

Conteúdo

Módulo 1: Introdução ao Node.JS

- APIs e coleção de dados
- Retornando a coleção
- Retornando um elemento único
- Tratamento de erros na API
- Aplicando buscas na API
- Conclusão
- Atividades Extras

Continuando agora iremos montar nossa requisição para retornar de forma simples nossa coleção completa através do endpoint **/ufs** sendo consumida por nosso Thunder Client.

vamos criar o arquivo **index.js**:

```
JS index.js > ...
1  import express from 'express';
2  import colecaoUf from './dados/dados.js';
3
4  const app = express();
5
6  app.get('/ufs', (req, res) => {
7    |   res.json(colecaoUf)
8  });
9
10 app.listen(8080, () => {
11   |   console.log('Servidor iniciado na porta 8080');
12 });
```

Antes de rodarmos **precisamos express**, notem que **importamos a nossa coleção de dados da pasta dados e dados.js e armazenamos ela na variavel colecaoUf** e depois criamos a rota **/ufs** e listamos o servidor na porta 8080.

```
JS index.js > ...
1  import express from 'express';
2  import colecaoUf from './dados/dados.js';
3
4  const app = express();
5
6  app.get('/ufs', (req, res) => {
7    res.json(colecaoUf)
8  });
9
10 app.listen(8080, () => {
11   console.log('Servidor iniciado na porta 8080');
12 });
```

```
JS index.js > ...
1  import express from 'express';
2  import colecaoUf from './dados/dados.js';
3
4  const app = express();
5
6  app.get('/ufs', (req, res) => {
7    res.json(colecaoUf)
8  });
9
10 app.listen(8080, () => {
11   console.log('Servidor iniciado na porta 8080');
12 });
```

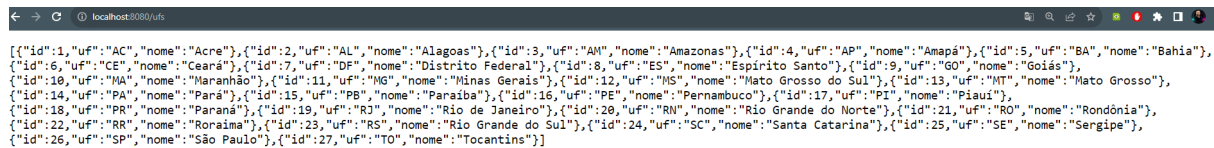
Para iniciarmos precisamos criar o package.json e depois instalarmos o express para conseguirmos executar o comando para rodar o servidor embutido, façam isso qualquer dúvida tem nas OTs passadas ok ?

Unisenai

Tudo ok, basta rodarmos o comando 👍

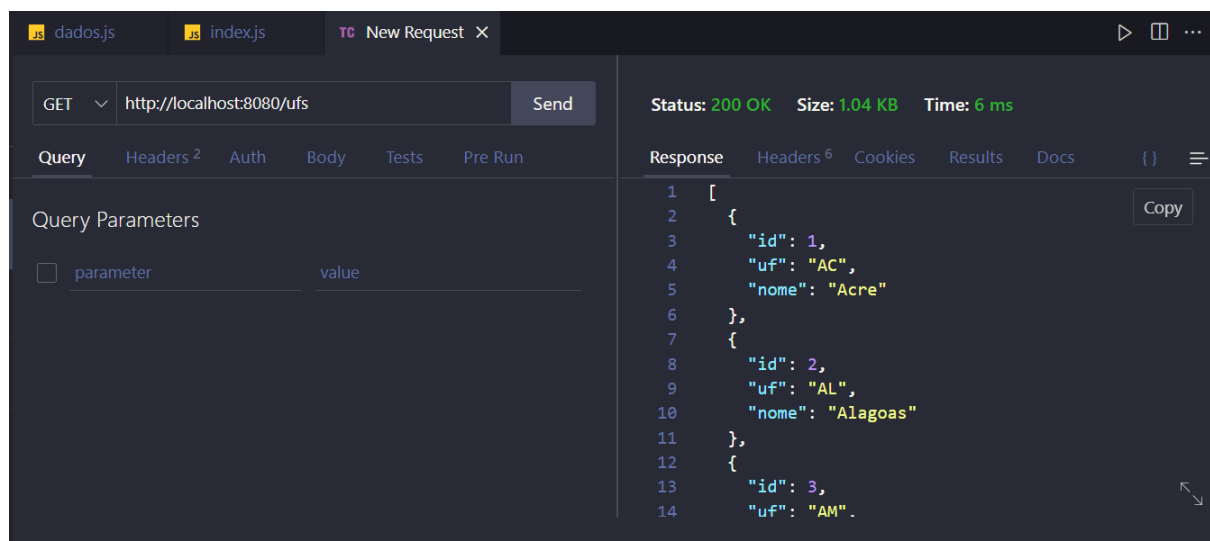
```
carlo@PC-UCHOA MINGW64 ~/OneDrive/Área de Trabalho/apiRest
$ node index.js
Servidor iniciado na porta 8080
```

