## O que vamos aprender?

Você vai relembrar alguns conceitos importantes sobre o que é o HTTP, além de entender o que é uma API e para que elas servem.
Você vai aprender como utilizar um dos mais famosos e importantes frameworks na construção de APIs com Node: o Express !
Você vai entender como ele funciona e como seu sistema de rotas e middlewares se encaixam para formar uma aplicação pronta para ir para a produção.

### Você será capaz de:

- Entender o que é o HTTP, o que é uma API e o que os dois têm a ver com o Express
- Escrever APIs utilizando Node e Express;
- Entender a estrutura de uma aplicação Express e como organizar seu código;
- Criar rotas e aplicar funções que processam requisições HTTP.

# Por que isso é importante?

O protocolo HTTP é a fundação da web moderna. Ele é a base da comunicação de boa parte do que acontece na web e, portanto, entender bem seu funcionamento é essencial para desenvolver boas aplicações web.
Inicialmente criado para transportar documentos e mensagens simples, o HTTP hoje é responsável pelo tráfego de todo tipo de informação na internet. Boa parte do que é enviado e recebido via HTTP são requisições e respostas a APIs HTTP . É sobre essas APIs que você aprenderá hoje.
APIs são pontos de comunicação entre dois sistemas diferentes, e APIs HTTP são as mais utilizadas para comunicação na web. Para ficar nítido, imagine que você precisa que seu Front-End consulte alguns dados do seu banco de dados. Não faz sentido colocar, por exemplo, o usuário e senha do banco no meio do seu JavaScript e criar uma conexão direta, faz? Se fizéssemos algo do tipo, estaríamos, dentre outras coisas, simplesmente expondo o acesso a todo nosso banco de dados a qualquer pessoa que executasse um "Inspecionar elemento" na página.
Mas então, como o Front-End se comunica com o banco de dados?
Entra no palco, o *Back-End* .
As APIs HTTP, que são o que forma o Back-End da maioria das aplicações web, são as responsáveis por ler os dados no banco e entregá-los para o Front-End, ou por receber dados do Front-End e armazená-los no banco de dados.
Inclusive, você já consumiu APIs HTTP em projetos como o Star Wars e o Online store, e agora chegou a hora de aprender a criar suas próprias APIs para que suas aplicações Front-End (sejam elas web, mobile, desktop ou de qualquer outro tipo) possam se comunicar com seu banco de dados e tomar proveito de regras que seu sistema venha a ter.
Um ponto importante sobre as APIs HTTP é que tudo o que está nelas é *reutilizável por qualquer cliente* . Se você cria um *endpoint* para cadastrar pessoas usuárias, por exemplo, todo o Front-End da sua aplicação vai consumir esse mesmo *endpoint* , não importa em qual aplicação seja usada (web ou mobile).
Utilizando APIs, fazemos ainda mais, com menos código!

### HTTP

Antes de falarmos mais sobre APIs HTTP, vamos relembrar os principais conceitos sobre esse protocolo e entender quais informações podemos enviar ou receber quando estivermos falando com um servidor HTTP.
Vamos relembrar o que compõe uma requisição HTTP. Para isso, analisaremos a requisição que é feita pelo navegador quando abrimos a página https://developer.mozilla.org .

Copiar

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept: text/html
```

Vejamos quais são as informações presentes nessa requisição:

- O método HTTP, definido por um verbo em inglês. Nesse caso, utilizamos o GET , que normalmente é utilizado para "buscar" algo do servidor, e é também o método padrão executado por navegadores quando acessamos uma URL. Veremos mais sobre verbos HTTP em breve.
- O caminho, no servidor, do recurso que estamos tentando acessar. Nesse caso, o caminho é / , pois estamos acessando a página inicial.
- A versão do protocolo (1.1 é a versão nesse exemplo).
- O local (host) onde está o recurso que se está tentando acessar, ou seja, a URL ou o endereço IP servidor. Nesse caso, utilizamos developer.mozilla.org , mas poderia ser localhost:3000 , caso você esteja trabalhando localmente.
- Os headers . São informações adicionais a respeito de uma requisição ou de uma resposta. Eles podem ser enviados do cliente para o servidor, ou vice-versa. Na requisição de exemplo, temos o header Host , que informa o endereço do servidor, e o header Accept , que informa o tipo de resposta que esperamos do servidor. Um outro exemplo bem comum são os tokens de autenticação, em que o cliente informa ao servidor quem está tentando acessar aquele recurso: Authorization: Bearer {token-aqui} . Alguns exemplos extras de headers podem ser vistos aqui. .

Esses são os dados transmitidos em uma request do tipo GET . No entanto, o GET não é o único método HTTP existente. Na verdade, existem 39 métodos diferentes! Os mais comuns são cinco: GET , PUT , POST , DELETE e PATCH , além do método OPTIONS , utilizado por clientes para entender como deve ser realizada a comunicação com o servidor.
A principal diferença entre os métodos é o seu significado. Cada método costuma dizer para o servidor que uma operação diferente deve ser executada. O método POST , por exemplo, costuma ser utilizado para criar um determinado recurso naquele servidor.
Além da diferença de significado, requisições do tipo POST , PUT e PATCH carregam mais uma informação para o servidor: o corpo, ou body . É no corpo da requisição que as informações de um formulário, por exemplo, são transmitidas.
Quando um servidor recebe uma requisição, ele envia de volta uma resposta . Veja um exemplo:

Copiar

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (aqui vêm os 29769 bytes da página solicitada)
```

A composição da resposta é definida por:

- A versão do protocolo (1.1 no nosso exemplo).
- O código do status, que diz se a requisição foi um sucesso ou não (nesse caso, deu certo, pois recebemos um código 200 ), acompanhado de uma pequena mensagem descritiva ( OK , nesse caso).
- Os Headers , no mesmo esquema da requisição. No caso do exemplo acima, o Content-Type diz para o navegador o que ele precisa fazer. No caso do HTML, ele deve renderizar o documento na página.
- Um body , que é opcional. Por exemplo, caso você submeta um formulário registrando um pedido em uma loja virtual, no corpo da resposta pode ser retornado o número do pedido ou algo do tipo.

Após a resposta, a conexão com o servidor é fechada ou guardada para futuras requisições (seu navegador faz essa parte por você).
Note que tanto requisições quanto respostas podem ter headers e um body. No entanto, é importante não confundir uma coisa com a outra: o body e os headers da requisição representam a informação que o cliente está enviando para o servidor . Por outro lado, o body e os headers da resposta representam a informação que o servidor está *devolvendo* para o cliente .

## APIs

API é uma sigla para A pplication P rogramming I nterface. Ou seja, *Interface de programação de aplicação* .
Isso quer dizer que uma API é, basicamente, qualquer coisa que permita a comunicação, de forma programática, com uma determinada aplicação.
Um tipo muito comum de API são as APIs HTTP, que permitem que códigos se comuniquem com aplicações através de requisições HTTP. É desse tipo de API que boa parte da web é feita.
Elas são extremamente importantes nos dias de hoje, em que temos múltiplos clients (web, apps mobile, tvs, smartwatches etc.) se comunicando com o mesmo servidor! É assim que seu Netflix está sempre sincronizado entre seu celular, seu computador e sua televisão. 😃
Nos projetos de front-end, você integrou várias APIs com suas aplicações.
Agora, veja o vídeo abaixo com mais detalhes sobre o que é uma API:

## Contextualizando

A partir de agora, você irá criar APIs, que vão *receber requisições* e *devolver dados* , passando por *validações* , *regras de negócio* , acesso ao *banco de dados* , etc.
Se compararmos uma aplicação web a um restaurante, o Front-End é a área das mesas , garçons e garçonetes: é onde a *comunicação direta com clientes* acontece, onde os pedidos são anotados, e também a parte que leva as receitas da cozinha até a mesa das pessoas.
O Back-End, por sua vez, é cozinha . É onde uma pessoa cozinheira, mediante o recebimento de um pedido, vai *preparar os ingredientes* , *montar a receita* e *devolvê-lo para que uma pessoa atendente apresente esse prato a quem o pediu* . É no Back-End que os dados serão filtrados , manipulados e preparados para envio ao Front-end. Esse, por sua vez, se encarrega de apresentá-los a quem fez o pedido.
Ainda na analogia da cozinha, uma API seria o quadro de pedidos que os setores de "Cozinha" e "Atendimento" usam para se comunicar:
Quando o client envia uma requisição para o Back-End , é como se uma pessoa atendente anotasse o pedido em um papel e o colocasse no balcão para ser preparado pela cozinha .

