

2020

Express direto ao ponto

APOSTILA BACKEND COM EXPRESS + MONGODB
JÔNATAS FERNANDES PIMENTA

Sumário

Parte 1: Criando e configurando o servidor	2
Parte 2: Criando as rotas e entendendo os métodos HTTP	3
Parte 3: Estruturando o projeto.....	9
Parte 4: O banco de dados.....	11
Bibliografia:	16

NodeJS

O que precisa ter instalado: NodeJS LTS, Insomnia Core e VsCode

Links para instalação: <https://nodejs.org/en/download/>, <https://insomnia.rest/download/>, <https://code.visualstudio.com/download>

Parte 1: Criando e configurando o servidor

Primeiro, você precisa criar uma pasta, pode ser em qualquer lugar. Dentro dela abra o CMD e digite o comando: `npm init -y`. Isso irá gerar os arquivos de pacotes do servidor, sempre que for criar um projeto novo você tem que executar esse comando. Agora precisamos instalar o Express, que será nossa ferramenta para a criação de nossa API, para isso basta digitar no CMD: `npm i express`. Agora finalmente iremos criar nosso arquivo, crie um arquivo chamado **app.js** e depois abra a pasta do seu projeto no VsCode. Nosso primeiro passo será importar o Express dentro de nosso projeto:

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

Nesse trecho de código eu armazenei o express dentro de um constante chamada express, sempre que quiser fazer a importação de um pacote ou até mesmo arquivo, será essa mesma estrutura: `const nome = require('nome_do_pacote_ou_diretorio');`

Agora precisamos “invocar” o express:

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

Na linha 2 simplesmente armazenei o express dentro da constante `app`.

Bem, agora precisamos configurar esse servidor, precisamos colocar ele para ficar escutando uma porta (na pratica vai ser melhor de entender). A porta pode ser qualquer valor de até 4 dígitos, porém, eu recomendo a porta 3000:

```

1  const express = require('express');
2  const app = express();
3
4  app.listen(3000);
5

```

Nosso servidor já estará funcionando se rodarmos ele, vamos fazer o teste. Instale o pacote nodemon escrevendo *npm i nodemon* no terminal, agora no terminal digite *nodemon app.js*. Como vocês podem ver, não retornou nenhum erro no terminal

```

C:\Users\DESK\Desktop\>nodemon app.js
[nodemon] 2.0.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching dir(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`

```

Deixe este terminal rodando sempre. O nodemon é um pacote que inicia o nosso servidor com hot refresh, que no caso, toda vez que salvarmos o arquivo, o servidor reinicia automaticamente, aí não precisamos ficar fechando e rodando o servidor a cada alteração.

Parte 2: Criando as rotas e entendendo os métodos HTTP

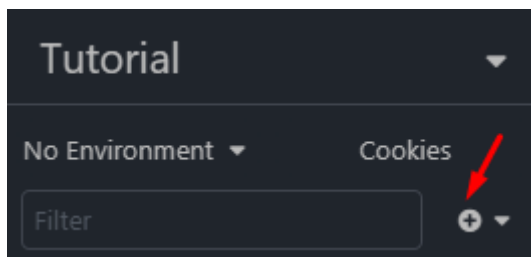
Vamos criar nossa primeira rota:

```

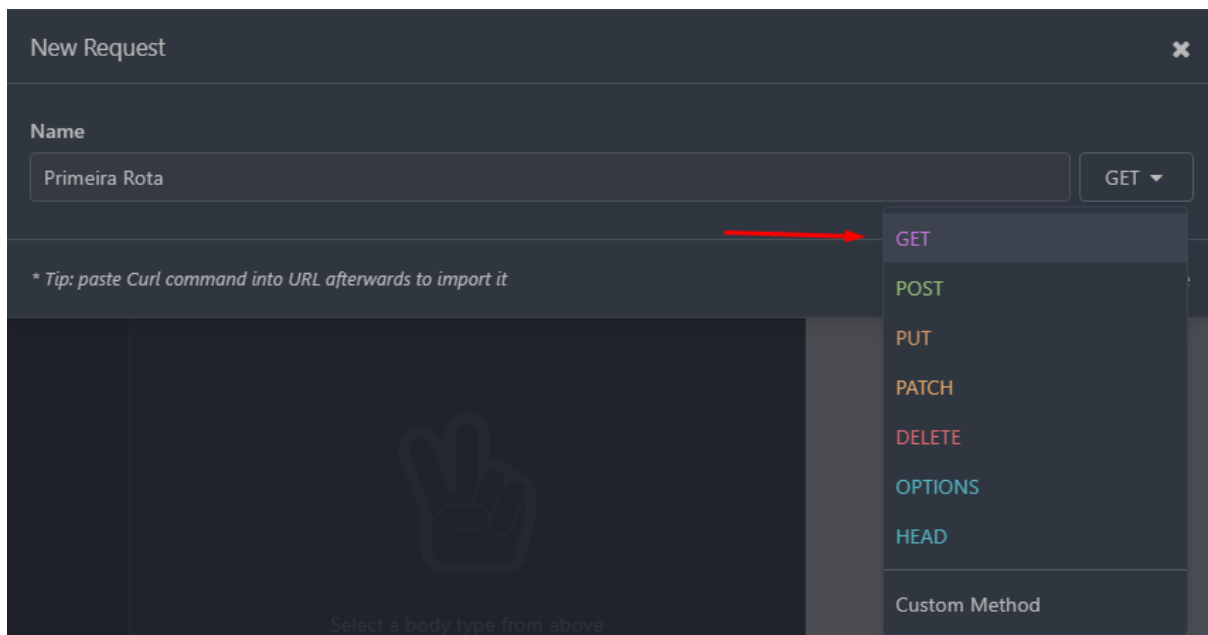
1  const express = require('express');
2  const app = express();
3
4  app.get('/', function(req, res) {
5    res.send('Olá mundo');
6  });
7
8  app.listen(3000);

