarillion"

}COPIAR CÓDIGO

[10:33] Quando eu fizer o *get* agora ele tem que me mostrar a partir do momento que indiquei que estou usando o Express.json. Se formos olhar no array já chegou mais um título "id": 3, "titulo": "O Silmarillion". Basicamente é isso, já temos o *get* e tem um *post*, já conseguimos ler dados da API, fazer uma requisição para obter livros e já conseguimos fazer uma requisição para cadastrar livros na variável em memória, por enquanto.

[11:12] Aos poucos vamos evoluir esse *CRUD* e depois passar todo esse código de forma mais organizada e melhor dividida acessando um banco de dados porque vai ficar cada vez mais interessante. Espero que você esteja curtindo e vejo você no próximo vídeo.





# 06 Para saber mais: introdução ao Postman

Vimos que para testar nossa API e fazer requisições do tipo POST, foi necessário utilizar o aplicativo Postman, que nos permitiu selecionar de forma simples o verbo HTTP a ser utilizado na requisição. Para complementar seus estudos, assista esse [Alura+](https://cursos.alura.com.br/extra/alura-mais/introducao-ao-postman-c182) que mostra alguns recursos adicionais do Postman.

[00:00] Já temos o *get* e o *post* funcionando, testados, vimos no Postman isso acontecendo, cadastramos um novo livro e recuperamos a listagem de livros. Agora vamos começar a programar e evoluir a rota para fazer a atualização, vamos supor que esse livro 3 que eu cadastrei como "O Silmarillion" eu não quero mais que seja esse, quero alterar o título.

[00:27] Vamos para o VS Code e vamos implementar em app.js abaixo do *post* o método *put*, o de atualização vai ser o *put*. Vai ser o mesmo conceito app.put passo o verbo, falo qual é a rota, vai ser para app.put('/livros') só que para eu fazer uma atualização preciso indicar qual livro quero atualizar.

[00:56] Geralmente usamos o que é o identificador único do objeto, da coleção, do banco de dados, vamos usar nesse caso o "Id", para fazer esse *put* eu vou inserir o barra e indicar que vai chegar um parâmetro nessa rota que é o "Id", app.put('/livros/:id'), indicamos isso colocando esse dois pontos Id (':id').

[01:17] E o segundo parâmetro do *put* vai ser a *arrow function* onde eu vou passar requisição, resposta e programar o que tem acontecer, app.put('/livros/:id', (req, res) => { }). Observe, para atualizar um livro vou passar lá na minha rota livros/3, por exemplo, que é o livro que eu quero atualizar e eu preciso localizar esse livro no array e substituir o título. Eu tenho que localizar qual é a posição do array onde tem esse "Id" e fazer a substituição naquela posição específica.

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

})COPIAR CÓDIGO

[02:02] Vamos precisar fazer isso tanto para atualizar quanto para excluir quanto no momento em que eu quiser fazer aquele *get* específico buscando pelo "Id". Ainda não fizemos, tem *get* geral não tem o *get* por Id. Para esses três casos vou ter que ir no array e procurar em que posição está o recurso que quero manipular.

[02:29] Como ainda não estamos trabalhando com banco vou criar uma função para localizar a posição no array que está o recurso que quero buscar e devolver essa posição para que eu possa fazer as manipulações que preciso.

[02:45] Embaixo do *put* eu vou fazer uma função, vamos digitar o function e chamar essa função de "buscaLivro", vou precisar passar um parâmetro para essa função que é o "Id", function buscaLivro(id). Depois que passar o "Id" preciso ir lá naquele array de livros e localizar, abre e fecha chaves function buscaLivro(id) { }.

function buscaLivro(id) {

}COPIAR CÓDIGO

[03:13] Se é uma função ela vai retornar alguma coisa, quero que ela retorne a posição que está o "Id" que passei dentro do array, vou incluir um *return* para retornar e método que temos para manipular array que é o *find index* que consigo achar dado algum parâmetro, em que índice, em que posição está o elemento que quero alterar, manipular ou visualizar.

[03:44] Vamos fazer return livros que é a variável que tem o nosso array e usar o *find index*, return livros.findIndex. E no *findIndex* me pede um predicado que é uma **função lambda** que vou dizer o que tem que acontecer, o que tenho que comparar ali dentro da minha função para identificar que aquele é o elemento, vou chamar, por exemplo, de return livros.findIndex(livro => ).