E ao acessarmos a rota <http://localhost:8080/ufs>



```
[{"id":1,"uf":"AC","nome":"Acre"}, {"id":2,"uf":"AL","nome":"Alagoas"}, {"id":3,"uf":"AM","nome":"Amazonas"}, {"id":4,"uf":"AP","nome":"Amapá"}, {"id":5,"uf":"BA","nome":"Bahia"}, {"id":6,"uf":"CE","nome":"Ceará"}, {"id":7,"uf":"DF","nome":"Distrito Federal"}, {"id":8,"uf":"ES","nome":"Espírito Santo"}, {"id":9,"uf":"GO","nome":"Goiás"}, {"id":10,"uf":"MA","nome":"Maranhão"}, {"id":11,"uf":"MG","nome":"Minas Gerais"}, {"id":12,"uf":"MS","nome":"Mato Grosso do Sul"}, {"id":13,"uf":"MT","nome":"Mato Grosso"}, {"id":14,"uf":"PA","nome":"Paraná"}, {"id":15,"uf":"PB","nome":"Paraíba"}, {"id":16,"uf":"PE","nome":"Pernambuco"}, {"id":17,"uf":"PI","nome":"Piauí"}, {"id":18,"uf":"PR","nome":"Paraná"}, {"id":19,"uf":"RJ","nome":"Rio de Janeiro"}, {"id":20,"uf":"RN","nome":"Rio Grande do Norte"}, {"id":21,"uf":"RO","nome":"Rondônia"}, {"id":22,"uf":"RR","nome":"Roraima"}, {"id":23,"uf":"RS","nome":"Rio Grande do Sul"}, {"id":24,"uf":"SC","nome":"Santa Catarina"}, {"id":25,"uf":"SE","nome":"Sergipe"}, {"id":26,"uf":"SP","nome":"São Paulo"}, {"id":27,"uf":"TO","nome":"Tocantins"}]
```

Acessando pelo Thunder Client:



GET http://localhost:8080/ufs Status: 200 OK Size: 1.04 KB Time: 6 ms

Query Parameters

parameter	value
parameter	value

Response

```
[
  {
    "id": 1,
    "uf": "AC",
    "nome": "Acre"
  },
  {
    "id": 2,
    "uf": "AL",
    "nome": "Alagoas"
  },
  {
    "id": 3,
    "uf": "AM"
  }
]
```

Assim fazemos o nosso teste de consumo de nossa API.

Neste momento, nossa API retorna toda a coleção de dados através do endpoint **/ufs**.

Vamos evoluir nossa API, para que ela possa retornar também uma **UF específica** (ao invés de toda a coleção). Isso ja vimos e iremos criar um novo endpoint:

localhost:8080/ufs/[ID da UF]

Veja um exemplo de consumo dessa nossa endpoint:

localhost:8080/ufs/19

localhost:8080/ufs/19

The screenshot shows a REST client interface. At the top, the method is 'GET' and the URL is 'localhost:8080/ufs/19'. Below this, there are tabs for 'Params', 'Authorization', 'Headers (6)', 'Body', 'Pre-request Script', and 'Test Results'. The 'Params' tab is selected, showing a table with 'Key' and 'Value' columns. Below the table, there are tabs for 'Body', 'Cookies', 'Headers (7)', and 'Test Results'. The 'Body' tab is selected, showing a JSON response in 'Pretty' format. The response is: { "id": 19, "uf": "RJ", "nome": "Rio de Janeiro" }. The 'JSON' tab is also visible, and there is a refresh button.