```

Nesse trecho de código, criei a rota "/" que é do tipo GET (explicarei depois), passei uma função com os parâmetros req e res. O req é um objeto de requisição e o res é um objeto de retorno e também coloquei o *res.send()* que irá retornar para nós o "Olá mundo". Você pode testar essa rota no Insomnia. Dentro do Insomnia, crie uma nova request:



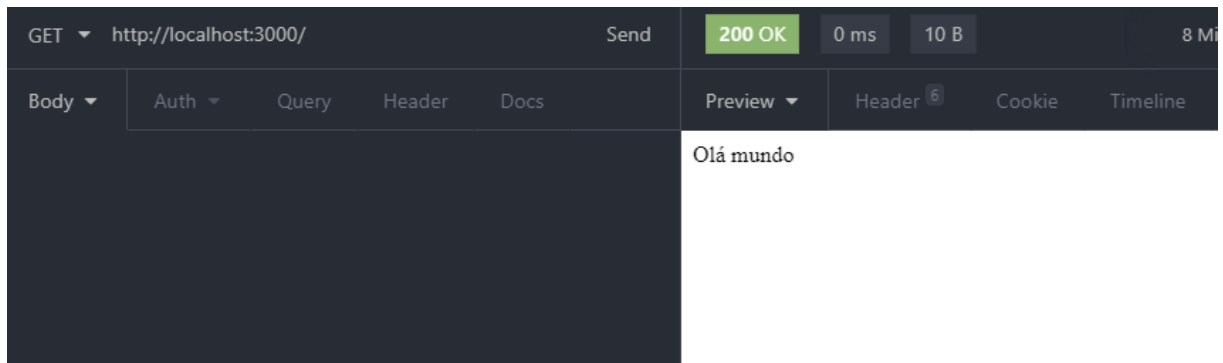
Pode dar qualquer nome e selecione o tipo GET:



Depois que criar a rota, clique nela e você vai ver um campo para colocar uma URL



A rota que colocaremos é <http://localhost:3000/>. A estrutura dessas URL será sempre [http://localhost:porta do servidor/rota](http://localhost:porta_do_servidor/rota). No caso nossa porta é 3000 e a nossa rota é simplesmente “/” se fosse “/home” seria só inserir o “/home” na URL. Quando você apertar Send verá que irá aparecer o nosso “Olá mundo” na parte branca do Insomnia.



Voltando a falar do GET agora. Existem vários métodos HTTP, os que mais usamos na criação das rotas é: GET, POST, PUT e DELETE

GET	Nessa requisição, solicitamos a representação de um recurso, pode ser um HTML, JSON, etc...
POST	Utilizado quando queremos criar um recurso. Nesse método os dados vão ao corpo da requisição e não na URL
PUT	Usamos para substituir/atualizar um recurso
DELETE	Exclui o recurso especificado

Tudo isso na pratica vai ficar mais simples. Vamos criar uma rota do tipo POST logo abaixo de nossa rota anterior:

```
app.post('/enviar', function(req, res) {  
  })
```

Antes de criarmos essa rota, temos que entender o que é um JSON, pois todas nossas requisições nas rotas POST e as solicitações nas rotas GET serão em arquivos JSON.

