Node.js: criando uma API Rest com Express e MongoDB

Criando o projeto com Node.js - Apresentação

Olá! Boas-vindas ao curso de **API REST com *Node.js*, *Express* e *MongoDB*** da Alura! Eu sou a Juliana Amoasei e irei te acompanhar ao longo dessa jornada.

> ***Audiodescrição:*** *Juliana é uma mulher branca de cabelos curtos e lisos pintados de azul, olhos castanhos, veste uma camisa preta da Escola de Programação e DevOps*

Para quem é este curso?

Este curso é destinado a você que está iniciando seus estudos em *back-end* e escolheu o *JavaScript* como linguagem, utilizando o Node.js.

Se já praticou as estruturas básicas da linguagem, como tipos de dados, *arrays*, objetos, funções e classes, e quer evoluir, criando suas primeiras aplicações, este curso é para você!

O que vamos aprender?

Neste curso, vamos aprender **o que é uma API**, sua importância e o papel das APIs no desenvolvimento web. Vamos **criar uma API** de uma livraria e entender como algumas coisas do mundo real, como livros, autores, editoras, são representados em uma API.

Essa API será criada com algumas ferramentas muito utilizadas no mercado, como o **Express** e o **MongoDB**, em conjunto com o **Node.js**.

Começaremos também a trabalhar com **bancos de dados**. Entenderemos alguns tipos existentes e integraremos a nossa API a um banco de dados MongoDB.

Identificaremos também algumas **partes principais de uma API REST**, como, por exemplo, as rotas, os modelos e os controladores.

Pré-requisitos

Para aproveitar melhor este curso, é importante que você já tenha estudado os cursos de **fundamentos do JavaScript**, que estão nos pré-requisitos da formação, e também ter concluído o curso de **HTTP**, que também está na lista dos pré-requisitos. Além disso, é interessante que você já tenha alguma prática com o **uso dos comandos no terminal**.

> *Durante o curso, além dos vídeos e das atividades extra, você também poderá contar com o apoio da Alura e de nossa comunidade no Discord e também no fórum.*

Vamos estudar e começar com a nossa primeira API REST!

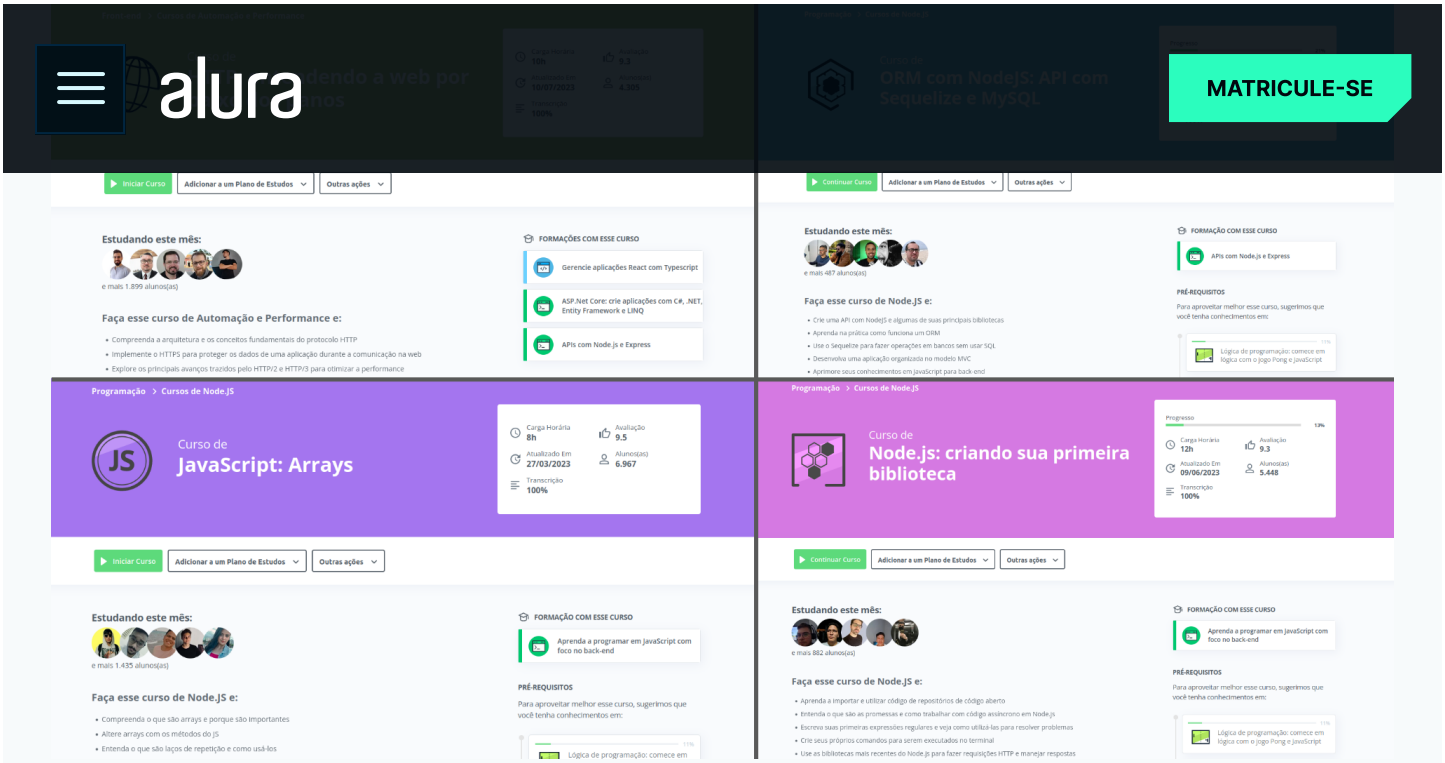
Neste curso, vamos construir o sistema interno de uma livraria. Quando entramos em uma livraria e desejamos informações sobre algum livro, autor ou editora, precisamos consultar o sistema.

