

Projeto:	Cadastro Simples – Docker compose		
Autor:	José Martinele Alves Silva		
Recursos:	Node	Mongo	Compose
Mês / Ano:	Set / 2021	Versão:	1.0

Sumário

1	Estrutura inicial – FASE 1	1
1.1	Composição do projeto	1
1.2	Estrutura das pastas e seus respectivos arquivos	1
1.2.1	Criando o arquivo descritor do node	1
1.2.2	Configurando o arquivo package.json	2
2	Configurando o ambiente com “compose – Docker” – FASE 2	3
2.1	Configurando o arquivo ‘app.js’	3
2.1.1	Importação das bibliotecas	3
2.1.2	Associando a API de Promise do Mongo a API de Promise do Node	3
2.1.3	Conexão com o banco de dados	4
2.2	Editando o arquivo ‘docker-compose.yml’	4
2.2.1	Definição da versão do Docker-compose	4
2.2.2	Definição dos serviços	4
2.2.3	Efetando testes.....	6
3	Finalizando o cadastro – FASE 3	8
3.1	Implementação final do Backend	8
3.1.1	Instalando mais duas bibliotecas	8
3.1.2	Excluindo linhas de testes.....	8
3.1.3	Definição de Middlewares	9
3.1.4	Mapeamento Objeto/Documento (ODM)	9
3.1.5	Criação da API REST (Rest API).....	10
3.1.6	Declaração das Rotas.....	10
3.2	Implementação final do Frontend	11
3.2.1	Explicação do código HTML	12
3.2.2	Codificando.....	13

1 Estrutura inicial – FASE 1

Projeto com “compose” do Docker

1.1 Composição do projeto

Neste projeto serão criados 3 containers, ou seja, neste caso serão utilizados 3 serviços, sendo que:

- a) Um serviço baseado em banco de dados, com o uso do Mongodb;
- b) Um serviço relativo ao Backend, com o uso do Node, feito em Javascript;
- c) Um serviço relativo ao Frontend, com o uso do Nginx, com uma página index.html, simples.

1.2 Estrutura das pastas e seus respectivos arquivos

Serão necessárias pastas para guardar os arquivos referentes ao Frontend, ao Backend, bem como o arquivo Docker-compose. Sendo assim, siga os passos abaixo:

1. Criar uma pasta de nome: node-mongo-compose;
2. Dentro da pasta node-mongo-compose, criar as pastas: frontend e backend;
3. Dentro da pasta backend, criar um arquivo de nome: **app.js**;
4. Dentro da pasta frontend, criar um arquivo de nome: **index.html**;
5. Estando na pasta: node-mongo-compose, criar um arquivo de nome: **docker-compose.yml**.

1.2.1 Criando o arquivo descritor do node

É necessário criar o arquivo descritor do node e para isso, pelo terminal, acesse a pasta: backend, localizada dentro da pasta node-mongo-compose e, execute o seguinte comando:

node-mongo-compose\backend > npm init -y <ENTER>

A saída deverá ser conforme abaixo, quando se tem o node instalado na máquina:

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
```

```
},  
"keywords": [],  
"author": "",  
"license": "ISC"  
}
```

Se o comando acima não funcionar, é porque não se tem o node instalado. Mas não se preocupe, pois o ambiente será em Docker. Porque todas as bibliotecas necessárias a este projeto serão atualizadas dentro da imagem criada para ambiente em Docker.

1.2.2 Configurando o arquivo package.json

Precisamos apenas do arquivo: package.json configurado, tanto que a pasta node_modules poderá ser excluída em seguida, pois os arquivos das dependências necessárias para o funcionamento deste projeto são baixados e atualizados no momento do uso da imagem.

Assim, execute no terminal, estando dentro da pasta “backend” a seguinte linha de comando:

```
> npm i --save express@4.15.3 mongoose@4.11.1 node-restful@0.2.6 body-parser@1.17.2 cors@2.8.3  
<ENTER>
```

NOTA: Observe que estão sendo instaladas as versões específicas.

Assim serão baixadas e instaladas todas as dependências necessárias.

Porém, VALE LEMBRAR: NÃO SÃO PRECISAS AS INSTALAÇÕES DESSAS DEPENDÊNCIAS E SIM DO ARQUIVO PACKAGE.JSON CONFIGURADO, conforme abaixo:

```
{  
  "name": "backend",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1"  
  },  
  "keywords": [],  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "body-parser": "^1.17.2",  
    "cors": "^2.8.3",  
    "express": "^4.15.3",  
    "mongoose": "^4.11.1",  
    "node-restful": "^0.2.6"  
  }  
}
```

Então, a pasta node_modules será excluída.

Em Linux, exclua a pasta node_modules com o comando: `rm -rf node_modules/` <ENTER>

Em Windows, faça a mesma exclusão com o comando: `del node_modules` <ENTER>

2 Configurando o ambiente com “compose – Docker” – FASE 2

A primeira coisa a ser feita é a implementação do backend do projeto. Então os comandos serão escritos no arquivo: `app.js`

2.1 Configurando o arquivo ‘app.js’

2.1.1 Importação das bibliotecas

A primeira coisa a ser feita, diz respeito a importação das bibliotecas:

```
const express = require('express')
const restful = require('node-restful')
const server = express()
const mongoose = restful.mongoose
```

Sendo:

Express → Servidor Web ‘express’ que será utilizado neste projeto;

Node-restful → Server para implementar os web-services de uma forma muito mais fácil;

server = express() → Para iniciar o “Server” a partir do método construtor do Express;

mongoose → Que está disponível a partir do node-restful

2.1.2 Associando a API de Promise do Mongo a API de Promise do Node

Será feita a conexão com o Mongodb (mongoose)

Porém antes é necessário fazer uma alteração para o mongoose use a API de promises do Node, ou seja, será associada a API de promises do Node para que o Mongoose use essa API, visto que a API do mongoose está depreciada.

```
//Database
mongoose.Promise = global.Promise
```

2.1.3 Conexão com o banco de dados

Agora sim vem as configurações de conexão com o banco de dados

```
// Agora sim a conexão
mongoose.connect('mongodb://db/mydb') //Atente para o nome 'db', que será um se
rviço no compose

// Teste
server.get('/', (req, res, next) => res.send('Backend'))

//Inicializando o Server
server.listen(3000)
```

NOTA: Se entrar na página da aplicação, que será em 'localhost' e aparecer a palavra: 'Frontend', isso atestará que a parte frontend está funcionando. Se entrar na aplicação no modo backend e aparecer a palavra 'Backend' irá indicar que a parte backend está funcionando.

2.2 Editando o arquivo 'docker-compose.yml'

Já estando no arquivo 'docker-compose.yml' a primeira coisa a ser definida é a versão do Docker-compose a ser usada, que será a versão 3.

2.2.1 Definição da versão do Docker-compose

```
version: '3'
```

2.2.2 Definição dos serviços

2.2.2.1 Definição do serviço 'db' com a imagem do mongodb, na versão 3.4

```
services:
  db:
    image: mongo:3.4
```

2.2.2.2 Definição do serviço 'backend':

Veja a partir da linha 5, da imagem de código a seguir.

- A. Será utilizada a imagem do **NODE**, na versão 8.1;
- B. Criação dos volumes: - ./backend:/backend
- C. Portas: - 3000:3000
- D. Com a execução do comando → command: bash -c "cd /backend && npm i && node app"

Em que serão executados 3 comandos de uma vez:

1. cd /backend, para entrar na pasta do volume criado;
2. npm i, para instalar as dependências; e
3. node app, para executar a aplicação, ou seja, o arquivo 'app.js'