JSON é um Objeto de Notação JavaScript (JavaScript Object Notation), o JSON não é nada mais nada menos do que um formato de troca de dados simples e rápida entre

sistemas. Ele é bem legível por humanos e também por máquinas. Aqui está um exemplo de sua sintaxe:

```
{
  "filme": "Senhor dos anéis",
  "genero": "Fantasia",
  "ano": "2002"
}
```

Também podemos passar objetos dentro dos objetos:

```
{
  "Filmes": {
    "filme": "Senhor dos anéis",
    "genero": "Fantasia",
    "ano": "2002"
  }
}
```

Agora, voltando a nossa rota. Em nossa rota POST eu vou querer fazer a requisição de “nome” e “idade”:

```
app.post('/enviar', function(req, res) {
  const { nome, idade } = req.body;

  res.send({ "nome_da_requisição": nome, "idade_da_requisição": idade });
})
```

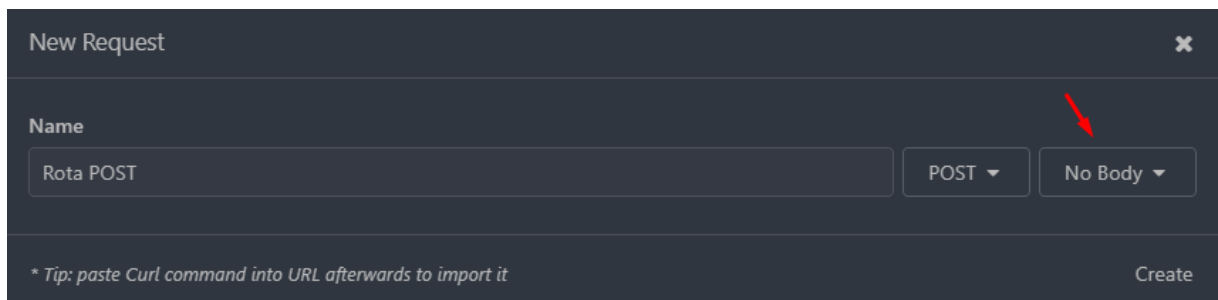
Criei uma constante e passei nome e idade para dentro dela, só que, passei dentro de “{}”, isso se chama desestruturação, fiz isso pois quero pegar apenas os atributos nome e idade de um arquivo JSON. O *req.body* apenas diz que estou fazendo a requisição no corpo da aplicação. Também coloquei para retornar um JSON mostrando o nome e a idade. Antes de testarmos precisamos fazer nossa API ler dados JSON, para isso basta adicionar *app.use(express.json());* logo após a criação da constante APP:

```

1  const express = require('express');
2  const app = express();
3
4  app.use(express.json());
5
6  app.get('/', function(req, res) {
7    res.send('Olá mundo');
8  });
9
10 app.post('/enviar', function(req, res) {
11   const { nome, idade } = req.body;
12   res.send({ "nome": nome, "idade": idade });
13 })
14
15 app.listen(3000);
16

```

Agora crie uma nova request no Insomnia, dessa vez do tipo POST e você verá uma opção chamada BODY:



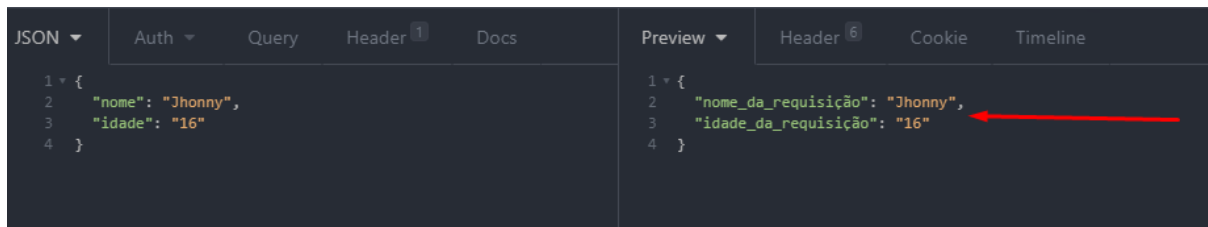
Clique e selecione o formato JSON. Agora clique em sua request e dentro do campo de texto digite:

```

JSON ▾  Auth ▾  Query  Header 1  Docs
1 {
2   "nome": "Jhonny",
3   "idade": "16"
4 }

