[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, () => {

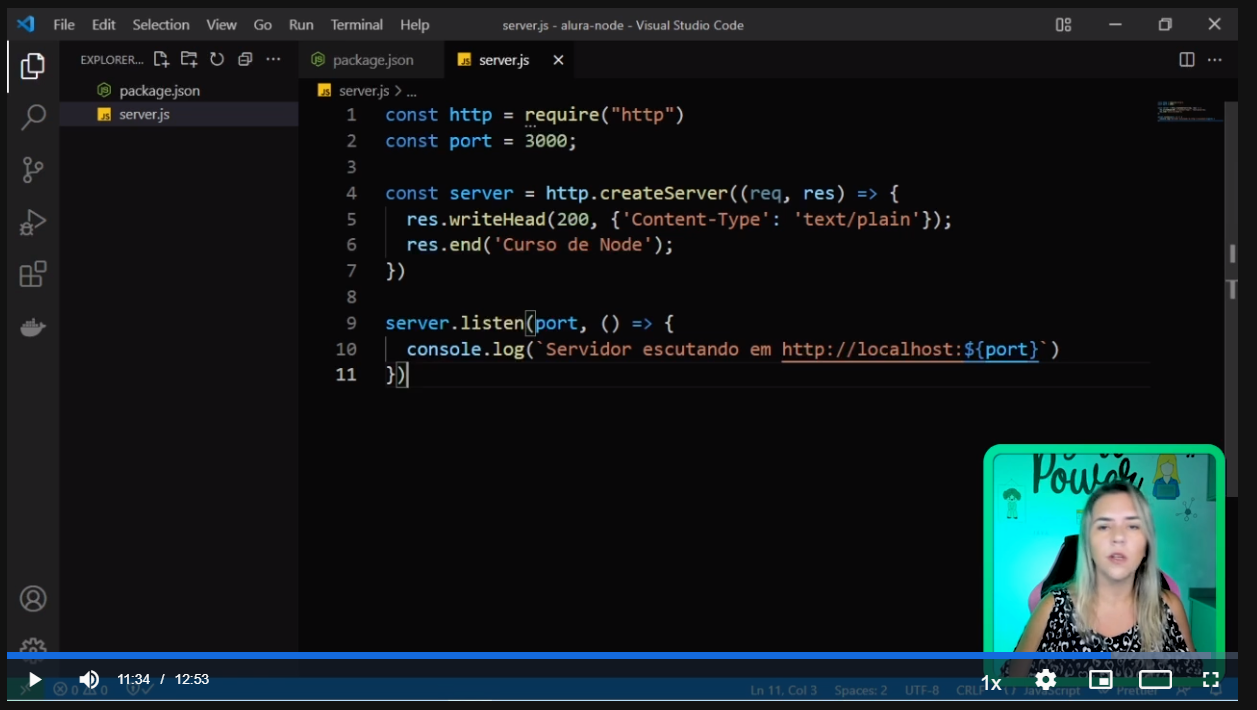
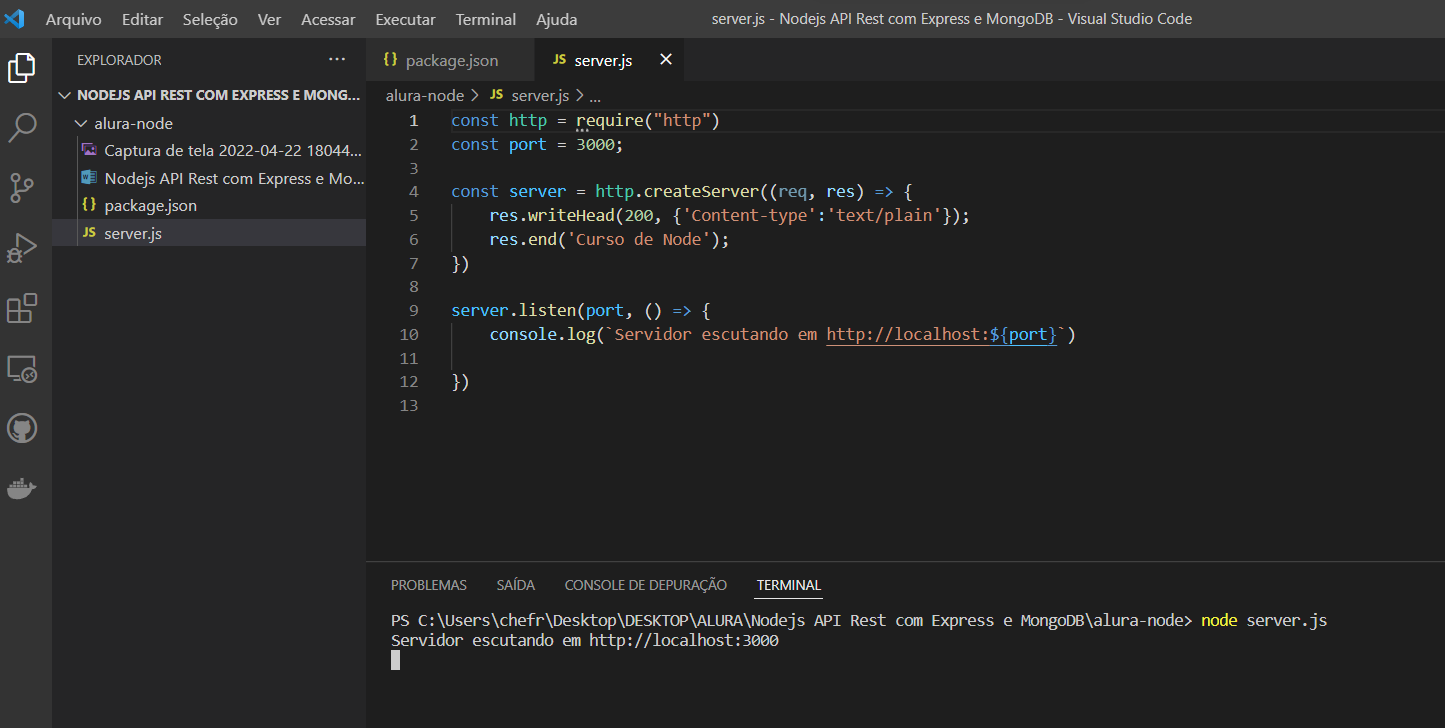
console.log(`Servidor escutando em http://localhost:${port}`)

})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.