

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

Nesta OT pretendo guiar vocês na criação de um CRUD com Node JS! Essa é a base para muitos, senão todos, os programadores back-ends das empresas.

O conhecimento que você terá ao concluir o conteúdo deste artigo permitirá que você crie suas próprias API REST! Seja para criar um CRUD de casas para alugar, ou até mesmo de quartos disponíveis em um hostel.

Você pode adaptar esse conhecimento e, com os ajustes certos, aplicá-lo a qualquer situação.

API REST

Os benefícios deste modelo de API REST é que podemos servir múltiplos clientes com o mesmo back-end, ou seja, um único código fornecido para Web Mobile ou até mesmo uma API pública.

É importante entender o fluxo de requisição e resposta, não vou entrar em detalhes, mas basicamente acontece nesta sequência:

- Requisição é feita por um cliente;
- Resposta retornada através de uma estrutura de dados (ex: JSON);
- O cliente recebe a resposta e processa o resultado.

Estas respostas utilizam métodos HTTP, que são:

- **GET** `http://minhaapi.com/users` → Buscar alguma informação no back-end
- **POST** `http://minhaapi.com/users` → Criar alguma informação no back-end
- **PUT** `http://minhaapi.com/users/1` → Editar alguma informação no back-end
- **DELETE** `http://minhaapi.com/users/1` → Deletar alguma informação no back-end

É importante também que você entenda sobre HTTP codes, que são os códigos HTTPs retornados de uma requisição, vejamos alguns exemplos mais comuns:

- **1xx**: HTTP codes iniciados em 1 são informativos:
- **102**: PROCESSING.
- **2xx**: HTTP codes iniciados com 2 são de sucesso:
- **200**: SUCCESS;
- **201**: CREATED.
- **3xx**: HTTP codes iniciados em 3 são de redirecionamento:
- **301**: MOVED PERMANENTLY;
- **302**: MOVED.
- **4xx**: HTTP codes iniciados em 4 são de erros do cliente:
- **400**: BAD REQUEST;
- **401**: UNAUTHORIZED;

- **404**: NOT FOUND.
- **5xx**: HTTP codes iniciados em 5 são erros do servidor:
- **500**: INTERNAL SERVER ERROR.

Vamos entender melhor sobre estes métodos, recursos/rotas e parâmetros na prática, aguenta só um pouquinho.

Benefícios da API REST

Os benefícios deste modelo de API REST é que podemos servir múltiplos clientes com o mesmo back-end, ou seja, um único código fornecido para Web Mobile ou até mesmo uma API pública.

Vamos iniciar a parte prática, criando a primeira versão de uma API que manipula coleções de dados.

Nesta versão inicial teremos uma API que retorna uma coleção de UFs.

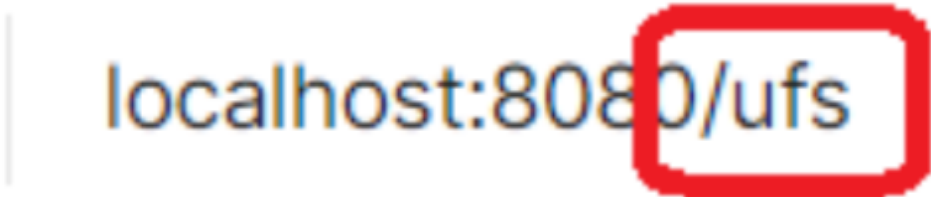
Essa API pode ser usada, por exemplo, por um front-end que solicite ao usuário o preenchimento de seu endereço. A API pode ser consumida para preencher um listbox com todas as UFs existentes.

Nesta primeira versão a API vai retornar a coleção completa de UFs para o cliente, através do endpoint **/dominio/ufs**:

Padrão de endpoints

Vamos ver agora um ponto importante na construção de APIs que lidam com coleções de dados: o padrão na construção dos endpoints. Veja o flow abaixo:

Observe o endpoint abaixo. Neste exemplo, temos uma API que retorna uma lista de UFs:



localhost:8080/ufs

Note que o nome do **endpoint (/ufs)** é o mesmo nome do contexto da coleção de dados (**UFs**).

Outro exemplo: digamos que você vai construir uma **API que retorna uma lista de times de futebol**.

Seguindo a mesma lógica, o endpoint usaria o nome do contexto da coleção:

```
localhost:8080/times
```

Essa lógica faz parte do **padrão REST**. Ou seja, uma **API** que retorna dados de uma **coleção**, deve usar um **endpoint** que represente o **contexto de dados**.