```


Pode colocar qualquer nome e qualquer idade, mas, tem que ser esses dois atributos pois foram os que nós pedimos em nossa API e aperte em Send. Você verá que ele retornou o que pedimos:



Dentro de nossas rotas também podemos adicionar códigos lógicos, por exemplo, quero fazer o seguinte: se o nome for Jhonny, quero que retorne Brabo, para isso seria simplesmente fazer:

```
app.post('/enviar', function(req, res) {
  const { nome, idade } = req.body;

  if(nome == 'Jhonny') {
    res.send('Brabo');
  } else {
    res.send({ "nome_da_requisição": nome, "idade_da_requisição": idade });
  }
})
```

Ou poderíamos fazer também:

```
app.post('/enviar', function(req, res) {
  const { nome, idade } = req.body;

  if(nome == 'Jhonny') {
    res.send({ "nome_da_requisição": nome + ', O mais Brabo', "idade_da_requisição": idade });
  } else {
    res.send({ "nome_da_requisição": nome, "idade_da_requisição": idade });
  }
})
```

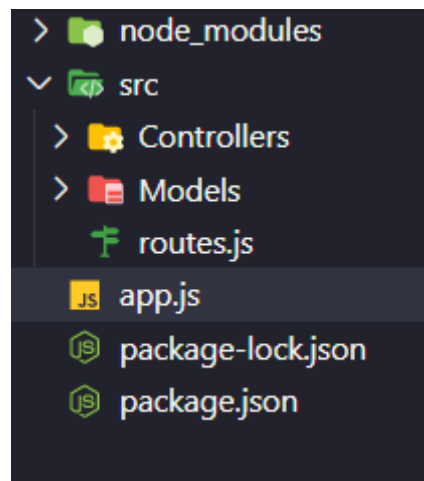
Não pense que as rotas do tipo GET são inúteis, irei dar uma explicação melhor. Vamos supor que você está criando uma API para um site e o mesmo tem um formulário de cadastro e login. Para a pessoa logar ela precisa informar o nome e a senha. A API teria que verificar se existe esse nome no banco e se o nome bate com a senha do banco de dados. Isso no caso seria uma rota do tipo GET, e, no cadastro, seria do tipo POST.

Exercícios:

- Crie uma nova API do zero, evite ao máximo olhar o código anterior
- Nessa API crie duas rotas, uma do tipo GET e outra do tipo POST. A GET deverá retornar um JSON com a propriedade “status” e o valor “working” a rota POST deverá receber uma idade, se a idade for menor do que 18, a API deverá retornar { “error”: “você não tem idade” } se não, { “status”: ok }

Parte 3: Estruturando o projeto

Bem, se deixássemos tudo dentro do **app.js** o código ficaria um nojo, horrível! Imagine que você tem banco de dados, várias rotas e etc... Ficaria gigantesco o código e mal otimizado. Para isso podemos separar nosso projeto em varias pastas. A nossa estrutura de pastas e arquivos ficará assim:



A primeira coisa a se fazer, é passar nossos códigos lógicos para a pasta **Controllers**, irei criar um arquivo lá dentro chamado **nameController.js** e ele ficará assim:

```

1  module.exports = {
2      async home(req, res) {
3          res.send('Olá mundo');
4      },
5
6      async nameBrabo(req, res) {
7          const { nome, idade } = req.body;
8
9          if(nome == 'Jhonny') {
10             res.send({ "nome_da_requisição": nome + ', O mais Brabo', "idade_da_requisição": idade });
11          } else {
12             res.send({ "nome_da_requisição": nome, "idade_da_requisição": idade });
13          }
14      }
15  }
16
17 }

```

A princípio parece assustador, mas, confie em mim, não é. Na primeira linha estou exportando nosso código, possibilitando que a gente o importe em outros arquivos. Logo depois criei duas funções assíncronas e copieie o coleie código lógico que estava no app (se não souber o que são funções assíncronas, acesse esse link: <https://imasters.com.br/desenvolvimento/funcoes-assincronas-e-retornos-como-o-async-await-tornaram-o-codigo-mais-legivel>). Agora precisamos configurar nossas rotas. Edite o arquivo **routes.js** e deixe igual ao meu:

```

1  const express = require('express');
2  const routes = express.Router();
3
4  const nameController = require('./Controllers/nameController');
5
6  routes.get('/', nameController.home);
7  routes.post('/enviar', nameController.nameBrabo);
8
9  module.exports = routes;

```

É bem simples de entender o que acontece aqui. Na 2 linha eu estou invocando a função Router do Express dentro de uma constante chamada routes. Na linha 4 eu importo nosso arquivo da pasta **Controllers** (se não tivéssemos colocado o *module.exports* lá atrás não seria possível importa-lo). E por fim nas linhas 6 e 7 eu recrio aquele esquema das rotas que estava em nosso **app.js**, passo a rota e como segundo parâmetro passo o que executar nessa rota.