Quando o servidor envia a resposta para a requisição do client , ele mostra essas informações ao usuário via Front-End. É como se a cozinha entregasse o prato que foi pedido para que o atendente o leve para a mesa da pessoa cliente.
Pra ilustrar, a coisa toda funciona como a imagem abaixo:

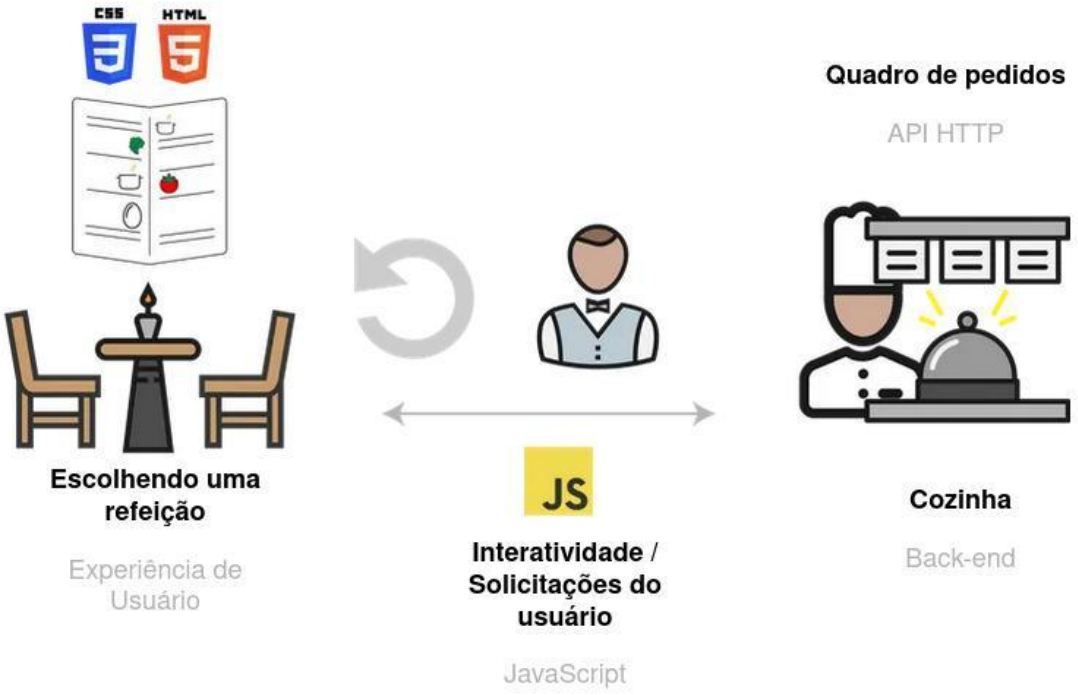


Imagem que demonstra o panorama de uma aplicação web
Daqui pra frente, vamos focar nos conceitos e na construção das APIs, visto que uma API bem feita, assim como um quadro de pedidos bem organizado, pode ser a chave para uma aplicação (ou um restaurante) bem sucedida. 👨‍🍳💻

## E o Express?

O express é um framework Node.js criado para facilitar a criação de APIs HTTP com Node. Ele nos fornece uma série de recursos e abstrações que facilitam a vida na hora de decidir quais requisições tratar, como tratá-las, quais regras de negócio aplicar e afins.
O framework foi construído pensando em um padrão de construção de APIs chamado de REST, que você vai estudar mais à frente. Seu objetivo é nos ajudar a construir APIs de forma mais fácil, essencialmente nos permitindo criar APIs altamente funcionais com metade do trabalho que teríamos para fazer isso "na mão".
Existem outras ferramentas semelhantes no mercado, mas o Express é largamente adotado na comunidade hoje, e dois dos motivos são:

- Ele foi lançado no final de 2010, ou seja, é um framework maduro e “testado em batalha”;
- Ele é um "unopinionated framework" (framework sem opinião). Isso significa que ele não impõe um padrão de desenvolvimento na hora de escrever sua aplicação.

Hoje, o Express faz parte da Node.js Foundation . Isso demonstra o quão relevante ele é para a comunidade.

## Criando uma aplicação com Express

Para começar a ter *um gostinho* do que é programar utilizando o Express, vamos criar o clássico "Hello, world!". Para isso, crie uma pasta chamada `hello-express` e, dentro dela, inicialize um novo pacote Node.js utilizando o npm.

Copiar

```
mkdir hello-express
cd hello-express
npm init -y
```

Agora, instale o Express e crie um arquivo `index.js` . Como qualquer aplicação Node.js, nossa API precisa de um entrypoint, ou seja, um ponto de partida. Por convenção, vamos utilizar o `index.js` .

Copiar

```
npm i express
touch index.js
```

Pronto, você já tem o que é necessário para criar sua primeira API HTTP com o Express. Vamos ao código!
Preencha o arquivo `index.js` com o seguinte conteúdo:

Copiar

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

const app = express(); // 1

app.get('/hello', handleHelloWorldRequest); // 2

app.listen(3001, () => {
  console.log('Aplicação ouvindo na porta 3001');
}); // 3

function handleHelloWorldRequest(req, res) {
  res.status(200).send('Hello World!'); // 4
}
```

E pronto! Esse pequeno script é o suficiente para:

1. Criar uma nova aplicação Express;
2. Dizer ao Express que, quando uma requisição com método `GET` for recebida no caminho `/hello` , a função `handleHelloWorldRequest` deve ser chamada;
3. Pedir ao Express que crie um servidor HTTP e escute por requisições na porta 3001;
4. Ao tratar uma requisição com método `GET` no caminho `/hello` , enviar o status HTTP `200` que significa `OK` e a mensagem "Hello world!".

Para iniciar a aplicação, execute o comando abaixo no diretório da aplicação.

Copiar