[04:17] Vou fazer uma condição : preciso saber se o "Id" do livro é igual ao "Id" que passei. Estou dizendo para ele: para cada elemento do array você verifica, você chama ele de livro e vê se o livro.id, que é a forma como referenciamos o atributo, o parâmetro do meu objeto livro se ele é igual ao "Id" que acabei de passar nesse parâmetro, return livros.findIndex(livro => livro.id == id). Isso é a condição para que eu diga que é verdadeiro, eu encontrei, essa é a posição que quero manipular.

function buscaLivro(id) {

return livros.findIndex(livro => livro.id == id)

}COPIAR CÓDIGO

[04:58] Fizemos essa função e agora preciso usar essa função no app.put, vou criar uma variável para receber esse índice que acabei de programar e no array naquela posição vou trocar o título, vai deixar de ter o título antigo e vai ter o título novo.

[05:24] Para fazer isso vou criar uma variável, chamar de index e passar para o meu *index* o "Id" que está vindo na requisição, let index = buscaLivro(). Estou pedindo para ele ir no "busca livro" e tenho que passar o "Id", o "Id" vem como parâmetro na requisição, vamos isso como let index = buscaLivro(req.params.id);, é que vai está o "Id "que eu passar lá no meu endereço, no meu *endpoint*.

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

let index = buscaLivro(req.params.id);

})COPIAR CÓDIGO

[06:11] Aqui não é exigido ponto e vírgula, mas gosto de colocar como boa prática, sempre finalizarmos uma instrução com ponto e vírgula. Tenho o meu índice, sei qual é se ele encontrou certo vou no meu array de livros na posição que recuperei na variável *index* e vou substituir o título pelo título que chegou também, vai ser livros na posição que eu encontrei ponto título vai receber o que veio no corpo da requisição no campo título, vai ser *req* ponto *body* porque já vai ter vindo no corpo.

[07:00] Aqui no *put* ele trabalha com duas situações, ele vem parâmetro no caminho do *endpoint* e vem parâmetro do corpo, está vindo o objeto no corpo da requisição. Vamos usar o *body* ponto título, que é o que está vindo, livros[index].titulo = req.body.titulo;.

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

let index = buscaLivro(req.params.id);

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

})COPIAR CÓDIGO

[07:19] Feito isso vou estar substituindo o que veio no corpo da requisição com o nome de título pela posição que localizei o título anterior. Depois disso podemos simplesmente devolver o array para que isso seja visível que o nome foi alterado, vou fazer um *res* ponto, pode colocar o *status* se não quiser colocar o *status* porque ainda estamos trabalhando com 200 para tudo, pode deixar sem o *status* que automaticamente já coloca como *default* o 200, colocamos JSON e mando devolver o livros, o próprio *array* inteiro, res.json(livros);.

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

let index = buscaLivro(req.params.id);

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

res.json(livros);

})COPIAR CÓDIGO

[08:08] E vamos fazer esse teste e ver se vai dar tudo certo. Vamos no Postman, quando fizermos o *get* não vai ter mais "O Silmarillion" porque estamos trabalhando com o Nodemon, cada alteração ele derruba o servidor e sobe o servidor de novo.

[08:27] Como eu já tenho ele no corpo, vou passá-lo, vou fazer um *post* e quando eu fizer o *get* o meu "O Silmarillion" já vai aparecer, ele já está no corpo ("*body*"): "id": 3, "titulo": "O Silmarillion". Se eu quiser alterar preciso primeiro passar o Id do livro que quero alterar no meu *endpoint*, vai ser livros/3 e vou pedir para alterar esse 3 mesmo, vai ser livros/3.

[09:01] No corpo da requisição eu não preciso passar o "Id", passo somente o título e digo que não vai mais ser "O Silmarillion" vai ser ”titulo”: "Beowulf". Por fim, falta mudar o verbo, no verbo aqui eu vou trocar para *put* para ele atualizar o livro.

[09:25] Quando eu selecionar o botão *send*, ele já me devolveu o array de livros inteiro, que foi o que programamos no *put* e o "Id:3" já foi atualizado, o Id 3 que era "O Silmarillion", ele já localizou no array e trocou, de fato já houve uma alteração, já conseguimos fazer isso mesmo não estando conectados a um banco de dados ainda.

[09:57] Cada vez vai ficando mais legal a API e a forma com que vamos entendendo e trabalhando com os verbos e com todos os métodos. Vamos voltar no VS Code em app.js, vou aproveitar que acabamos de fazer esse método para o *put* e vou duplicá-lo para programarmos o *get* por Id, podemos copiar esse código app.put, vamos jogá-lo lá em cima para ficar organizado todos os *gets*, uma embaixo do outro e colei e vou trocar para *get*.