Coleção de produtos:

```
localhost:8080/produtos/
```

Coleção de jogos:

```
localhost:8080/jogos/
```

Alguns sistemas possuem **mais de um contexto de dados**. Por exemplo, em uma mesma **API**, podemos ter os contextos **cursos**, **usuários** e **vendas**:

- localhost:8080/cursos

- **localhost:8080/usuarios**
- **localhost:8080/vendas**

Os contextos também podem ter uma relação de hierarquia. Por exemplo, cursos e aulas.

Nesse caso, os endpoints seguem a hierarquia:

localhost:8080/cursos/80/aulas

Não se preocupe com isso agora, pois vamos trabalhar somente com um contexto de dados.

Domínios

Vamos adotar uma palavra muito usada na programação back-end: domínios. Veja mais a seguir.

Nossa API de exemplo possui apenas um domínio: UFs

Dessa forma , o mesmo domínio será utilizado em todos os endpoints de consumo:

localhost:8080/ufs

localhost:8080/ufs/1

localhost:8080/ufs/busca?=rio

Iniciando a construção da API

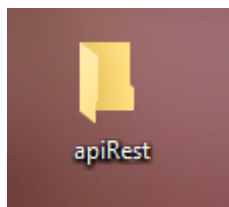
Já sabemos que nossa **API** de exemplo possui apenas um domínio: **UFs**, também vimos que o endpoint deve ser o nome do domínio, dessa forma teremos:

<http://localhost:8080/ufs>

Vamos iniciar a construção da API, começando pela coleção de dados de UFs, segue abaixo:

Vamos começar montando a coleção de dados de UFs:

Crie uma pasta nova na área de trabalho chamada **apiRest**:



Após isso, abra com VScode e crie um arquivo chamado **dados.js**, dentro da pasta dados, que contenha **toda a coleção. Exemplo:**


```
const colecaoUf = [  
  {  
    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"  
  },  
]
```

Note que temos um campo **ID**.

O **ID** é um número sequencial - por exemplo, **"AC"** terá **ID=1**, **"AL"** terá **ID=2**, assim por diante.

```
const colecaoUf = [  
  {  
    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"  
  },  
]
```

O uso do **ID** nas coleções de dados é um **padrão REST**.

Ou seja, sempre que sua API **manipular dados**, a coleção deve possuir um **campo ID** para cada registro.

Vejamos outro exemplo para compreendermos:

```
const colecaoTimes = [
  {
    id: 1,
    nome: "Flamengo"
  },
  {
    id: 2,
    nome: "Atlético Mineiro"
  },
  {
    id: 3,
    nome: "Palmeiras"
  },
  {
    id: 4,
    nome: "São Paulo"
  }
]
```

Por fim , **exportaremos** a coleção para que possamos chamá-la no **arquivo index.js** que iremos criar, **essa parte de exportar já aprendermos nas OTs iniciais**.

```
{
  id: 26,
  uf: "SP",
  nome: "São Paulo"
},
{
  id: 27,
  uf: "TO",
  nome: "Tocantins"
}]

export default colecaoUf
```

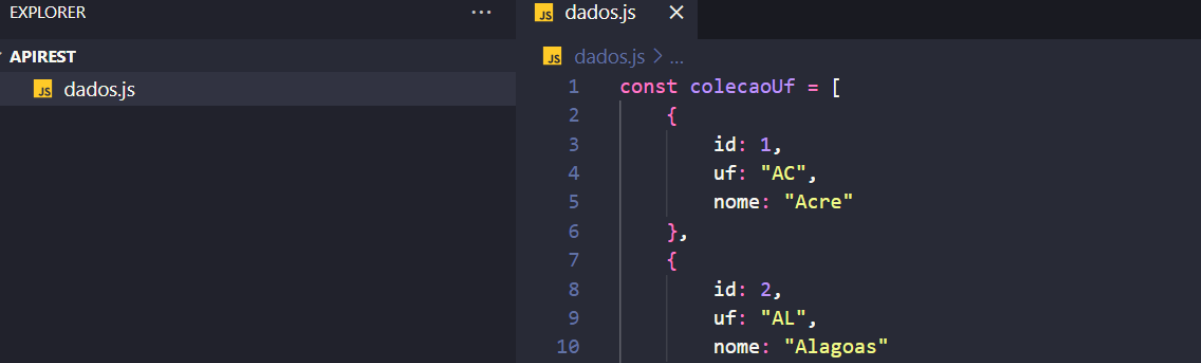
Unisenai

Segue código da coleção de dados completa abaixo:

Link no gist do github para facilitar a vida de vocês:

<https://gist.github.com/uchoamaster/bb3bfdf04382bb8800c761be2ab3d745>

Por fim ficará assim :



```
1  const colecaoUf = [
2    {
3      id: 1,
4      uf: "AC",
5      nome: "Acre"
6    },
7    {
8      id: 2,
9      uf: "AL",
10     nome: "Alagoas"
```

Continuamos na próxima OT...