```
node index.js
```

Agora, vá até o seu navegador (pode ser o Chrome) e abra a url http://localhost:3001/hello .
Parabéns, você criou sua primeira aplicação node com Express.
Você vai perceber que diferente dos scripts que desenvolvemos até aqui que executavam e acabavam quando chegava ao final do script nossa aplicação vai ficar executando *ad eternum* . Para parar a aplicação pressione `CTRL+C` no seu terminal.
Assista ao vídeo abaixo que consolida tudo que vimos até agora para criar uma aplicação com Express ou caso preferir passe para a próxima seção.

## Nodemon

Uma vez que nossa API está rodando e fazemos modificações no seu código é preciso parar e reiniciar a aplicação executando novamente o `node index.js` . Faça um teste: Deixe sua aplicação rodando e modifique o código da rota `/hello` para ficar assim:

Copiar
```
function handleHelloWorldRequest(req, res) {
  res.status(200).send('Olá mundo!');
}
```

Abra o navegador e faça uma requisição novamente para a URL http://localhost:3001/hello. Você vai perceber que o código continua retornando a mensagem `'Hello World!'` . Para que a mudança seja aplicada você deve parar a aplicação (CTRL+c) e iniciar a aplicação novamente. É bem trabalhoso ter que fazer isso sempre que mudarmos qualquer coisa no nosso código, não é mesmo?
Para facilitar nosso fluxo de desenvolvimento podemos utilizar um pacote chamado Nodemon que reinicia a aplicação toda vez que editamos e salvamos os nossos arquivos. Para utilizar esse pacote, vamos começar instalando ele na nossa aplicação.

Copiar
```
npm i nodemon -D
```

Observe que passamos o parâmetro `-D` que indica ao npm que esse pacote deve ser instalado como uma dependência de desenvolvimento. Por enquanto, não precisamos nos preocupar com o que isso significa. Para poder automatizar o uso do nodemon, vamos abrir nosso arquivo `package.json` e adicionar a seguinte linha:

Copiar
```
//...
// "scripts": {
//    "test": "echo \"Error: no test specified\" && exit 1",
      "dev": "nodemon index.js"
//  },
// ...
```

Agora, para executarmos nossa aplicação, vamos utilizar o seguinte comando:

Copiar
```
npm run dev
```

Pronto, agora sempre que fizermos qualquer alteração no nosso código e salvarmos o arquivo, o Nodemon automaticamente reinicia a aplicação para aplicar as modificações. Faça alguns testes mudando a mensagem retornada e fazendo uma nova requisição para a URL que fizemos.
⚠️ Atenção ⚠️ Apesar de ser uma ferramenta muito útil para desenvolvimento, o Nodemon não deve ser utilizado para rodar a aplicação, pois caso seja disponibilizada para a pessoa usuária final (ou seja, em produção), podemos ter problemas de reinicialização da aplicação, devido ao fato de que qualquer alteração em qualquer arquivo afete a aplicação, fazendo com que toda ela seja reiniciada. Para executar uma aplicação *em produção* , deve-se utilizar o script `start` com o comando `node index.js` .
Agora, podemos partir de cabeça para entender como criar uma API utilizando o Express.

## Roteamento

O aspecto mais básico de uma API HTTP se dá através de suas rotas , também chamadas de *endpoints* . Uma rota ou endpoint é definida pelo método HTTP e caminho .
Na nossa aplicação de "Hello, world!", por exemplo, registramos uma rota `GET /hello` . Repare que, se tentarmos utilizar qualquer outro método ou qualquer outro caminho para acessar essa rota, receberemos um erro do Express, juntamente com um status `404 - Not Found` .
A forma que o Express trabalha com rotas é a seguinte: ao registrarmos uma rota, estamos dizendo para o Express o que fazer com requisições que contenham aquele *método* e *caminho* . Voltando para a nossa cozinha, é como se estivéssemos definindo, no nosso quadro de pedidos, que os pedidos que levam carne devem ser, primeiro, preparados pela pessoa responsável pela chapa, enquanto pedidos que sejam compostos apenas de vegetais (como saladas) devem ser preparados pela pessoa responsável pelo corte de legumes e verduras.
Ou seja, o roteamento consiste em "direcionar" uma requisição para que seja atendida por uma determinada parte do nosso sistema.
No Express, nós registramos uma rota utilizando a assinatura `app.METODO(caminho, callback)` , onde a função de callback recebe três parâmetros: `request` , `response` e `next` .

- `request` : comumente chamado de `req` ; contém as informações enviadas pelo cliente ao servidor.
- `response` : geralmente chamado de `res` ; permite o envio de informações do servidor de volta ao cliente.
- `next` : função que diz para o Express que aquele callback terminou de ser executado, e que ele deve prosseguir para executar o próximo callback para aquela rota. Este parâmetro é opcional e você entenderá melhor o uso do next em breve.

As rotas respondem a requisições que satisfaçam a condição método HTTP + caminho .

Copiar

```
const express = require('express');
const app = express();

/* Rota com caminho '/', utilizando o método GET */
app.get('/', function (req, res) {
  res.send('hello world');
});

/* Rota com caminho '/', utilizando o método POST */
app.post('/', function (req, res) {
  res.send('hello world');
});

/* Rota com caminho '/', utilizando o método PUT */
app.put('/', function (req, res) {
  res.send('hello world');
});

/* Rota com caminho '/', utilizando o método DELETE */
app.delete('/', function (req, res) {
  res.send('hello world');
});

/* Rota com caminho '/' para qualquer método HTTP */
app.all('/', function (req, res) {
  res.send('hello world');
```

```
});

/* Ou podemos encadear as requisições para evitar repetir o caminho */
app
  .route('/')
  .get(function (req, res) {
        // Requisições para rota GET `/` são resolvidas aqui!
    res.send('hello world get');
  })
  .post(function (req, res) {
        // Requisições para rota POST `/` são resolvidas aqui!
    res.send('hello world post');
  });

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

## Estruturando uma API

Para entendermos na prática como utilizar o Express e o seu sistema de rotas para criar uma API funcional, vamos partir do seguinte cenário: Temos uma aplicação que permite gerenciar uma lista de receitas disponíveis, com seus respectivos nomes, preço e tempo médio de preparo. Essa aplicação é uma API que implementa CRUD, ou seja, um conjunto de endpoints que permite listar, pesquisar, cadastrar, editar e remover os itens dessa lista de receitas. Até o final do dia, vamos implementar uma API que permite fazer todas essas operações.
Vamos começar implementando o endpoint que retorna a lista de receitas na rota `/recipes` quando a requisição for do tipo `GET` . A lista de receitas virá de uma array que vamos definir no código. Siga o exemplo abaixo:

Copiar

```
/* index.js */
const express = require('express');
const app = express();

const recipes = [
  { id: 1, name: 'Lasanha', price: 40.0, waitTime: 30 },
  { id: 2, name: 'Macarrão a Bolonhesa', price: 35.0, waitTime: 25 },
  { id: 3, name: 'Macarrão com molho branco', price: 35.0, waitTime: 25 },
];

app.get('/recipes', function (req, res) {
  res.json(recipes);
});

app.listen(3001, () => {
  console.log('Aplicação ouvindo na porta 3001');
});
```

Agora, deixamos de usar o método .send para usar o método .json . O método .send é um método que consegue retornar a resposta de uma requisição de uma forma genérica, adaptando o tipo do retorno ao que vai ser retornado. Mas para deixar mais evidente que o que vamos devolver é um JSON usamos o método .json .
Para testar nossa aplicação, podemos fazer uma requisição usando o próprio navegador, colocando a URL http://localhost:3001/recipes . Porém como nem todo tipo de requisição HTTP pode ser feita diretamente pelo navegador, é recomendado utilizar algum cliente HTTP. Os mais famosos são o Postman e o Insomnia .
Existe uma terceira possibilidade: usar um comando chamado httpie que permite fazer uma requisição direto pelo terminal. Para instalar esse comando siga as instruções da documentação . Uma vez instalado, execute o comando abaixo:

Copiar
```
http :3001/recipes
```

Observe que não é preciso colocar a URL completa, já que o HTTPie assume que as requisições são feitas para localhost por padrão. Após rodar o comando você deve conferir que ele vai retornar uma resposta como mostrado abaixo.

Copiar
```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 205
Content-Type: application/json; charset=utf-8
Date: Fri, 20 Aug 2021 22:06:58 GMT
ETag: W/"cd-wMzyMLQRx8RrJ9Bl5KB9X7VuhcA"
Keep-Alive: timeout=5
X-Powered-By: Express

[
    {
        "id": 1,
        "name": "Lasanha",
        "price": 40,
        "waitTime": 30
    },
    {
        "id": 2,
        "name": "Macarrão a Bolonhesa",
        "price": 35,
        "waitTime": 25
    },
    {
        "id": 3,
        "name": "Macarrão com molho branco",
        "price": 35,
        "waitTime": 25
    }
]
```

Ok! Mas o que isso significa de fato? Esse JSON que foi retornado pode ser utilizado por uma aplicação front-end para renderizar essa lista no navegador utilizando o método fetch, que foi utilizado bastante nos nossos exercícios e projetos desde o módulo de fundamentos e principalmente nos projetos de front-end. A diferença é que agora a requisição vai ser feita para uma API que você

mesmo desenvolveu e que roda na sua máquina. A estrutura básica de uma requisição utilizando o fetch pode ser escrita da seguinte forma:

Copiar

```
fetch('http://localhost:3001/recipes')
  .then(resp => resp.json())
```

Para dar mais visibilidade, imagine um componente React que precisa exibir essa lista. Ele ficaria mais ou menos assim:

Copiar

```
class receitasList extends Component {
 constructor(props) {
  super(props);
  this.state = {
    recipes: [],
    isLoading: true,
  };
 }

 componentDidMount() {
  fetch('http://localhost:3001/recipes')
    .then(response => response.json())
    .then((recipes) => this.setState(
      {
        recipes,
        isLoading: false,
      },
    ));
 }

 render() {
  const { recipes, isLoading } = this.state;

  return (
    <div>
      <div>
        {isLoading && <Loading />}
        <div className='card-group'>
          {recipes.map((recipe) => (
            <div key={recipe.id}>
              <h1>{recipe.name}</h1>
              <span>Preço: {recipe.price}</span>
              <span>Tempo de preparo: {recipe.waitTime}</span>
            </div>
          ))}
        </div>
      </div>
    </div>
  );
 }
}
```

⚠️ Observação: Para uma aplicação back-end receber requisições de uma aplicação front-end, ou qualquer outra aplicação, é preciso instalar um módulo que *libera* o acesso da nossa API para outras

aplicações. Para isso basta instalar um módulo chamado cors usando `npm i cors` e adicionar as seguintes linhas no seu arquivo `index.js` .