Normalmente, perguntamos sobre o livro de determinado autor, a descrição do livro ou se pertence a uma editora específica, no gênero ficção-científica, por exemplo. A partir dessa consulta, queremos saber o preço do livro para avaliar se iremos comprá-lo ou não.

Quando falamos de um sistema que busca e armazena informações, como adicionar um livro ao catálogo, corrigir o nome de um livro, encontrar todos os livros de ficção científica, nos referimos, na maior parte das vezes, à **construção de uma API** que concentre as informações que queremos e as maneiras como queremos interagir com esses dados.

Entendendo APIs

Podemos encontrar APIs funcionando em praticamente tudo que acessamos na internet. Abaixo, são exibidas quatro telas diferentes da plataforma da Alura, que são páginas de cursos.



Se você acessar o site da Alura agora e navegar por cursos diferentes, notará que todas as telas seguem um padrão, ou seja, um ***layout*** onde o nome do curso é exibido, informações de quem está estudando, informações sobre carga horária, avaliação, quantidade de pessoas inscritas no curso, ementa do curso, e assim por diante.

Na prática, temos que pensar que a Alura tem milhares de cursos e cada página HTML referente a cada um deles não fica pré-montada. Isso não seria prático ou viável. O que acontece é que temos APIs que fornecem todos os dados e todas as informações que o front-end consome e utiliza para montar as telas.

Todas as informações que vemos em *e-commerces*, plataformas e redes sociais são o resultado de uma API que possui todos os dados necessários, por exemplo, no caso da Alura, são os dados dos cursos e dos estudantes, e na nossa livraria, serão os dados dos livros, das editoras, e assim por diante, em conjunto com o front-end que pega esses dados e os exibe na tela de modo que nossa pessoa usuária consiga interagir com eles.

O que é uma API?

Já sabemos que será construída uma API durante esse curso. Mas, o que significa API? **API** é um acrônimo para *Application Programming Interface* (Interface de Programação de Aplicações). O termo que devemos focar é "interface". Sempre que nos depararmos com esse termo, devemos pensar em um ponto de contato.

Como funciona uma aplicação?

Vamos analisar alguns exemplos sobre como funciona uma aplicação desde o front-end até o acesso ao banco de dados. De forma bastante simplificada, o fluxo começa pelo **front-end**, que é a **interface** do nosso produto com a pessoa usuária, fazendo algum tipo de comunicação com o **back-end**, que é o que estamos construindo neste curso.

O front-end se comunica com o back-end. Ele pede informações sobre um livro, envia dados de um cadastro, por exemplo, e o back-end, por sua vez, se comunica com o que chamamos de **camada de dados**. A parte de dados é mais importante de qualquer sistema, porque é literalmente onde estão todas as informações.