```
1  version: '3'
2  services:
3    db:
4      image: mongo:3.4
5    backend:
6      image: node:8.1
7      volumes:
8        - ./backend:/backend
9      ports:
10       - 3000:3000
11     command: bash -c "cd /backend && npm i && node app"
12
13
```

NOTA: A porta 3000 já é padrão do node e foi mantida.

2.2.2.3 Definição do serviço 'frontend':

Veja a partir da linha 12, da imagem de código a seguir.

- A. Será utilizada a imagem do **NGINX**, na versão 1.13;
- B. Criação dos volumes: - ./frontend:/usr/share/nginx/html/ ;
- C. Portas: - 80:80

```

1  version: '3'
2  services:
3    db:
4      image: mongo:3.4
5    backend:
6      image: node:8.1
7      volumes:
8        - ./backend:/backend
9      ports:
10       - 3000:3000
11      command: bash -c "cd /backend && npm i && node app"
12    frontend:
13      image: nginx:1.13
14      volumes:
15        - ./frontend:/usr/share/nginx/html/
16      ports:
17        - 80:80

```

2.2.3 Efetuando testes

Agora, para essa fase inicial, devem ser feitos testes, tanto da parte Frontend, como da parte Backend:

2.2.3.1 Subindo a aplicação no Docker

Estando na pasta: node-mongo-compose, digite o seguinte comando a partir do terminal:

```
> docker-compose up -d <ENTER>
```

Aguarde, pois todas as dependências serão baixadas, caso não existam na máquina, então instaladas e a aplicação “app.js” será executada.

2.2.3.2 Testando o frontend

Abra o navegador e na barra de endereço, digite o seguinte:

Localhost <ENTER>

Se aparecer a palavra “Frontend”, significa que o servidor Web Nginx de seu projeto subiu normalmente.

2.2.3.3 Testando o backend

Ainda no navegador, digite o seguinte endereço:

Localhost:3000 <ENTER>, note a porta 3000, definida no Docker-compose.yml.

Se aparecer a palavra “Backend”, significa que o servidor backend de seu projeto subiu normalmente, ou seja, além do fato do Node estar funcionando, foi efetuada a conexão com o banco de dados MongoDB normalmente, pois só funcionaria se essa conexão tivesse sido efetuada com êxito.

Assim, está garantido que todos os 3 containers dessa aplicação estão funcionando normalmente.

Obviamente, o Frontend ainda não acessa o Backend, mas isso será resolvido na FASE 3 desse projeto.

3 Finalizando o cadastro – FASE 3

Agora será feita a implementação do backend, ou seja, dos web services, usando node, bem como uma página para que se possa consumir essa API criada a partir do Node.

Ressaltando que há 3 containers que foram implementados até a FASE 2, levantados a partir do compose.

Obviamente, por se tratar de uma aplicação simples, algumas dessas implementações serão feitas em um único arquivo. Em uma situação mais profissional, as implementações seriam quebradas em vários outros arquivos.

3.1 Implementação final do Backend

3.1.1 Instalando mais duas bibliotecas

Serão necessárias mais duas dependências para essa aplicação, que dizem respeito do bodyParser e ao Cors, conforme pode ser visto nas linhas: 5 e 6, na imagem a seguir.

```
1  const express = require('express')
2  const restful = require('node-restful')
3  const server = express()
4  const mongoose = restful.mongoose
5  const bodyParser = require('body-parser')
6  const cors = require('cors')
```

Sendo que:

- O bodyParser se faz necessário para fazer um parser. Veja citação abaixo:

Para manipular a solicitação HTTP POST em Express.js versão 4 e acima, você precisa instalar o módulo de middleware chamado body-parser.

body-parser extrai a parte do corpo inteiro de um fluxo de solicitação de entrada e o expõe em req.body.

O middleware era parte do Express.js anterior, mas agora você precisa instalá-lo separadamente.

Esse módulo body-parser analisa os dados codificados JSON, buffer, string e URL enviados usando a solicitação HTTP POST. Instale body-parser usando NPM como mostrado abaixo.

- Já o Cors será preciso, já que será acessado o backend a partir de um frontend.

3.1.2 Excluindo linhas de testes

Serão **excluídas** as linhas de teste das FASES anteriores, conforme abaixo:

```
17 //Teste
18 server.get('/', (req, res, next) => res.send('Backend'))
```

3.1.3 Definição de Middlewares

Para a declaração dos “Middlewares” necessários para o funcionamento da aplicação, como pode ser visto a seguir...