Copiar

```
// const express = require('express');
// const app = express();
const cors = require('cors');

app.use(cors());
```

Vamos falar um pouco mais sobre esse módulo no conteúdo de amanhã, mas caso deseje testar integração entre front-end e back-end é necessário fazer esse ajuste no código da API.

Para Fixar

1. Crie uma array `drinks` com os seguintes obejtos dentro dela e uma rota `GET /drinks` que retorna a lista de bebidas.

Copiar

```
const drinks = [
    { id: 1, name: 'Refrigerante Lata', price: 5.0 },
    { id: 2, name: 'Refrigerante 600ml', price: 8.0 },
    { id: 3, name: 'Suco 300ml', price: 4.0 },
    { id: 4, name: 'Suco 1l', price: 10.0 },
    { id: 5, name: 'Cerveja Lata', price: 4.5 },
    { id: 6, name: 'Água Mineral 500 ml', price: 5.0 },
];
```

2. Modifique tanto a rota de bebidas como de receitas para retornar a lista ordenada pelo nome em ordem alfabética.

Pronto, já temos uma rota da nossa API que devolve a lista dos receitas disponíveis, mas não precisamos parar por aqui. E se quiséssemos conseguir acessar uma receita específica pelo seu `id` ? ou mesmo procurar por todas os receitas que tem a palavra Macarrão no nome? Além disso, como fazemos para permitir adicionar, editar ou remover receitas da lista através da nossa API?
Tudo isso é o que vamos ver daqui em diante.

## Parâmetros de rota

No caso em que precisamos acessar objetos específicos, o express tem alguns recursos para que conseguimos passar informações além da rota que desejamos buscar. Vamos começar falando de parâmetro de rotas.
Você provavelmente já se deparou com URLs no seguinte formato `http://<site>/noticias/489` ou `http://<site>/pedidos/713` . Esses valores que são passados nas rotas que geralmente devolvem uma página seguindo o mesmo template mas com um conteúdo diferente é implementado graças ao parâmetro de rota. Imagina se para cada nova notícia ou pedido você tivesse que criar uma rota específica exatamente com `/noticias/489` ou `/pedidos/713` ? o trabalho das pessoas desenvolvedoras seria passar o dia inteiro escrevendo rotas. Para facilitar esse processo, utilizamos parâmetros de rota, que no Express, podem ser definidos da seguinte forma: `/<rota>/<:parametro>` onde `:parametro` vai servir para qualquer valor que vier na URL com aquele prefixo específico.

No caso da nossa API de receitas podemos montar uma rota que recebe o id como um parâmetro de rota da seguinte forma:

Copiar

```
// const express = require('express');
// const app = express();
//
// const recipes = [
//   { id: 1, name: 'Lasanha', price: 40.0, waitTime: 30 },
//   { id: 2, name: 'Macarrão a Bolonhesa', price: 35.0, waitTime: 25 },
//   { id: 3, name: 'Macarrão com molho branco', price: 35.0, waitTime: 25 },
// ];
//
// app.get('/recipes', function (req, res) {
//   res.json(recipes);
// });

app.get('/recipes/:id', function (req, res) {
  const { id } = req.params;
  const recipe = recipes.find((r) => r.id === parseInt(id));

  if (!recipe) return res.status(404).json({ message: 'Recipe not found!'});

  res.status(200).json(recipe);
});

// app.listen(3001, () => {
//   console.log('Aplicação ouvindo na porta 3001');
// });
```

No código acima, o que fizemos foi adicionar uma rota `/recipes/:id` , ou seja qualquer rota que chegar nesse formato, independente do id ser um número ou string vai cair nessa segunda rota, em vez de cair na rota `/recipes` que definimos no tópico anterior. Para acessar o valor do parâmetro enviado na URL fizemos a desestruturação do atributo `id` do objeto `req.params` . Começamos a entender que o objeto `req` traz informações a respeito da requisição. É importante que o nome do parâmetro nomeado na rota seja igual ao atributo que você está desestruturando. Por exemplo, se na definição da rota estivesse escrito `/recipes/:nome` teríamos que usar `const { nome } = req.params` .
Na sequência usamos uma função que conhecemos lá no nosso bloco 8 sobre HOF, que é o find. Vamos usar bastante daqui em diante várias HOF, então caso você tenha esquecido ou ainda tenha dúvidas sobre o uso delas, recomendamos fortemente que você revisite o conteúdo do bloco 8 para dar uma revisada sobre o uso dessas funções.
Implementamos uma busca na array `receitas` para encontrar a receita onde o id é igual ao valor que recebemos como parâmetro, tendo o cuidado de converter o valor para inteiro, já que por padrão todo parâmetro de rota é uma string. No final, apenas retornamos o objeto `receita` que corresponde a receita encontrada.
Esse código não está tratando possíveis cenários de erro como por exemplo se o id que chegar na rota for formado por letras. Como nosso foco por enquanto é entendermos como montar a API, vamos falar desses possíveis tratamentos de erros em um momento mais a frente.
⚠️ Atenção: Perceba que na linha com o if colocamos um `return` . Isso serve para indicar para o express que ele deve quebrar o fluxo e não executar a linha `res.status(200).json(recipe);` . Caso você não coloque o `return` , sua requisição vai funcionar mas você vai ver um erro como este abaixo no log do seu terminal:

Copiar

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
```

Esse erro significa que o Express entendeu que você está tentando retornar duas respostas para o cliente. Por isso é preciso ter cuidado para sempre que existir uma condição que pode quebrar o fluxo principal colocar um return antes do `res.json` para não ter esse problema. Este é um erro bem comum para quem está começando a utilizar Express, então caso tenha esse problema, você já sabe o que fazer a partir de agora.
Faça uma requisição para esse endpoint passando um id qualquer.

Copiar
```
http :3001/recipes/1
```

O retorno da nossa requisição será algo parecido com o seguinte conteúdo:

Copiar
```
{
  "id": 1,
  "name": "Lasanha",
  "price": 40,
  "waitTime": 30
}
```

Se passarmos um id que não existe nosso retorno vai ser diferente.

Copiar
```
http :3001/recipes/777
> { message: 'Recipe not found!'}
```

Para Fixar

1. Crie uma rota `GET /drink/:id` para retornar uma bebida pelo `id` .

Entendemos como utilizar parâmetro de rota, mas imagine o cenário em que você quer pesquisar as receitas pelo nome, e eventualmente além de pesquisar pelo nome, ao mesmo tempo para pegar os receitas que sejam no máximo 30 reais. Poderíamos até utilizar o parâmetro de rotas para isso, mas teríamos rotas um pouco mais difíceis de usar pois precisaríamos nos preocupar com a ordem que os parâmetros são organizados e isso diminui a legibilidade das rotas da nossa API. Para isso, existe uma segunda forma de enviar parâmetros através de uma URL, essa forma é conhecida como *query string* .

## Query String

Provavelmente você também já deve ter se deparado com URLs nesse formato `/produtos?q=Geladeira&precoMaximo=2000` . Para pessoas comuns é bem difícil interpretar o que são todas essas letrinhas no final da URL depois do sinal de interrogação. Essa string depois do `?` é uma query string. Nesse caso está sendo passado dois parâmetros: `q` com o valor *Geladeira* e `precoMaximo` com o valor `2000` .
Geralmente o recurso de `query string` é usado em funcionalidades de pesquisas como quando você utiliza além da barra de pesquisa, filtros avançados para definir o preço máximo, marca e outras classificações em e-commerces.
Para nosso exemplo, vamos definir uma rota `/pratos/pesquisar?nome=Macarrão` que permita, inicialmente, buscar uma lista de receitas filtrando pelo nome. Vamos usar o código abaixo.

Copiar
```
// ...
```