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

let index = buscaLivro(req.params.id);

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

res.json(livros);

})COPIAR CÓDIGO

[10:33] A única diferença é que eu não vou ter a linha livros[index].titulo = req.body.titulo; para substituir título só vou ter a busca pelo índice e na hora de devolver vou devolver somente o que está na posição que ele localizou. O *get* por Id o usuário vai passar no caminho da requisição o Id, ele vai localizar a posição do array e vai devolver um JSON com aquele livro específico.

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

let index = buscaLivro(req.params.id);

res.json(livros[index]);

})COPIAR CÓDIGO

[11:09] Vamos testar agora no Postman, se fizermos o *get* novamente só vai ter dois porque ele reiniciou tudo. Vamos verificar, eu tenho o um e o dois, se eu colocar um válido que é o dois ('/2') ele já trouxe para mim somente o dois, que é "O Hobbit". Parte do *CRUD* está pronto, só falta basicamente a exclusão. Já temos a listagem de livros, a parte de obter um livro por Id, a criação de um livro e a alteração de um livro. Daqui a pouco finalizamos o CRUD e espero que esteja gostando.

[00:00] Agora vamos finalizar o *CRUD* porque já fizemos o *get*, o *post*, o *put* e agora vamos fechar com o *delete* para conseguir excluir um livro lá do array.

[00:11] Eu vou copiar e colar esse método *put* em app.js e vou programar o *delete* porque é bem parecido, app.delete('/livros/:id', (req, res) => {. Eu vou introduzir um outro conceito que é atribuição via desestruturação que é um recurso interessante do JavaScript que consigo atribuir para uma outra variável valores retirados de um array ou de um objeto. Por exemplo, a requisição é um objeto se declarar uma variável Id entre chaves, consigo pegar e dizer só let {id} = req.params; que ele já vai conseguir pegar exatamente isso o req.params.id e atribuir para a variável Id.

Código em app.js para copiar e alterar:

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

let index = buscaLivro(req.params.id);

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

res.json(livros);

})COPIAR CÓDIGO

[01:02] Eu vou deixar um material extra falando mais sobre atribuição via desestruturação porque é um recurso interessante que vale a pena conhecer. E vou usar essa variável "Id" no lugar desse (req.parms.id), uso let index = buscaLivro(id);, essa variável que atribui vou usar para buscar o livro que quero apagar. E vou substituir essa linha livros[index].titulo = req.body.titulo;, eu não quero trocar o título, quero de fato apagar uma posição do meu array, o elemento que está em uma determinada posição.

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

let {id} = req.params;

let index = buscaLivro(id);

res.json(livros);

})COPIAR CÓDIGO

[01:34] Vamos chamar o array, que é o livros e eu vou usar o método *splice* para poder apagar e aqui nesse método eu tenho que passar dois parâmetros, qual é a posição de início e quantos elementos vou excluir.

[01:50] No caso, vou usar a posição *index*, que é onde eu recuperei qual é o índice do elemento que quero excluir e o segundo parâmetro vai ser "1" que é para apagar somente aquele elemento, não quero apagar uma sequência de elementos a partir dali, eu quero apagar somente aquele livro, livros.splice(index, 1);.

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

let {id} = req.params;

let index = buscaLivro(id);

livros.splice(index, 1);

res.json(livros);

})COPIAR CÓDIGO

[02:08] E no retorno vou mandar uma mensagem, ao invés de devolver um JSON já que vou ter excluído, vou devolver uma mensagem, vou usar o *string template* no livro, passar res.sen(Livro ${id} removido com sucesso);. Basicamente isso, está pronto, com essa função já conseguimos apagar um livro e confirmar que ele foi apagado no Postman.

Parte do código em app.js alterado:

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

let {id} = req.params;

let index = buscaLivro(id);

livros.splice(index, 1);

res.send(`Livro ${id} removido com sucesso`);

})COPIAR CÓDIGO

[02:40] Vamos no Postman, buscar todos os livros ele vai ter o "1" e o "2", "O Senhor dos Anéis" e "O Hobbit" e vou pedir para ele apagar o livro "1", que é "O Senhor dos Anéis". Vou utilizar o método *delete*, então vou alterar o verbo para *delete*, localhost:3000/livros/1, que é o elemento que quero apagar. Selecionei o botão *send* e retornou a mensagem que eu configurei, "Livro 1 removido com sucesso".