Key	Value
-----	-------

Body Cookies Headers (7) Test Results

Pretty Raw Preview Visualize JSON

```
1 {  
2   "id": 19,  
3   "uf": "RJ",  
4   "nome": "Rio de Janeiro"  
5 }
```

Ou seja , nossa **API** vai passar a ter duas rotas de consumo:

Rota completa: localhost:8080/ufs

Rota com elemento único: localhost:8080/ufs/19

Faremos isso por dois motivos.

Motivo 1:

No **Padrão REST**, sempre que a **API** manipula uma coleção, devemos oferecer duas rotas:

Uma para retornar toda coleção: localhost:8080/ufs

E outra para retornar um item único da coleção: localhost:8080/ufs/19

Motivo 2:

O contexto do sistema. Normalmente, um sistema front-end vai ter uma página de **lista** e outra de **detalhes**.

Então vamos evoluir nossa **API** para que ela retorne apenas uma **UF**, através de uma nova rota.

Sintaxe do padrão REST

Antes de iniciar a implementação, vamos ver um pouco sobre a sintaxe de criação desta nova rota de acordo com o **padrão REST**.

No **REST**, a rota para consumo de **apenas um item** da coleção segue o padrão de endpoint abaixo:

`localhost:8080/ufs/19`

url

domínio
da coleção

ID do
elemento

O **ID** utilizado no endpoint será o **mesmo** utilizado na coleção de dados. Dessa forma a API vai saber qual item da coleção retornar.

`localhost:8080/ufs/19`

```
{
  id: 18,
  uf: "PR",
  nome: "Paraná"
},
{
  id: 19,
  uf: "RJ",
  nome: "Rio de Janeiro"
},
{
  id: 20,
  uf: "RN",
  nome: "Rio Grande do Norte"
},
}
```

Outro ponto importante é que a **API** deve retornar **todos** os campos do elemento.

```
{  
  id: 19,  
  uf: "RJ",  
  nome: "Rio de Janeiro"  
},
```

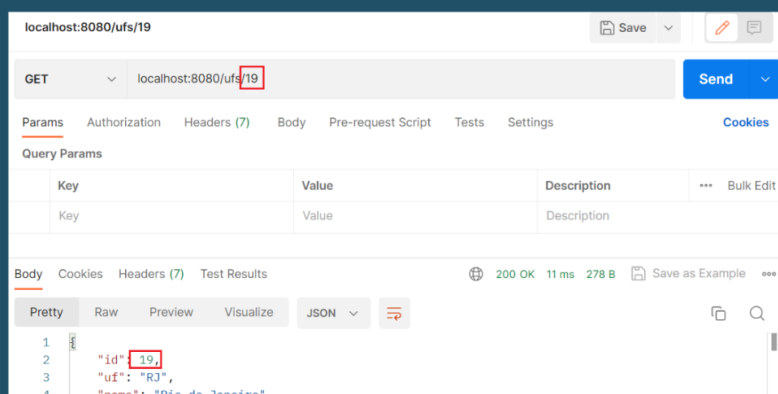
Coleção de dados



```
{  
  "id": 19,  
  "uf": "RJ",  
  "nome": "Rio de Janeiro"  
}
```

Postman

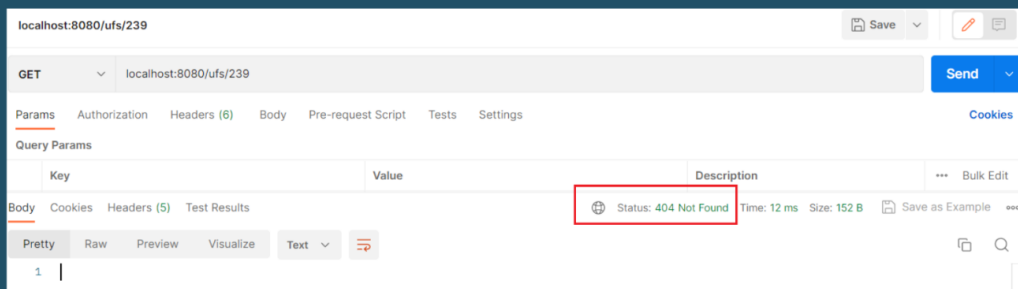
Por exemplo, observe que o ID se **'repete'**. Apesar de ter sido informado no endpoint, a API também o **retorna como corpo** da mensagem:



E por fim:

Unisenai

Por último, se o cliente consumir a **API** passando um **ID que não existe** na coleção, a API deve retornar um código de **erro 400** (objeto não encontrado). Exemplo: `localhost:8080/ufs/293`



O objetivo é que a **API** passe a oferecer **duas rotas** de consumo.