Agora tudo o que precisamos fazer é importar nossas rotas dentro do **app.js**:

```

1  const express = require('express');
2  const app = express();
3
4  app.use(express.json());
5
6  app.use(require('./src/routes'));
7
8  app.listen(3000);

```

Se você testar as rotas no Insomnia novamente, verá que está funcionando normalmente.

Parte 4: O banco de dados

Nesse tutorial, iremos usar o tipo de banco de dados não relacional (noSQL) e para utilizarmos, iremos usar o MongoDB (se não souber o que é de uma lida aqui: <https://blog.totalcross.com/pt/banco-de-dados-relacional-nao-relacional/>). Iremos armazenar músicas em nosso banco. Iremos ter os campos: nome, gênero e ano. Para começar, você vai precisar instalar o pacote mongoose, que nos permite utilizar o Mongo em nossa API (você já deveria saber baixar os pacotes, mas se não souber: *npm i mongoose*). Agora crie um arquivo chamado **songModel.js** dentro da pasta **Models** e deixe igual ao meu:

```

1  const mongoose = require('mongoose');
2
3  const songSchema = mongoose.Schema({
4    nome: String,
5    genero: String,
6    ano: Number
7  });
8
9  module.exports = mongoose.model('Song', songSchema);

```

Na 3 linha eu criei uma constante chamada songSchema e dentro dela o nosso Schema (os Schemas são o equivalente as tabelas do SQL normal). Na nossa Schema coloquei os nossos atributos e defini o tipo deles e na linha 9, exportei nossa tabela.

Em nossa pasta **Controller** agora, vamos criar um arquivo chamado **songController.js** e vamos criar uma função para adicionar músicas ao banco:

```
1  const songSchema = require('../Models/songModel');
2
3  module.exports = {
4    async createSong(req, res) {
5      const { nome, genero, ano } = req.body;
6      const song = await songSchema.create({ nome, genero, ano });
7      return res.json(song);
8    }
9  }
```

A única coisa nova nesse trecho de código é a linha 6. Nela coloquei o *await* pois sempre que estivermos mexendo com o banco temos que usar o *await*. Logo depois chamei a função *create* e passei os dados que quero que sejam enviados ao banco.

Agora precisamos criar uma nova rota em nosso **Routes.js**:

```
1  const express = require('express');
2  const routes = express.Router();
3
4  const songController = require('./Controllers/songController');
5  const nameController = require('./Controllers/nameController');
6
7  routes.get('/', nameController.home);
8  routes.post('/enviar', nameController.nameBrabo);
9
10 routes.post('/createsong', songController.createSong);
11
12 module.exports = routes;
```

E é de extrema importância conectarmos ao MongoDB, se não, nada disso irá funcionar. Para isso vá em **app.js** e adicione a seguinte linha de código:

```
mongoose.connect('mongodb://localhost:27017/nodeapi', {
  useNewUrlParser: true, useUnifiedTopology: true
});
```

Os “use” são apenas para não mostrar uns avisos no terminal, coloque eles por padrão. Não se esqueça de importar o mongoose dentro do **app.js** o código por inteiro do **app.js** fica assim:

```
1  const express = require('express');
2  const app = express();
3  const mongoose = require('mongoose');
4  const cors = require('cors');
5
6  app.use(express.json());
7  app.use(cors());
8
9  mongoose.connect('mongodb://localhost:27017/nodeapi', {
10    useNewUrlParser: true, useUnifiedTopology: true
11  });
12
13  app.use(require('./src/routes'));
14
15  app.listen(3000);
```

Note na linha 7 que adicionei isso recentemente. Isso nos permite enviar os dados somente a nossa aplicação frontend, isso impede que pessoas tentem acessar nossas requests.