[03:06] Se eu fizer o *get* agora geral pedindo a listagem completa, observe que ele só vai ter o 2 ele removeu corretamente somente o "1" e já atualizou a variável em memória, o array, mantendo somente o livro que quisemos que ficasse.

[03:27] Basicamente fechamos o *CRUD* usando o *Express*, nesse momento ainda está tudo no mesmo arquivo, não vai ficar assim a partir da próxima aula a API já vai está com uma cara mais profissional, vamos separar em pastas, separar as funcionalidades em *controller*, em *model*, em *rotes*, vai ter acesso a banco, vai ficar cada vez melhor para utilizarmos.

[03:55] Basicamente deu para dar uma introdução no *Express* e no conceito de *CRUD*, como manipulamos os recursos e fazemos um *CRUD* simples em memória. Vejo vocês no próximo vídeo para começarmos a usar o MongoDB, persistir esses dados no banco e recuperar depois para que não percamos o que foi feito a cada *livereload*. Até a próxima.

# 10Faça como eu fiz: listando seus livros

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103628/next)

Agora é a sua vez de criar sua API!

Crie uma API com uma requisição GET para retornar uma lista com seus livros favoritos!

VER OPINIÃO DO INSTRUTOR

### Opinião do instrutor

Para criar uma API, primeiro precisamos criar uma pasta chamada api-livros, por exemplo iniciá-la com o comando npm init -y. Com o terminal aberto, vamos instalar o Express através do comando npm install express.

Depois, vamos criar um arquivo chamado server.js, e dentro dele vamos declarar o pacote express e instanciá-lo.

import express from “express”

const app = express()COPIAR CÓDIGO

Para que nossa API comece a aceitar requisições, ela precisa escutar por novas requisições em uma determinada porta, ou seja, um local no nosso computador que irá receber requisições. Para isso, vamos usar o método listen() da nossa variável app, passando como parâmetro, o número da porta que vamos ouvir e uma função que será executada quando nossa API já estiver aceitando requisições, com essa função podemos emitir uma mensagem no terminal com console.log() avisando que está tudo funcionando:

import express from “express”

const app = express()

app.listen(3000, () => console.log(‘API já está funcionando e aceitando requisições!’))COPIAR CÓDIGO

Agora, para declarar nossa primeira requisição, temos que escolher seu método. Como queremos enviar informações que estão dentro da nossa API, vamos usar o método HTTP chamado GET. Todo método HTTP possui uma função com o mesmo nome na instância do express. Então, nós podemos usar o método app.get() para declarar uma rota, ou seja, um caminho da nossa API que seja acessível.

Nesse método, precisamos indicar qual o trecho da URL (ou rota, endpoint, caminho) que será acessado e uma função que vai gerar a nossa resposta. Através da URL, podemos identificar com o que estamos trabalhando. Sempre identificamos documentos e coleções de documentos pelo seu nome no plural. Então, se tivermos uma URL como /clientes, sabemos que se trata de uma API que trabalha com uma coleção de clientes. Como vamos fazer uma lista de livros, nossa URL será /livros, e no segundo parâmetro vamos fornecer uma função que recebe um objeto de requisição e resposta:

app.get(‘/livros’, (requisicao, resposta) => {

})COPIAR CÓDIGO

Dentro da função, vamos gerar as informações e usar o objeto resposta para enviar as informações que queremos na requisição. Como precisamos enviar uma lista de livros e queremos enviá-la em formato JSON, vamos formatar a resposta para devolver um Array com os livros. Podemos fazer isso diretamente na resposta, como foi demonstrado no vídeo ou criar uma variável à parte para isso, conforme abaixo:

app.get(‘/livros’, (requisicao, resposta) => {

const livros= [

{ id: 1, nome: 'Senhor dos Anéis' },

{ id: 2, nome: 'O Hobbit' },

{ id: 3, nome: 'O Silmarillion' }

]

resposta.send(livros)

})COPIAR CÓDIGO

Feito isso, o código da nossa API ficará assim:

import express from “express”

const app = express()

app.get(‘/livros’, (requisicao, resposta) => {

const livros= [

{ id: 1, nome: 'Senhor dos Anéis' },

{ id: 2, nome: 'O Hobbit' },

{ id: 3, nome: 'O Silmarillion' }

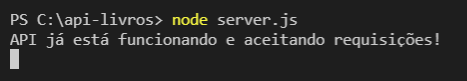
]

resposta.send(livros)

})

app.listen(3000, () => console.log(‘API já está funcionando e aceitando requisições!’))COPIAR CÓDIGO