```
20 //Middlewares
21 server.use(express.urlencoded({extended:true}))
22 server.use(express.json())
23 server.use(cors())
```

Sendo que:

express.urlencoded... → Para fazer o parser do formato ‘urlencoded’, que vem no corpo da requisição; justamente quando se submete um formulário HTML. Daí a necessidade desse filtro. Até a versão 4.16.0 do Express usava-se: “bodyParser.urlencoded...” a partir dessa versão, usa-se:

express.json... → Para fazer o parser do json;

server.use(cors()) → E um filtro também para o Cors.

NOTA: PARA ESSA APLICAÇÃO SERÁ UTILIZADA A VERSÃO ANTERIOR DO PARSER, DEVIDO A COMPATILIDADE COM OUTRAS DEPENDÊNCIAS, CONFORME ABAIXO:

```
20 //Middlewares
21 server.use(bodyParser.urlencoded({ extended: true }))
22 server.use(bodyParser.json())
23 server.use(cors())
```

3.1.4 Mapeamento Objeto/Documento (ODM)

Será feito o mapeamento de uma única entidade de nome “Client” com um único atributo, sendo este do tipo String e requerido. Poderia ser qualquer nome para essa entidade, desde de que tenha relevância com a aplicação.

```

25 //ODM - Mapeamento Objeto/Cliente
26 const Client = restful.model('Client', {
27   name: { type: String, required: true }
28 })

```

3.1.5 Criação da API REST (Rest API)

Este padrão 'Client.methods' que vem do 'Node Restful'. Ele vai fazer a integração entre a parte de persistência usando o banco de dados Mongo com a parte do Express para criar as rotas, rotas essas para fazer o Crud: Get, Post, Put, Delete e Find by Id), baseado nos métodos listados abaixo.

Tratam-se das rotas, como rota padrão para listar (...), para incluir (POST), para alterar (PUT), para excluir registros (DELETE)...

```

30 //Rest API
31 Client.methods(['get', 'post', 'put', 'delete']) //Array de métodos
32 Client.updateOptions({new: true, runValidators: true}) /* Essa linha é apenas referência,
33 para caso haja necessidade de validação das entradas ou tratamentos de erros */

```

Sendo assim, serão criadas todas as rotas.

NOTA: A linha 32 foi colocada apenas como referência, pois como essa aplicação é simples não haverá validação e/ou tratamento de exceções.

3.1.6 Declaração das Rotas

Serão registradas as todas as rotas, com uma única chamada, usando como base, ou seja, como raiz da rota: '/clients'.