Se você testar a rota agora e enviar os atributos verá o seguinte resultado no Insomnia:

Method	URL	Status	Time	Size
POST	http://localhost:3000/createsong	200 OK	453 ms	103 B

JSON	Auth	Query	Header	Docs
<pre>1 { 2 "nome": "The Odyssey", 3 "genero": "Progressive Metal", 4 "ano": 2002 5 }</pre>				

Preview	Header	Cookie
<pre>1 { 2 "_id": "5eaab6ff1179002d94e8d1e6", 3 "nome": "The Odyssey", 4 "genero": "Progressive Metal", 5 "ano": 2002, 6 "__v": 0 7 }</pre>		

Deu certo! Nossa música está agora cadastrada em nosso banco de dados. Podemos fazer uma verificação para saber se já está cadastrada, e se não estiver, cadastrar a música, basta adicionar o seguinte em nossa função createSong do nosso controller:

```

4   async createSong(req, res) {
5       const { nome, genero, ano } = req.body;
6
7       let song = await songSchema.findOne({ nome });
8
9       if(!song) {
10          const song = await songSchema.create({ nome, genero, ano });
11          return res.json(song);
12      } else {
13          return res.json({ "error": "Song already exists" });
14      }
15  }
16  }

```

Iremos criar agora uma rota para mostrar uma musica passando o ID dela na rota:

```

18  async getSongs(req, res) {
19      const song = await songSchema.findById(req.params.id);
20      res.json({ "song": song });
21  }

```

E agora criar a rota dela no **Routes.js**:

```

11  routes.get('/getsongs/:id', songController.getSongs);

```

Na linha 19 da imagem anterior, no lugar de ID podia ser qualquer nome que você quisesse. O *req.params* apenas faz uma requisição a partir da rota que no caso colocamos */getsongs/:id* ou seja, no lugar de *“:id”* colocaremos o ID de uma musica criada:

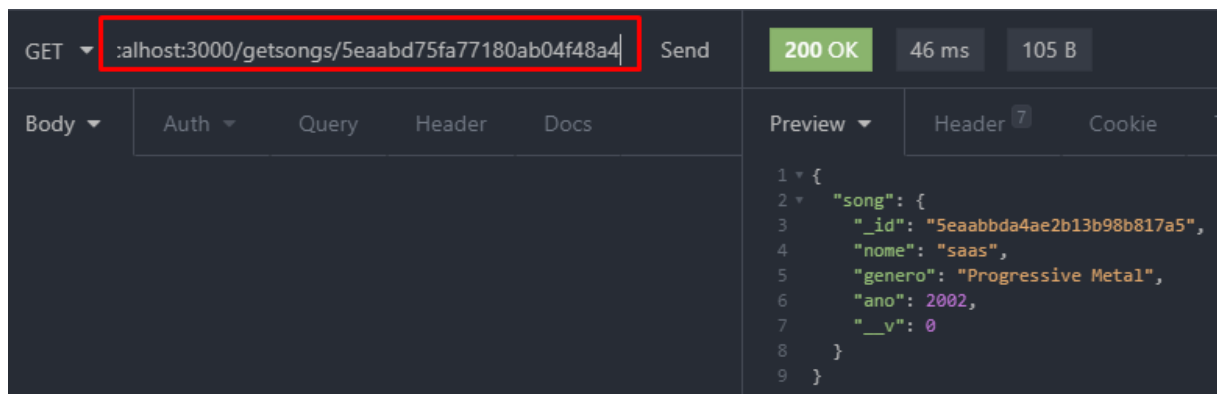
The screenshot shows a REST client interface. On the left, the 'JSON' tab displays the request body:

```
{ "nome": "The Trooper", "genero": "Heavy Metal", "ano": 1983 }
```

. On the right, the 'Preview' tab shows the response body:

```
{ "_id": "5eaabd75fa77180ab04f48a4", "nome": "The Trooper", "genero": "Heavy Metal", "ano": 1983, "__v": 0 }
```

. The `"_id"` field in the response is highlighted with a red rectangle.



Simples não é mesmo? É basicamente isso. Agora como exercício, estude os outros métodos que tem do mongoose, como o `findByIdAndDelete`, entre outros. Também como exercício, crie uma API do zero, evitando ao máximo olhar nos códigos anteriores, porém, nessa sua nova API, quero que faça com banco de dados e um CRUD completo no mesmo. (CRUD = Create, Update e Delete).

Bibliografia:

- <https://expressjs.com/pt-br/guide/routing.html>
- <https://docs.mongodb.com/manual/introduction/>
- <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>