```
app.get('/recipes/search', function (req, res) {
  const { name } = req.query;
  const filteredRecipes = recipes.filter((r) => r.name.includes(name));
  res.status(200).json(filteredRecipes);
});


// app.get('/recipes/:id', function (req, res) {
//  const { id } = req.params;
//  const recipe = recipes.find((r) => r.id === parseInt(id));
//  if (!recipe) return res.status(404).json({ message: 'Recipe not found!'});
//
//  res.status(200).send(recipe);
// });

// ...
```

Perceba, que nessa rota, utilizamos `req.query` e desestruturamos o atributo nome, para na sequência usar como parâmetro de busca. Dessa vez usamos uma outra HOF, a função `filter` , para filtrar os receitas que contenham ( `.includes` ) o nome recebido através da query string e no final devolvemos a lista de receitas obtidas por esse filtro.
Note que nossa rota ficou apenas com o prefixo `/recipes/search` já que os parâmetros enviados query string, não dependem desse prefixo e sim das informações que vem após o uso da `?` na URL. É importante entender que na URL podemos colocar quantos parâmetros desde que eles sigam o formato `<chave>=<valor>` e que entre cada parâmetro, exista o `&` para definir que ali está sendo passado um novo parâmetro.
⚠️ Observação: Quando houver rotas com um mesmo radical e uma destas utilizar parâmetro de rota, as rotas mais específicas devem ser definidas sempre antes. Isso porque o Express ao resolver uma rota vai olhando rota por rota qual corresponde a URL que chegou. Se colocamos a rota `/recipes/search` depois da rota `/recipes/:id` , o Express vai entender que a palavra `search` como um id e vai chamar a callback da rota `/recipes/:id` . Tenha atenção a esse detalhe ao utilizar rotas similares na definição da sua API.
Faça a requisição para testar esse nosso novo endpoint.

Copiar

```
http :3001/recipes/search name==Macarrão # Para testar pelo navegador use a URL completa

http://localhost:3001/recipes/search?name=Macarrão
```

A resposta da nossa API vai ser essa:

Copiar

```
[
    {
        "id": 2,
        "name": "Macarrão a Bolonhesa",
        "price": 35,
        "waitTime": 25
    },
    {
        "id": 3,
        "name": "Macarrão com molho branco",
        "price": 35,
        "waitTime": 25
    }
]
```

Vamos agora refatorar nosso código para que ele também seja capaz de filtrar pelo preço máximo passando um segundo parâmetro através da query string.

Copiar

```
// ...

app.get('/recipes/search', function (req, res) {
    const { name, maxPrice } = req.query;
    const filteredRecipes = recipes.filter((r) => r.name.includes(name) && r.price < parseInt(maxPrice));
    res.status(200).json(filteredRecipes);
})

// ...
```

Não foi preciso alterar a definição da rota, apenas no código do callback foi feita uma alteração para desestruturar também o atributo `maxPrice` do objeto `req.query` e foi adicionada uma condição na chamada da função `filter` para filtrar os objetos pelo nome e pelo valor do atributo `maxPrice` enviado na requisição. Vamos testar nosso endpoint depois da modificação. Você pode testar usando no navegador a URL http://localhost:3001/recipes/search?name=Macarrão &maxPrice=40 ou usando o HTTPie como no exemplo abaixo.

Copiar

```
http :3001/recipes/search name==Macarrão maxPrice==40 # nesse caso do HTTPie não é necessário usar o &
```

O retorno da nossa requisição vai devolver a seguinte resposta:

Copiar

```
[
    {
        "id": 2,
        "name": "Macarrão a Bolonhesa",
        "price": 35,
        "waitTime": 25
    },
    {
        "id": 3,
        "name": "Macarrão com molho branco",
        "price": 35,
        "waitTime": 25
    }
]
```

## Para Fixar

1. Modifique o código da nossa rota para que ela receba um terceiro parâmetro através de query string com o atributo `minPrice` e modifique o filter para trazer apenas os receitas onde o valor da receita seja maior ou igual ao o valor enviado como parâmetro, mantendo os filtros anteriores.
2. Implemente uma rota `/drinks/search` que permita pesquisar pelo atributo `name` usando query string.

Nosso próximo passo é entender como conseguir receber informações uma forma segura quando precisamos persistir informações, ou seja, quando precisarmos salvar dados do lado do back-end. Não será usado query string e sim o `body` da requisição.

## Recebendo dados no body da requisição.

Toda requisição HTTP, possui um cabeçalho e um corpo, como foi mencionado anteriormente. Mas o que isso significa na prática?
Acabamos de ver que é possível receber dados da URL, via query string, porém vamos imaginar que precisamos salvar dados sensíveis como uma senha, número de algum documento importante. Se enviarmos isso na URL qualquer pessoa que conseguir espiar o tráfego de rede entre o cliente e o servidor vai ter acesso a essas informações. Uma forma que o protocolo HTTP encontrou para resolver isso foi criando o tráfego através do corpo da requisição, praticamente o que acontece é uma compressão dos dados enviados que só serão descomprimidos do lado do back-end. Isso além de não deixar as informações trafegadas tão expostas acaba deixando a requisição um pouco mais rápida já que ocorre um processo de serialização dos dados enviados. Porém aqui cabe um detalhe, geralmente para enviar dados no body da requisição você precisa usar algum tipo específico de requisição, como por exemplo, utilizando o verbo HTTP `POST` . Até então só vimos exemplos de rotas mapeadas para o verbo `GET` , Vamos ver como fica agora para esse novo verbo.
Antes disso, precisamos fazer uma pequena etapa que é instalar o pacote `bodyParser` . Como explicamos, os dados enviados pelo front-end são comprimidos, e para que conseguimos remontar os dados enviados precisamos *parsear* as informações para um formato compreensível para o back-end, esse formato no caso vai ser JSON. Para instalar esse pacote execute o comando:

Copiar
```
npm i body-parser
```

Agora no arquivo `index.js` , faça a modificação abaixo:

Copiar
```
// const express = require('express');
const bodyParser = require('body-parser');

// const app = express();
app.use(bodyParser.json());

// ...
```

Agora vamos implementar nossa rota que vai receber dados no body da requisição.

Copiar
```
// ...
app.post('/recipes', function (req, res) {
  const { id, name, price } = req.body;
  recipes.push({ id, name, price});
  res.status(201).json({ message: 'Recipe created successfully!'});
});
```

Perceba, que repetimos a rota `/recipes` , só que agora usando o método `.post` . No Express, é possível ter rotas com o mesmo caminho desde que o método (ou verbo) HTTP utilizado seja diferente, na outra rota definimos o que acontece para o método `GET` . Por falar nisso, fica a pergunta, como vamos conseguir fazer requisições já que por padrão as requisições que fazemos ou no navegador ou no *fetch api* são do tipo `GET` ?

Vamos começar pelo `fetch-api` , usando o código abaixo.

Copiar