Codando uma nova rota

Vamos atualizar o arquivo `index.js`, para que a API tenha uma segunda rota.

Na primeira versão da alteração, vamos implementar uma segunda rota de teste, que vai usar o **endpoint**: <http://localhost:8080/ufs/teste>.

A API vai retornar para o cliente um json contendo o valor estático
“teste”:"teste”.

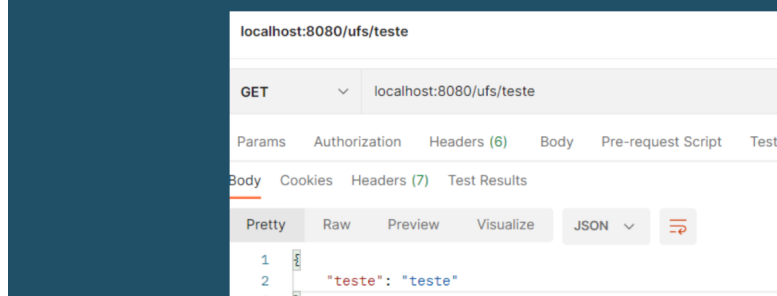
Código para inserir no arquivo index.js:

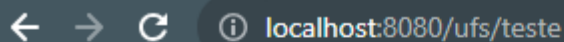
```
JS index.js > ...
1  import express from 'express';
2  import colecaoUf from './dados/dados.js';
3
4  const app = express();
5
6  app.get('/ufs', (req, res) => {
7    res.send(colecaoUf)
8  });
9
10
11  app.get('/ufs/teste', (req, res) => {
12    res.send({ "teste": "teste" })
13  }
14  );
15
16  app.listen(8080, () => {
17    console.log('Servidor iniciado na porta 8080');
18  });
```

Nova rota

Agora ao testarmos:

Vamos começar com uma **versão simples** deste novo consumo. A **nova rota** será /ufs/teste e o retorno da API será um **json** com o valor “teste”:"teste”.





← → ↻ ⓘ localhost:8080/ufs/teste

```
{"teste": "teste"}
```

Neste momento nossa API possui duas rotas de consumo:

http://localhost:8080/ufs - Retorna todas as UFs da coleção;

http://localhost:8080/ufs/teste - Retorna uma mensagem de teste para o cliente.

Nosso objetivo agora é evoluir a segunda rota, para que ela retorne uma UF específica da coleção. Dessa forma, a rota vai passar a ser /domínio/ID do elemento:

https://localhost:8080/ufs/ID do elemento

Para essa alteração, vamos precisar utilizar o método find do JavaScript - ele permite buscar um elemento específico dentro de uma coleção de dados.

Evoluindo nosso código da rota nova

Agora vamos fazer a nova rota retornar um elemento específico da coleção de dados, aplicando o método find.

Novo código para nosso index.js.

```
import express from 'express';
import colecaoUf from './dados/dados.js';

const app = express();

app.get('/ufs', (req, res) => {
  res.json(colecaoUf)
});

app.get('/ufs/:iduf', (req, res) => {
  const idUF = parseInt(req.params.iduf);
  const uf = colecaoUf.find(u => u.id === idUF);

  res.json(uf);
});

app.listen(8080, () => {
  console.log('Servidor iniciado na porta 8080');
});
```

A ideia agora é **evoluir** o código da rota. Vamos trocar a **rota estática** /ufs/teste , por uma **rota dinâmica**, que tenha a **sintaxe** /ufs/id do elemento.

Por exemplo:

https://localhost:8080/ufs/14

https://localhost:8080/ufs/25

Veja o método `app.get` atualizado:

```
JS index.js > ...
1  import express from 'express';
2  import colecaoUf from './dados/dados.js';
3
4  const app = express();
5
6  app.get('/ufs', (req, res) => {
7    res.json(colecaoUf)
8  });
9
10
11  app.get('/ufs/:iduf', (req, res) => {
12    const idUF = parseInt(req.params.iduf);
13    const uf = colecaoUf.find(u => u.id === idUF);
14
15    res.json(uf);
16  }
17  );
18
19  app.listen(8080, () => {
20    console.log('Servidor iniciado na porta 8080');
21  });
```

Perceba a nova sintaxe no endpoint:

`app.get('/ufs/:iduf', (req,res)`

“:” + nome do parâmetro(iduf), indica um valor dinâmico recebido em `app.get`

```
app.get('/ufs/:iduf', (req, res) => {  
  const idUF = req.params.iduf;  
  
  res.send({ "teste": `${idUF}` });  
});
```

O nome do parâmetro pode ser qualquer valor, Neste momento e exemplo chamamos de iduf.