Agora executamos nossa API pelo terminal com o comando node server.js, vamos ver a mensagem de que nossa API já está funcionando:



Para testar, podemos acessar a URL http://localhost:3000/livros pelo Postman:



# 11O que aprendemos?

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103629/next)

## Nessa aula, aprendemos:

* A instalar o Express e os benefícios de se usar um framework para o desenvolvimento de aplicações Web;
* Criar um servidor local usando o Express e configurando-o para escutar requisições em uma porta específica do nosso computador;
* Como estruturar as requisições da API, utilizando corretamente os verbos HTTP e padronizando o endpoint de acordo com o padrão REST;
* Devolver dados como resposta às requisições, usando o padrão JSON;
* Utilizar o Postman para fazer as requisições do tipo GET, POST, PUT e DELETE, cujas últimas três não conseguimos simular pelo navegador.

# Banco relacional x NoSQL

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/nodejs-api-rest-express-mongodb/task/103582/next)

Play Video

## Transcrição

[00:00] Na última aula instalamos a biblioteca do Express e já vimos como lidar com as requisições lá dentro da aplicação. Vimos como obter uma listagem de livros, como cadastrar um livro, como atualizar e agora chegou o momento de persistirmos os dados, conectar com algum banco de dados e fazer esse trabalho de persistir os dados e devolver os que estão no banco.

[00:27] Uma das dúvidas mais comuns é que tipo de banco utilizar. Aqui no curso vamos usar o MongoDB, mas é muito comum você como desenvolvedor back-end ter que tomar essa decisão de escolher qual banco você vai utilizar no projeto. E aí sempre tem esse questionamento, eu uso um banco relacional ou não relacional?

[00:51] Eu vou listar aqui algumas características de forma breve dos dois tipos de banco para ajudar você na sua decisão de usar ou um tipo de banco ou outro e lembrando que para os exemplos que eu falar aqui temos cursos na Alura tanto de bancos relacionais quanto de banco NoSQL. Se você achar que vale a pena investir, escolher um banco para investir e estudar procure os cursos aqui da Alura que você vai ter um excelente aproveitamento, vai melhorar o seu aproveitamento no curso.

[01:28] Banco relacional ou NoSql, como escolher entre um ou outro? Trabalhar com dados tem sido cada vez mais complexo na área de TI porque temos uma variedade de dados muito grande. Vamos pensar no WhatsApp, por exemplo, são bilhões de pessoas todos os dias, todos os momentos mandando mensagens de texto, de vídeo, links, áudio, você tem muitos tipos de dados e você precisa armazenar isso em algum lugar.

[02:04] Como escolher entre um banco relacional ou um banco não relacional? O banco relacional tem como característica a utilização de tabelas para armazenamento dos dados, você tem tabelas com linhas e colunas, cada coluna é como se fosse um campo e cada linha é um registro aí nessa tabela, de forma bem assimilar a uma planilha, por exemplo, que você cadastra lá, você tem linhas colunas e você insere os seus dados.

[02:34] Ele trabalha com relacionamento através de chaves primárias e chaves estrangeiras. Eu exemplifiquei aqui com duas tabelas, uma tabela de livros e uma tabela de autores. Observe que na tabela de baixo que é a tabela de autores eu tenho aqui o ID que é uma chave primária do autor que vai ser um identificador único para esse autor, eu tenho o nome do autor e a nacionalidade.

[03:04] Na tabela de cima que é a tabela de livros eu tenho um ID do livro que é uma chave primária, o identificador único do livro, tem o título e tenho aqui um autor\_id.

[03:20] Esse autor ele vai fazer uma chave estrangeira com a tabela de autor para eu não ter que ter na minha tabela de livros especificando o nome do autor, a nacionalidade do autor e todos os campos que eu quisesse para autor, eu referencio esse autor apenas com um código ali que é a chave estrangeira. A chave estrangeira que está aqui na tabela de livros faz referência a chave primária da tabela de autores.

[03:53] Por exemplo, no meu primeiro livro eu disse que o autor\_id é o autor 2, isso me faz crer que o autor desse primeiro livro, Anjos e Demônios, é o Dan Brown. Essa relação entre tabelas acontece através das chaves, temos as chaves primárias e as chaves estrangeiras que vai fazer eles se relacionarem.

[04:18] Para termos uns exemplos de bancos, temos o MySQL que é um banco relacional, tem o PostgreSQL, tem o SQL Server, são tipos de bancos relacionais. Todos tem SQL porque a manipulação de dados nos bancos relacionais se baseiam em SQL, Structured Query Language, que é uma linguagem de consulta estruturada.