```
fetch(`http://localhost:3001/recipes/`, {
  method: 'POST',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    id: 4,
    title: 'Macarrão com Frango',
    price: 30
  })
});
```

Diferente do que fizemos para fazer uma requisição do tipo `GET` , dessa vez passamos um segundo parâmetro que é um objeto formado pelos atributos `method` , `headers` , `body` . Vamos entender o que é cada um.

- `method` : Método HTTP utilizado, como vimos no primeiro bloco temos 4 que são mais utilizados (GET, POST, PUT e DELETE).
- `headers` : Define algumas informações sobre a requisição como o atributo `Accept` que diz qual o tipo de dado esperado como resposta dessa requisição e o `Content-Type` que sinaliza que no corpo da requisição está sendo enviado um JSON.
- `body` : É o corpo da requisição. Como no HTTP só é possível trafegar texto, é necessário transformar o objeto JavaScript que quermos enviar para uma string JSON. Por isso que do lado do nosso back-end precisamos utilizar o `bodyParser` para transformar as informações que foram trafegadas como string JSON, de volta para um objeto JavaScript.

Não é possível fazer requisições POST diretamente pelo navegador como fizemos para requisição para rota `GET /recipes` . Por isso devemos utilizar aplicações como o Insomnia ou Postman para fazer requisições de qualquer tipo diferente do GET. Vamos usar o HTTPie para executar nossa requisição.

Copiar

```
http POST :3001/recipes id:=4 name='Macarrão com Frango' price:=30 // execute apenas essa linha!
> HTTP/1.1 201 Created
> Connection: keep-alive
> Content-Length: 32
> Content-Type: application/json; charset=utf-8
> Date: Sat, 21 Aug 2021 19:26:46 GMT
> ETag: W/"20-bnfMbzwQ0XaOf5RuS+0mkUwjAeU"
> Keep-Alive: timeout=5
> X-Powered-By: Express
>
> {
>     "message": "Recipe created successfully!"
> }
```

Nos campos id e preco usamos `:=` enquanto em nome colocamos apensas `=` . Fazemos isso, pois o operador `=` envia os dados como string, enquanto com `:=` o dado é enviado como número.
⚠️ Observação: Como estamos trabalhando com a lista de receitas através de uma array, sempre que nossa aplicação é reiniciada, a array volta ao formato original, com os 3 objetos que definimos direto no código. Portanto, caso as receitas que você cadastrou sumam repentinamente da listagem, provavelmente foi por essa causa, os dados estão apenas armazenados em memória.

Vamos voltar para nosso código para entender a implementação.

Copiar

```
// ...

app.post('/recipes', function (req, res) {
  const { id, name, price } = req.body;
  recipes.push({ id, name, price});
  res.status(201).json({ message: 'Recipe created successfully!'});
});

// ...
```

Na primeira linha desestruturamos os atributos `id` , `price` e `preco` do objeto `req.body` para na segunda linha usarmos esses valores para inserir um novo objeto dentro da array `receitas` . Na terceira e última linha retornamos uma resposta com o status 201, que serve para sinalizar que ocorreu uma operação de persistência de uma informação e um json com o atributo `message` . Pronto, temos uma rota que permite cadastrar novos receitas na nossa array.
E o headers?
Assim como podemos enviar informações no body da requisição, também é possível enviar informações no header da mesma. Vamos imaginar que precisamos ter uma rota que recebe um token para ser validado, a regra da validação é checar se o token possui 16 caracteres.

Copiar

```
// ...

app.get('/validateToken', function (req, res) {
  const token = req.headers.authorization;
  if (token.length !== 16) return res.status(401).json({message: 'Invalid Token!'})';

  res.status(200).json({message: 'Valid Token!'})'
});

// ...
```

Para fazer uma request enviando informações no headers, utilizando o HTTPie podemos usar o seguinte comando:

Copiar

```
http :3001/validateToken Authorization:abc # vai devolver token inválido
http :3001/validateToken Authorization:S6xEzQUTypw4aj5A # vai devolver token válido
```

Diferente de informações enviadas no corpo da requisição que usavam `=` ou `:=` para determinar o valor de um atributo, definimos atributos do headers usando `:` , passamos a chave `Authorization` que é uma chave bem comum de se utilizar nesse tipo de autenticação. No conteúdo de amanhã teremos exemplos mais práticos sobre o uso de headers. Por enquanto é mais uma forma de enviar dados para nossa API.
Veja o vídeo abaixo que consolida parte do nosso aprendizado até aqui ou se preferir avançe para a próxima seção.

## Para Fixar

1. Crie uma rota `POST /drinks` que permita adicionar novas bebidas através da nossa API.
2. Modifique o código acima da rota `POST /recipes` para que receba e salve a receita com o atributo `waitTime` .

Com isso, já temos metade de um CRUD implementado. Já conseguimos Criar (Create) e Ler (Read) dados através da nossa API, por mais que seja de uma forma mais simples, lendo e salvando em uma array, isso já é o suficiente para termos uma primeira noção de como funciona algumas coisas do Express e para que serve alguns verbos HTTP, além de revisar algumas funções que aprendemos lá no bloco sobre HOFs.
Para finalizar o dia vamos entender como Atualizar (Update) e Remover(Delete) dados através da nossa API, além de lidar com rotas não mapeadas.

## Atualizando e deletando objetos através da API

Além dos métodos GET E POST , o HTTP também possui os métodos PUT e DELETE que são convencionalmente utilizados para rotas que, respectivamente, editam e removem objetos. O Express tem métodos específicos para definir rotas para esses dois verbos. Vamos começar dando um exemplo do uso do PUT .

Copiar

```
// ...

app.put('/recipes/:id', function (req, res) {
  const { id } = req.params;
  const { name, price } = req.body;
  const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipeIndex === -1) return res.status(404).json({ message: 'Recipe not found!' });

  recipes[recipeIndex] = { ...recipes[recipeIndex], name, price };

  res.status(204).end();
});
// ...
```

Observe que definimos uma rota que recebe o id como parâmetro de rota, e os campos nome e preço através do body da requisição. É um padrão sempre mandar o id como parâmetro de rota e os atributos que vão ser alterados, no body, pois é uma boa prática do RESTful, conteúdo que vamos ver mais a frente. Depois apenas usamos o método find para encontrar a receita correspondente ao id, e atualizamos os atributos para os valores recebidos. Por fim, devolvemos uma resposta HTTP com o status 204, que serve para indicar que algo foi atualizado e utilizamos o método `.end()` que indica que a resposta vai ser retornada sem retornar nenhuma informação
Vamos fazer essa requisição usando o HTTPie.

Copiar

```
http PUT :3001 /recipes/2 name='Macarrão ao alho e óleo' price:=40 # execute apenas essa linha!
> HTTP/1.1 204 No Content
> Connection: keep-alive
> Date: Fri, 20 Aug 2021 22:19:35 GMT
> ETag: W/"25-ySvLeHwVHESCh2r//vitH6caTaI"
> Keep-Alive: timeout=5
> X-Powered-By: Express
```

Como utilizamos o `.end()` no nosso callback da rota `PUT /recipes/:id` não retornamos nada, apenas o status 204, que indica que a requisição foi completada com sucesso.
Agora é a vez de implementarmos uma rota que permita remover receitas da nossa lista. Para isso vamos criar uma rota para requisições do tipo `DELETE` no caminho `/recipes/:id` .

Copiar

```
//...

app.delete('/recipes/:id', function (req, res) {
  const { id } = req.params;
  const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipeIndex === -1) return res.status(404).json({ message: 'Recipe not found!' });

  recipes.splice(recipeIndex, 1);

  res.status(204).end();
});

//...
```

Note que novamente utilizamos o id como parâmetro de rota. Essa é uma convenção que como vimos, serve para sempre que precisamos trabalhar com id seja para pesquisar, editar e remover objetos através da nossa API. É possível fazer a mesma coisa enviando o id como query string ou no body da requisição, mas usar parâmetro de rota acaba sendo a forma mais simples de mandar esse tipo de dado entre todas as opções disponíveis.
Vamos fazer uma requisição usando o HTTPie novamente.

Copiar

```
http DELETE :3001 /recipes/3 # execute apenas essa linha!
> HTTP/1.1 204 No Content
> Connection: keep-alive
> Date: Fri, 20 Aug 2021 22:25:44 GMT
> ETag: W/"23-nD7qnlOhswfi0qOrye68khRdynU"
> Keep-Alive: timeout=5
> X-Powered-By: Express
```

Novamente por termos usado o status HTTP 204, a resposta da nossa requisição vem sem trazer um conteúdo. Tudo bem, já que o objetivo dessa rota é apenas excluir um registro da nossa array de receitas. Teste fazer a requisição para listar os receitas e você vai conferir que a receita com id 3 realmente foi removido.
No front-end, para fazer requisições do tipo PUT e DELETE através do fetch api podemos utilizar os exemplos de código abaixo:

Copiar