Dessa forma, quando o cliente **consumir**:

`https://localhost:8080/14`

O parâmetro "iduf" **terá o valor 14**.

GET localhost:8080/ufs/14



```
app.get('/ufs/:iduf', (req, res) => {  
  const idUF = req.params.iduf;  
  
  res.send({ "teste": `${idUF}` });  
});
```

O valor do parâmetro é lido através de **req.params**

```
1 app.get('/ufs/:iduf', (req, res) => {  
  const idUF = req.params.iduf; 2  
3 res.send({ "teste" : `${idUF}` });  
}  
);
```

- 1 A rota **recebe** um valor dinâmico como parâmetro.
- 2 idUF é **definido** com o valor do parâmetro.
- 3 A API **retorna** um json, contendo o valor de idUF.

Por exemplo, se o cliente **consumir**:

`https://localhost:8080/ufs/19`

Código	Valor
<pre>app.get('/ufs/:iduf', (req, res) => { const idUF = req.params.iduf;</pre>	A constante idUF terá o valor 19;
<pre> res.send({"teste": `\${idUF}`});</pre>	vai gerar o retorno em json: <pre>{ "teste": "19" }</pre>

Com essa rota já conseguimos testar **diferentes endpoints** no **Postman**. Veja os exemplos:



Usamos o método **find** do JavaScript para buscar a UF na coleção de dados, através do ID. O retorno é salvo na **variável uf**.

```
app.get('/ufs/:iduf', (req, res) => {  
  const idUF = parseInt(req.params.iduf);  
  const uf = colecaoUf.find(u => u.id === idUF);  
  
  res.json(uf);  
});
```

```
app.get('/ufs/:iduf', (req, res) => {  
  const idUF = parseInt(req.params.iduf);  
  const uf = colecaoUf.find(u => u.id === idUF);  
  
  res.json(uf);  
});
```

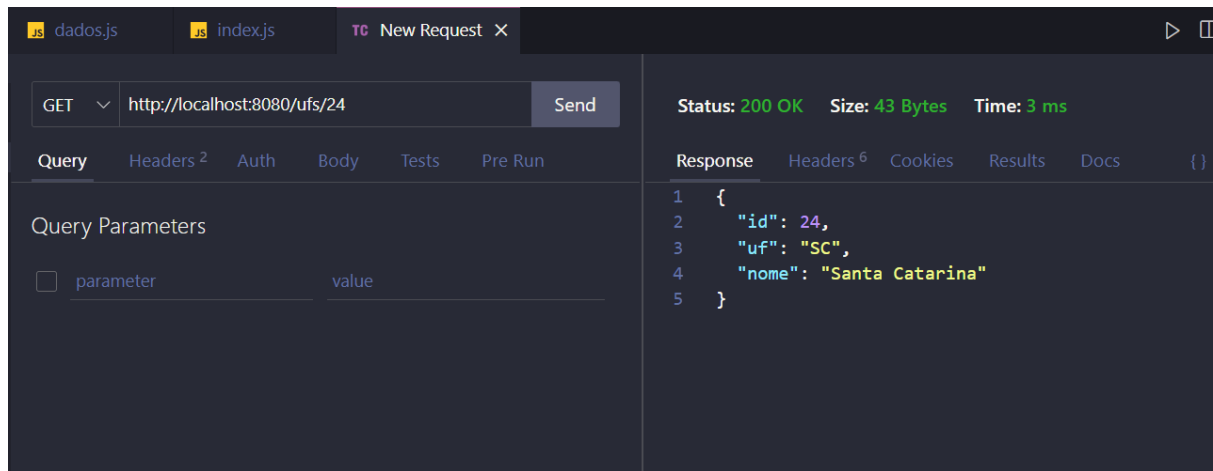
Por fim, passamos a variável **uf** para o método **json**, uma vez que a coleção de dados **não está nesse formato**.

Testando:

← → ↻ ⓘ localhost:8080/ufs/24

```
{"id":24,"uf":"SC","nome":"Santa Catarina"}
```


Unisenai



Continuamos na próxima OT, por aqui finalizamos mais esta etapa.