Portanto, toda a comunicação, isto é, toda a interface feita com os dados é através de uma API. É o que chamamos de back-end do sistema, onde nós desenvolvemos essas interfaces.

Em resumo: a comunicação começa no front-end, que envia uma requisição ao back-end solicitando dados de um livro ou enviando algum cadastro. Nessa requisição, podem estar incluídos dados de uma pessoa usuária, por exemplo. Na sequência, o back-end processa essas informações e vai até a seção de dados para acessar tais informações.

Podemos exemplificar mais efetivamente com uma situação: o back-end precisa confirmar se determinado livro está na base de dados ou se certa pessoa usuária existe na nossa base.

A seção de dados retorna com essas informações e, se tudo estiver correto, por exemplo, se os dados do livro solicitado forem obtidos, o back-end tem a função de processar essas informações para que o front-end consiga interpretá-las.

Para concluir, o front-end vai exibir todas essas informações na tela para a pessoa usuária final ou realizar o processamento que precisa fazer na parte do front-end.

Conclusão

Se ainda houver dúvida sobre como é uma API, qual sua aparência, e o que vamos entregar, não se preocupe, pois é justamente isso que vamos desenvolver durante o curso. Vamos elaborar a interface do nosso programa, com a seção de dados e o que vamos fornecer da nossa livraria.

Vamos começar a desenvolver?

Agora que já entendemos o que é uma API e o que será construído durante o curso, vamos construir esse projeto a partir do zero.

Criando o servidor

Criação de um projeto *Node*

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

```
npm init -y
```

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

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

`package.json`:

```
  "name": "3266-express-mongo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

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

```
node -v
```

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

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

```
{
  "name": "3266-express-mongo",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",

// código omitido
```

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

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

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

Criação do servidor

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

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

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

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

server.js:

```
import http from "http";
```

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

Protocolo HTTP

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

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

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

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

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

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

Criando um servidor local HTTP

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

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

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

server.js:

```
// código omitido
```

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

```
const server = http.createServer((req, res) => {
  res.writeHead();
});
```

HTTP *headers*

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

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

```
GET / HTTP/1.1
Host: www.google.com
User-Agent: curl/7.68.0
Accept: text/javascript
X-Test: hello
```

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

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

```
HTTP/1.1 200 OK
Date: Jul 2023 00:19:01 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Content-Security-Policy-Report-Only: object-src
```

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

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

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

Dando continuidade à escrita da função

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

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

server.js:

```
const server = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
});
```

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

Node.js".


```
const server = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Curso de Node.js");
});
```

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

Criando uma conexão com o servidor

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

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

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

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

server.js:

```
server.listen(3000, () => {
  console.log("servidor escutando!");
});
```

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

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

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

```
const PORT = 3000;
```

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

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

```
server.listen(PORT, () => {
  console.log("servidor escutando!");
});
```

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

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

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

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

```
import http from "http";

const PORT = 3000;

const server = http.createServer((req, res) => {
  res.writeHead(200, { "Content-Type": "text/plain" });
  res.end("Curso de Node.js");
});
```

```
});
```

Executando o arquivo

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

```
node server.js
```

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

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

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

Sobre o curso Node.js: criando uma API Rest com Express e MongoDB

O curso Node.js: criando uma API Rest com Express e MongoDB possui **207 minutos de vídeos**, em um total de **67 atividades**. Gostou? Conheça nossos outros **cursos de Node.JS** em **Programação**, ou leia nossos artigos de Programação.

Matricule-se e comece a estudar com a gente hoje! Conheça outros tópicos abordados durante o curso:

- Criando o projeto com Node.js
- Express e primeiras rotas
- Persistindo dados
- Evoluindo a API
- Adicionando funcionalidades

alura

MATRICULE-SE

Nossas redes e apps

Institucional

Sobre nós

Trabalhe conosco

Para Empresas

Para Escolas

Política de Privacidade

Compromisso de Integridade

Termos de Uso

Status

A Alura

Formações

Como Funciona

Todos os cursos

Depoimentos

Instrutores(as)

Dev em <T>

Luri by ChatGPT

Conteúdos

Alura Cases

Imersões

Artigos

Podcasts

Artigos de educação corporativa

Fale Conosco

Email e telefone

Perguntas frequentes