[04:55] Essa linguagem tem uma sintaxe padrão para que você faça operações nesse banco, por exemplo, se eu quero recuperar todos os livros eu vou fazer uma consulta colocando essas palavras reservadas: select \* from livros e se eu quero incluir uma condição eu vou colocar: where id > 2. É uma forma com que eu faço a consulta, que eu organizo a consulta e a operação que eu quero fazer.

[05:27] Por exemplo, nessa segunda eu estou fazendo uma exclusão na tabela de autores para o autor Paulo Coelho, estou usando o delete from, estou falando de qual é a tabela que eu vou apagar e estou incluindo a condição e a mesma coisa embaixo, estou inserindo dados em uma tabela, eu tenho essa palavra reservada insert into.

[05:53] A mesma coisa seria se eu quisesse juntar dados entre tabelas, eu iria usar uma linguagem estruturada para fazer esse tipo de operação. Por isso que tudo é meio que SQL porque eles usam o SQL para fazer as queries e fazer as operações.

[06:19] Vamos falar um pouco agora dos bancos NoSQL, que é Not Only SQL, não apenas linguagem estruturada de consultas. Nesses casos, nos bancos não relacionais, os dados não precisam estar divididos em tabelas. Os dados podem estar em uma estrutura única de um documento, por exemplo, eu posso ter uma única coleção chamada livros e dentro de livros eu tenho um cadastro de tudo ali dentro daquele livro, eu tenho o cadastro do autor, eu tenho o objeto autor, eu tenho o objeto editora com vários dados da editora.

[06:56] Vou dar um exemplo aqui, aquele mesmo caso de tabela de livros e autores, aqui seria um documento representando um livro e dentro desse documento eu tenho um objeto que já referência o autor.

[07:13] Eu já tenho ali o cadastro daquele autor completo, eu não preciso buscar esse dado externamente quando eu for pegar todas as informações de um livro, já está tudo aqui. É por isso que os bancos não relacionais têm sido muito utilizados para big data porque fica mais fácil obtermos as informações sem ter que fazer vários joins em várias tabelas vendo qual é a chave que relaciona um campo ao outro. Assim eu consigo buscar todos os documentos e as informações já estão todas ali.

[07:49] Isso cada dia mais tem ganhado o mercado por conta disso, pela facilidade não só de buscar dados, mas você de escalar, distribuir esses dados em vários clusters ali. Cada vez mais você vai ouvir falar em bancos não relacionais.

[08:09] Esses bancos têm vários tipos de armazenamento, por exemplo, bancos que são orientados a documentos, o MongoDB é um exemplo desse, ele é um tipo de banco que vai armazenar uma coleção de documentos do tipo Json, por exemplo, que é o que vamos ver aqui no curso.

[08:30] Tem bancos que armazenam com o tipo chave-valor, que aí tem Oracle, tem o Infinity DB. Tem também bancos que armazenam com o tipo de grafos, o Neo4J é um exemplo desses. E para citar mais um exemplo têm bancos que o tipo de armazenamento é colunas amplas, podemos citar, por exemplo, o Cassandra, o HBase.

[08:59] Existem vários modelos de bancos NoSQL e você vai decidir baseado no seu negócio qual é a melhor estratégia, qual é o melhor banco para utilizar para o seu projeto. Aqui, como eu citei, vamos usar o MongoDB, aqui é a página principal do MongoDB e se olharmos em "Produtos" ele tem vários tipos de produtos, ele tem o Mongo Atlas que é o serviço em nuvem que é o que vamos utilizar na próxima aula, vamos configurar uma conta no Atlas e fazer todo o cadastro para usar ele em nuvem.

[09:37] Você tem uma versão Enterprise que é uma versão paga, tem a Community que é uma versão gratuita e que você pode instalar local, tem o Realm e tem várias opções de ferramentas para você ter acesso a esses dados. Vamos falar um pouco sobre elas também no material complementar que é o Compass, o Shell, tem plugin do VS Code, tem o Atlas CLI que também vamos ter acesso aos dados por ele e tem conectores para você fazer a conexão ao banco de dados. Um exemplo é o Mongoose que vamos usar aqui no curso também.

[10:15] Na próxima aula vamos colocar a mão na massa, começar a usar o MongoDB e eu espero que vocês gostem dessa parte de persistência e de evoluir a nossa API. Espero por vocês.