```
// Requisição do tipo PUT
fetch(`http://localhost:3001/recipes/2`, {
  method: 'PUT',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'Macarrão ao alho e óleo',
    price: 40
  })
});

// Requisição do tipo DELETE
fetch(`http://localhost:3001/recipes/4`, {
```

```
  method: 'DELETE',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  }
});
```

### Para Fixar

1. Crie uma rota `PUT /drinks/:id` que permita editar os atributos de uma bebida.
2. Crie uma rota `DELETE /drinks/:id` que permita remover uma bebida.

### O que acontece se fizermos uma requisição para uma rota que não existe?

Se tentarmos fazer uma requisição para uma rota que não mapeamos na noss API, você vai observar que o Express retorna a seguinte resposta.

Copiar

```
http GET :3001/xablau
> <!DOCTYPE html>
> <html lang="en">
> <head>
> <meta charset="utf-8">
> <title>Error</title>
> </head>
> <body>
> <pre>Cannot GET /xablau</pre>
> </body>
> </html>
```

Porém essa não é uma forma muito compreensível de entender que essa rota `/xablau` não foi mapeada. Para retornar uma resposta mais amigável podemos usar o método `app.all` da seguinte forma:

Copiar

```
//...
app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

app.listen(3001);
```

Agora se tentarmos acessar uma requisição para uma rota não mapeada vamos ter a seguinte resposta.

Copiar

```
{
    "message": "Rota '/xablau' não existe!"
}
```

O método `app.all` serve para mapear uma rota que pode ser de qualquer verbo HTTP e o valor `*` é um `wildcard` , ou seja um expressão coringa que indica que indepedente da rota que chegar aqui ele vai capturar e executar a callback que por sua vez retorna uma resposta com status `404` .

⚠️ Cuidado: Essa definição de rota generalista com `app.all('*')` deve ser sempre a última definição de rota da nossa API. Caso o contrário algumas requisições podem cair antes neste callback e não executar o callback para a rota específica. Para exemplificar vamos definir um callback para responder a rota `/xablau` .

Copiar

```
//...
app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

// nunca vai chegar nessa rota!
app.get('/xablau', function (req, res) {
    return res.status(404).json({ message: `Xablau!`});
});

app.listen(3001);
```

Se você fizer a requisição com o código acima vai ver que o Express vai continuar a trazer a mesma resposta `"Rota '/xablau' não existe!"` . Agora inverta as duas definições de rotas de lugar e observe que a resposta retornada passa a ser a correta.

Copiar

```
//...

// agora vai chegar nessa rota!
app.get('/xablau', function (req, res) {
    return res.status(404).json({ message: `Xablau!`});
});

app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

app.listen(3001);
```

Copiar

```
http :3001/xablau
> {
>     "message": "Rota '/rota' não existe!"
> }
```

## Atualizando e deletando objetos através da API

Além dos métodos GET E POST , o HTTP também possui os métodos PUT e DELETE que são convencionalmente utilizados para rotas que, respectivamente, editam e removem objetos. O Express tem métodos específicos para definir rotas para esses dois verbos. Vamos começar dando um exemplo do uso do PUT .

Copiar

```
// ...

app.put('/recipes/:id', function (req, res) {
  const { id } = req.params;
```

```
  const { name, price } = req.body;
  const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipeIndex === -1) return res.status(404).json({ message: 'Recipe not found!' });

  recipes[recipeIndex] = { ...recipes[recipeIndex], name, price };

  res.status(204).end();
});
// ...
```

Observe que definimos uma rota que recebe o id como parâmetro de rota, e os campos nome e preço através do body da requisição. É um padrão sempre mandar o id como parâmetro de rota e os atributos que vão ser alterados, no body, pois é uma boa prática do RESTful, conteúdo que vamos ver mais a frente. Depois apenas usamos o método find para encontrar a receita correspondente ao id, e atualizamos os atributos para os valores recebidos. Por fim, devolvemos uma resposta HTTP com o status 204, que serve para indicar que algo foi atualizado e utilizamos o método `.end()` que indica que a resposta vai ser retornada sem retornar nenhuma informação
Vamos fazer essa requisição usando o HTTPie.

Copiar

```
http PUT :3001 /recipes/2 name='Macarrão ao alho e óleo' price:=40 # execute apenas essa linha!
> HTTP/1.1 204 No Content
> Connection: keep-alive
> Date: Fri, 20 Aug 2021 22:19:35 GMT
> ETag: W/"25-ySvLeHwVHESCh2r//vitH6caTaI"
> Keep-Alive: timeout=5
> X-Powered-By: Express
```

Como utilizamos o `.end()` no nosso callback da rota `PUT /recipes/:id` não retornamos nada, apenas o status 204, que indica que a requisição foi completada com sucesso.
Agora é a vez de implementarmos uma rota que permita remover receitas da nossa lista. Para isso vamos criar uma rota para requisições do tipo `DELETE` no caminho `/recipes/:id` .

Copiar

```
//...

app.delete('/recipes/:id', function (req, res) {
  const { id } = req.params;
  const recipeIndex = recipes.findIndex((r) => r.id === parseInt(id));

  if (recipeIndex === -1) return res.status(404).json({ message: 'Recipe not found!' });

  recipes.splice(recipeIndex, 1);

  res.status(204).end();
});

//...
```

Note que novamente utilizamos o `id` como parâmetro de rota. Essa é uma convenção que como vimos, serve para sempre que precisamos trabalhar com id seja para pesquisar, editar e remover objetos através da nossa API. É possível fazer a mesma coisa enviando o id como query string ou no body da requisição, mas usar parâmetro de rota acaba sendo a forma mais simples de mandar esse tipo de dado entre todas as opções disponíveis.

Vamos fazer uma requisição usando o HTTPie novamente.

Copiar

```
http DELETE :3001 /recipes/3 # execute apenas essa linha!
> HTTP/1.1 204 No Content
> Connection: keep-alive
> Date: Fri, 20 Aug 2021 22:25:44 GMT
> ETag: W/"23-nD7qnlOhswfi0qOrye68khRdynU"
> Keep-Alive: timeout=5
> X-Powered-By: Express
```

Novamente por termos usado o status HTTP 204, a resposta da nossa requisição vem sem trazer um conteúdo. Tudo bem, já que o objetivo dessa rota é apenas excluir um registro da nossa array de receitas. Teste fazer a requisição para listar os receitas e você vai conferir que a receita com id 3 realmente foi removido.
No front-end, para fazer requisições do tipo PUT e DELETE através do fetch api podemos utilizar os exemplos de código abaixo:

Copiar

```
// Requisição do tipo PUT
fetch(`http://localhost:3001/recipes/2`, {
  method: 'PUT',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    name: 'Macarrão ao alho e óleo',
    price: 40
  })
});

// Requisição do tipo DELETE
fetch(`http://localhost:3001/recipes/4`, {
  method: 'DELETE',
  headers: {
    Accept: 'application/json',
    'Content-Type': 'application/json',
  }
});
```

### Para Fixar

1. Crie uma rota PUT /drinks/:id que permita editar os atributos de uma bebida.
2. Crie uma rota DELETE /drinks/:id que permita remover uma bebida.

### O que acontece se fizermos uma requisição para uma rota que não existe?

Se tentarmos fazer uma requisição para uma rota que não mapeamos na noss API, você vai observar que o Express retorna a seguinte resposta.

Copiar

```
http GET :3001/xablau
> <!DOCTYPE html>
```

```
> <html lang="en">
> <head>
> <meta charset="utf-8">
> <title>Error</title>
> </head>
> <body>
> <pre>Cannot GET /xablau</pre>
> </body>
> </html>
```

Porém essa não é uma forma muito compreensível de entender que essa rota `/xablau` não foi mapeada. Para retornar uma resposta mais amigável podemos usar o método `app.all` da seguinte forma:

Copiar

```
//...
app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

app.listen(3001);
```

Agora se tentarmos acessar uma requisição para uma rota não mapeada vamos ter a seguinte resposta.

Copiar

```
{
    "message": "Rota '/xablau' não existe!"
}
```

O método `app.all` serve para mapear uma rota que pode ser de qualquer verbo HTTP e o valor `*` é um `wildcard` , ou seja um expressão coringa que indica que indepedente da rota que chegar aqui ele vai capturar e executar a callback que por sua vez retorna uma resposta com status `404` .
⚠️ Cuidado: Essa definição de rota generalista com `app.all('*')` deve ser sempre a última definição de rota da nossa API. Caso o contrário algumas requisições podem cair antes neste callback e não executar o callback para a rota específica. Para exemplificar vamos definir um callback para responder a rota `/xablau` .

Copiar

```
//...
app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

// nunca vai chegar nessa rota!
app.get('/xablau', function (req, res) {
    return res.status(404).json({ message: `Xablau!`});
});

app.listen(3001);
```

Se você fizer a requisição com o código acima vai ver que o Express vai continuar a trazer a mesma resposta `"Rota '/xablau' não existe!"` . Agora inverta as duas definições de rotas de lugar e observe que a resposta retornada passa a ser a correta.

Copiar

```
//...
```

```
// agora vai chegar nessa rota!
app.get('/xablau', function (req, res) {
    return res.status(404).json({ message: `Xablau!`});
});

app.all('*', function (req, res) {
    return res.status(404).json({ message: `Rota '${req.path}' não existe!`});
});

app.listen(3001);
```

Copiar

```
http :3001/xablau
> {
>     "message": "Rota '/rota' não existe!"
> }
```

# Exercícios

Hora de pôr a mão na massa!
back-end

### Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma branch, com arquivos e commits específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

```
$ git checkout main
```

6. `$ git pull`
7. A partir da main, crie uma branch com o nome exercicios/26.4 *(bloco 26, dia 4)*
8. Copiar
9. `$ git checkout -b exercicios/26.4`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

```
$ mkdir back-end
```

12. `$ cd back-end`

13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

```
$ mkdir bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
```

15. $ cd bloco-26-introducao-ao-desenvolvimento-web-com-nodejs
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-4-express-http-com-nodejs
```

18. $ cd dia-4-express-http-com-nodejs
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório ~/trybe-exercicios/back-end/block-26-introducao-ao-desenvolvimento-web-com-nodejs/dia-4-express-http-com-nodejs. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/26.4
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

21. modified:   exercicio-1

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
para não adicionar arquivos indesejados acidentalmente
```

23. $ git add --all

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. $ git commit -m "Mensagem descrevendo alterações"
26. Você pode visualizar o log de todos os commits já feitos naquela branch com git log.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD -> exercicios/26.4)
```

```
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o alert e coloquei pra quando clicar aparecer uma imagem do lado direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300

    Resolvendo o exercício 1 usando eventListener
```

29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch exercicios/26.4 não vai existir no repositório remoto, então precisamos configurar o remote utilizando a opção --set-upstream (ou -u, que é a forma abreviada).
30. Copiar
31. 
```
$ git push -u origin exercicios/26.4
```
32. Após realizar o passo 9, podemos abrir a Pull Request a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com -u anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34. 
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch exercicios/26.4, e disponíveis no repositório remoto do GitHub. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊🤜

---

## Agora a prática

Inicie os exercícios criando uma aplicação Node.js com os comandos já aprendidos.

1. Crie uma rota `GET /ping`
    1. Sua rota deve retornar o seguinte JSON: `{ message: 'pong' }`
2. Crie uma rota `POST /hello`
    1. Sua rota deve receber, no body da requisição, o seguinte JSON: `{ "name": "<nome do usuário>" }`
    2. Sua rota deve retornar o seguinte JSON: `{ "message": "Hello, <nome do usuário>!" }` .
3. Crie uma rota `POST /greetings`
    1. Sua rota deve receber o seguinte JSON: `{ "name": "<nome do usuário>", "age": <idade do usuário> }` .
    2. Caso a pessoa usuária tenha idade superior a 17 anos, devolva o JSON `{ "message": "Hello, <nome do usuário>!" }` com o status code `200 - OK` .
    3. Caso a pessoa usuária tenha 17 anos ou menos, devolva o JSON `{ "message": "Unauthorized" }` com o status code `401 - Unauthorized` .
4. Crie uma rota `PUT /users/:name/:age` .
    1. Sua rota deve retornar o seguinte JSON: `{ "message": "Seu nome é <name> e você tem <age> anos de idade" }` .
5. Crie uma API de dados das personagens de Simpsons

- Crie um arquivo chamado `simpsons.json` e popule com os seguintes dados:

Copiar

```
[
  {
    "id": "1",
    "name": "Homer Simpson"
  },
  {
    "id": "2",
    "name": "Marge Simpson"
  },
  {
    "id": "3",
    "name": "Bart Simpson"
  },
  {
    "id": "4",
    "name": "Lisa Simpson"
  },
  {
    "id": "5",
    "name": "Maggie Simpson"
  },
  {
    "id": "6",
    "name": "Ned Flanders"
  },
  {
    "id": "7",
    "name": "Montgomery Burns"
```

```
  },
  {
    "id": "8",
    "name": "Nelson Muntz"
  },
  {
    "id": "9",
    "name": "Krusty"
  },
  {
    "id": "10",
    "name": "Milhouse Van Houten"
  }
]
```

- Utilize o modulo `fs` do Node para ler/escrever arquivos.
- Caso algum erro ocorra, deve ser retornado um código 500 (Internal Server Error).
- Caso dê tudo certo, a resposta deve voltar com status 200 OK .
- Para testar sua API durante o desenvolvimento, utilize ferramentas que permitem fazer requisições HTTP, como Postman , Insomnia ou httpie .

6. Crie um endpoint `GET /simpsons`
    1. O endpoint deve retornar um array com todos os simpsons.
7. Crie um endpoint `GET /simpsons/:id`
    1. O endpoint deve retornar o personagem com o `id` informado na URL da requisição.
    2. Caso não exista nenhum personagem com o `id` especificado, retorne o JSON `{ message: 'simpson not found' }` com o status `404 - Not Found` .
8. Crie um endpoint `POST /simpsons` .
    1. Este endpoint deve cadastrar novos personagens.
    2. O corpo da requisição deve receber o seguinte JSON: `{ id: <id-da-personagem>, name: '<nome-da-personagem>' }` .
    3. Caso já exista uma personagem com o id informado, devolva o JSON `{ message: 'id already exists' }` com o status `409 - Conflict` .
    4. Caso a personagem ainda não exista, adicione-a ao arquivo simpsons.json e devolva um body vazio com o status `204 - No Content` . Para encerrar a request sem enviar nenhum dado, você pode utilizar `res.status(204).end();` .

## Bônus

1. Adicione autenticação a todos os endpoints.
    1. O token deve ser enviado através do header `Authorization` .
    2. O token deve ter exatamente 16 caracteres.
    3. Caso o token seja inválido ou inexistente, a resposta deve possuir o status `401 - Unauthorized` e o JSON `{ message: 'Token inválido!' }` .
2. Crie uma rota `POST /signup`
    1. A rota deve receber, no body da requisição, os campos `email` , `password` , `firstName` e `phone` .
    2. Caso algum dos campos não esteja preenchido, a response deve possuir status `401 - Unauthorized` e o JSON `{ message: 'missing fields' }` .

3. Caso todos os parâmetros estejam presentes, a rota deve gerar um token aleatório válido, e a resposta deve conter o status `200 - OK` , e o JSON `{ token: '<token-aleatorio>' }` .

- Para gerar o token você pode utilizar a função `randomBytes` , do módulo `crypto` do Node, dessa forma:

Copiar

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

function generateToken() {

  return crypto.randomBytes(8).toString('hex');

}

module.exports = generateToken;
```

---