```

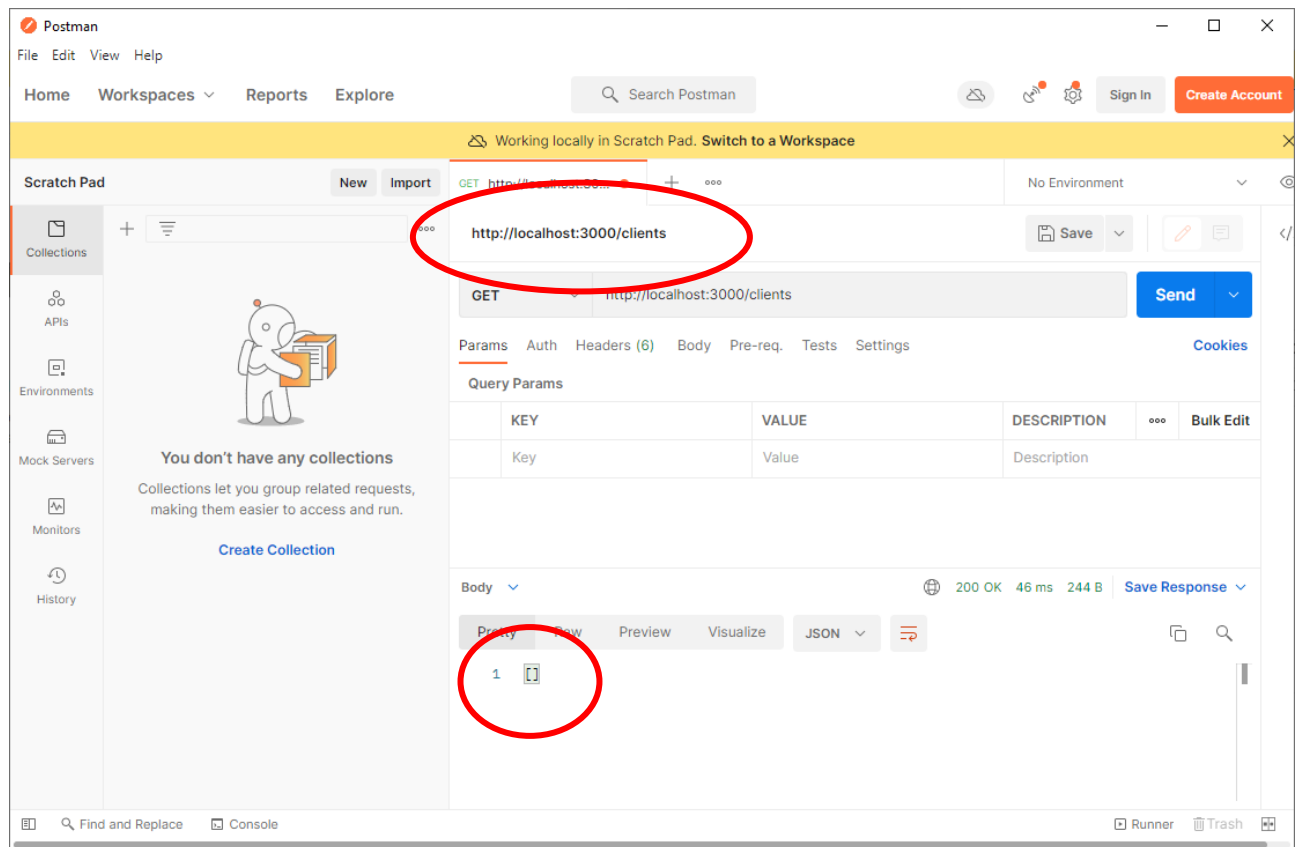
35 // Routes
36 Client.register(server, '/clients')

```

Para efetuar um teste, Salve todos os arquivos, suba sua aplicação no Docker com a instrução:

> Docker-compose up -d <ENTER>, então, abra o Postman® e teste com uma requisição GET, conforme imagem abaixo:

Se após a execução retornar um ARRAY vazio ("[]"), indica que a rota GET já está funcionando, conforme a implementação do código até o presente momento.



3.2 Implementação final do Frontend

Agora o arquivo visado é o index.html, em que será inserido um código base, bem simples para a implementação do Frontend, conforme pode ser visto abaixo:

```

1  <!DOCTYPE html>
2  <html lang="PT-BR">
3
4  <head>
5      <meta charset='utf-8'>
6      <title>Cadastro Simples</title>
7      <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
8  </head>
9
10 <body>
11     <div class="container">
12         <h1>Cadastro Simples</h1>
13         <hr>
14         <div>
15             <input name='id' type='hidden' />
16             <div class="form-group">
17                 <label for="Name"></label>
18                 <input class='form-control' name='name' placeholder='Digite aqui o nome...' />
19             </div>
20             <button class='btn btn-success' save>Salvar</button>
21         </div>
22         <!-- Abaixo, uma Tabela que será criada dinamicamente usando jquery -->
23         <table class="table" id='clients'>
24             <thead>
25                 <tr>
26                     <th>Nome</th>
27                     <th>Ações</th>
28                 </tr>
29             </thead>
30             <tbody id="clientsRows"></tbody>
31         </table>
32     </div>
33
34     <script src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js'></script>
35
36     <script>
37
38     </script>
39 </body>
40 </html>

```

3.2.1 Explicação do código HTML

Na linha 7 foi adicionada uma referência oficial do CDN do Bootstrap, referente ao CSS;

Foi adicionado um formulário simples, com o ID escondido, como pode ser visto na linha 15;

Há um botão para salvar os dados digitados, com a propriedade específica “save”;

A partir da linha 23 há a estrutura de uma tabela que será criada dinamicamente, usando JQuery. Obviamente existem excelentes frameworks que auxiliam na criação de sistemas como: Vue, React e Angular, por exemplo.

Então fica claro que o conteúdo da Tabela será gerado de forma dinâmica;

Na linha 34 foi adicionada uma referência do Javascript;

3.2.2 Codificando

Para a codificação, como objetivo, será criado o script para consumir a API, gerar a Tabela e fazer o cadastro funcionar.

3.2.2.1 Caminho da API

A primeira coisa a ser feita é a definição de uma constante que será o caminho da API;

```
36 <script>
37 //Caminho da API
38 const API = 'http://localhost:3000'
```

3.2.2.2 Definição do método 'getClients()' que carrega os clientes cadastrados.

Esse método irá carregar os clientes cadastrados. Trata-se de uma chamada AJAX, que fará a carga.

```
40 const getClients = () =>{
41   $.ajax({
42     url: `${API}/clients`,
43     success: clients => {
44       console.log(clients)
45     }
46   })
47 }
```

3.2.2.3 Chamada do método getClients()

Chamada do método getClients(), para que sejam carregados os clientes, quando a página for carregada.

```
49 // Com o código abaixo, Quando a página for carregada, será chamado o método getClients
50 $((() => {
51   getClients()
52 }))
```

3.2.2.4 Criação do método saveClient()

Agora será criado o método saveClient(), que pega o valor do campo ID e também o valor do campo nome da página HTML.

```

49  const saveClient = () => {
50      const _id = $('[name=id]').val()
51      const name = $('[name=name]').val()
52      /* Na chamada AJAX abaixo, neste mesmo método, serão efetuadas duas chamadas
53      será feita uma chamada POST, caso o ID não esteja preenchido, para se fazer
54      uma inserção, e PUT, caso o ID esteja "setado", para se fazer uma alteração.*/
55      $.ajax({
56          method: _id ? 'PUT' : 'POST',
57          url: `${API}/clients/${_id}`,
58          data: _id ? { _id, name } : { name },
59          success: getClients
60      })
61  }

```

Explicando o código acima:

- Nas linhas 50 e 51 será pegos os dados dos campos do formulário HTML;
- Na linha 50, a constante criada como “_id”, tem um propósito de ser assim, pois lá no banco de dados MongoDB, este campo está definido dessa forma;
- Na linha 55 começa uma chamada AJAX e já na linha 56, há uma dupla chamada neste método, para essa chamada AJAX, em que caso o ID esteja vazio o método será direcionado para INSERÇÃO de dados e, caso o ID não esteja definido, então a chamada passa a fazer a alteração do dado;
- Na linha 57, há a definição da URL, sendo passado o caminho da API, lembrando que a mesma é uma constante com todo o caminho da aplicação, bem como também informado o ID do cliente;
- Na linha 58, determina quais dados serão encaminhados, sendo que: caso o ID não esteja definido, serão passados tanto o ID como o nome e, caso o ID já esteja definido, apenas o campo nome será encaminhado.

3.2.2.5 Associando o clique do botão Salvar com a função saveClient()

Agora é necessário associar a ação do clique do botão Salvar com a função que foi definida acima, a função saveClient(). Para tal, proceda com a codificação abaixo:

```

63      /* Com o código abaixo, Quando a página for carregada, será chamado
64      o método getClients */
65      $((() => {
66          getClients()
67          //Associação do botão com a função saveclient()
68          $('[save]').click(saveClient)
69      })

```

Explicação:

- Observe na linha 68, do código acima, a associação do atributo “save”, pertencente ao Botão “Salvar”, com o método saveClient().

NOTA: da mesma forma que se acessa o atributo de um botão, com o ID por exemplo, se acessa outra TAG HTML com um atributo personalizado.

3.2.2.6 Fazendo a tabela ser “renderizada” de forma correta – CRIAÇÃO DOS BOTÕES

Agora, a Tabela definida no corpo do arquivo index.html, deverá ser renderizada de forma dinâmica e correta, com a inserção de botão para atualização e exclusão, ou seja, a tabela vai ter o nome do cliente, o botão de alterar e o botão de excluir. Assim, o código será inserido logo abaixo da definição da constante API, que está localizada no início do SCRIPT.

Então à função para essa funcionalidade:

```
37 //Caminho da API
38 const API = 'http://localhost:3000'
39
40 //Criação do método para gerar o botão
41 const createButton = (label, type) => {
42   //Abaixo a forma de se criar em JQuery um elemento dinamicamente
43   return $('<button>').addClass(`btn btn-${type}`).html(label)
44 }
```

Explicação:

- A. O método createButton(), foi criado abaixo da constante API;
- B. Este método recebe o “label” e o “type” como parâmetros;
- C. Na linha 43, pode ser visto a forma de criação, em JQuery, de um elemento dinamicamente.
- D. Ainda na linha 43, pode ser visto a aplicação de uma classe com “type”, que será o seu tipo atribuído, nessa template string, bem como o conteúdo do botão em html.

3.2.2.7 Fazendo a tabela ser “renderizada” de forma correta – RENDERIZANDO AS COLUNAS DA TABELA

Agora será criado um método que fará a renderização das colunas da tabela existentes na página HTML.

```
46 const renderRows = clients => {
47   const rows = clients.map(client => {
48     const updateButton = createButton('Atualizar', 'warning')
49     const removeButton = createButton('Remover', 'danger')
50
51     return $('<tr>')
52       .append($('<td>').append(client.name))
53       .append($('<td>').append(updateButton).append(removeButton))
54   })
55
56   $('#clientsRows').html(rows)
57 }
```


Explicação:

- A. Será criado o método `renderRows()`, que irá receber a lista de clientes, como pode ser visto na linha 46, acima;
- B. Já na linha 47, será capturada uma lista de clientes, de objetos em JSON, que será transformada em uma lista de componentes, ou seja, uma lista de linhas na tabela HTML; para que isso seja feito, é usado o método de array: `map()`; assim, cada cliente irá gerar uma linha;
- C. Nas linhas 48 e 49 estão sendo criados os botões que aparecerão na tabela dinamicamente; para tal, está sendo chamado o método `createButton()`, com os parâmetros de características dos botões;
- D. A partir da linha 51 estão sendo geradas as linhas, que serão criadas com a concatenação – `append` – das TAGs HTML de criação de linhas em tabela, bem como com a concatenação do campo nome, juntamente com os botões de atualizar e excluir;
- E. Para finalizar o método `renderRows()`, na linha 56 da imagem acima, será pego o “**rows**”, que é o resultado da conversão do array, feito pelo “`map()`”, colocando esse resultado dentro da **TAG tbody**, que possui “**ID=clientsRows**”;

3.2.2.8 Fazendo a tabela ser “renderizada” de forma correta – CHAMANDO A FUNÇÃO `renderRows()`

Para que de fato funcione, será necessário fazer a chamada do método `renderRows()`, passando a lista de clientes obtida a partir do método `getClients()`; essa chamada é feita dentro do próprio método `getClients()`, como pode ser visto na linha 63 da imagem abaixo:

```
59      const getClients = () => {
60          $.ajax({
61              url: `${API}/clients`,
62              success: clients => {
63                  renderRows(clients)
64              }
65          })
66      }
```

3.2.2.9 Criando a função de remoção – função `removeClient()`

O método de exclusão é feito a partir de uma chamada AJAX, como pode ser visto na imagem abaixo:

```

59     const removeClient = client => {
60         $.ajax({
61             method: 'DELETE',
62             // Abaixo a forma padrão do REST para chamada de uma exclusão
63             url: `${API}/clients/${client._id}`,
64             /* Para atualizar a lista de clientes, é feita a
65             chamada do método getClients() */
66             success: getClients
67         })
68     }

```

Explicação:

- A. Na linha 59, é feita a criação do método removeClient(), que recebe como parâmetro 1 “client” – cliente;
- B. A partir da linha 60, é feita uma chamada AJAX;
- C. Na linha 61 é definido o método de exclusão: DELETE;
- D. A URL que definirá a exclusão é vista na linha 63, em que é passado o caminho da API, já com o ID do cliente a ser excluído;
- E. O método removeClient() é finalizado com o método success(), em que é feita a atualização da lista, já tendo sido efetivada a exclusão do cliente.

3.2.2.10 Fazendo o botão Remover funcionar

Obviamente, embora a função de exclusão tenha sido criada, o botão ainda está sem ação, pois é necessário fazer a associação do botão com essa função e isso pode ser visto na imagem abaixo:

```

46     const renderRows = clients => {
47         const rows = clients.map(client => {
48             const updateButton = createButton('Atualizar', 'warning')
49             const removeButton = createButton('Remover', 'danger')
50             // Chamando a função removeButton, passando o cliente como parâmetro
51             removeButton.click(() => removeClient(client))
52
53             return $('<tr>')
54                 .append($('<td>').append(client.name))
55                 .append($('<td>').append(updateButton).append(removeButton))
56         })
57
58         $('#clientsRows').html(rows)
59     }

```

Explicação:

- A. Dentro do método renderRows(), na linha 51, note a criação da instrução que fará o botão excluir funcionar, em que ocorre a associação do clique do botão removeButton, com a função removeClient(), no qual é passado como parâmetro o cliente a ser excluído.

3.2.2.11 Criando a função de Atualização – função loadClient()

Para que seja feita a atualização, é preciso pegar a linha do cliente corrente, assim que seja clicado no botão atualizar e atualizar o formulário; também é necessário atualizar o campo oculto “id”, ou seja, o campo: `<input name='id' type='hidden' />`

É a partir do ID oculto no formulário que será efetuada a atualização do registro.

A função que permite efetuar a atualizar é escrita acima da função `removeClient()` e é vista na imagem abaixo:

```
61 |      const loadClient = client => {
62 |          $('[name=id]').val(client._id)
63 |          $('[name=name]').val(client.name)
64 |      }
```

Explicação:

- A. Essa função é criada acima do método `removeClient()`;
- B. Na linha 61, do código presente na imagem acima é criado o método `loadClient()`, que recebe como parâmetro o “cliente” corrente;
- C. Então é passado o valor ID que recebeu de “`client._id`”, na linha 62;
- D. Também é passado o valor do nome, que foi recebido de “`client.name`”, como pode ser visto na linha 63.

3.2.2.12 Fazendo o botão Atualizar funcionar

Agora é preciso fazer a associação do botão com a função `loadClient()`; para tal, observe o código na imagem a seguir:

```
46 |      const renderRows = clients => {
47 |          const rows = clients.map(client => {
48 |              const updateButton = createButton('Atualizar', 'warning')
49 |              //Chamando a função loadClient, passando o cliente como parâmetro
50 |              updateButton.click(() => loadClient(client))
51 |          })
52 |          You, 3 minutes ago • Uncommitted changes
53 |          const removeButton = createButton('Remover', 'danger')
54 |          //Chamando a função removeButton, passando o cliente como parâmetro
           removeButton.click(() => removeClient(client))
```

Explicação:

Na linha 50, logo abaixo da criação da constante “`updateButton`”, ocorre a criação da associação do clique do botão “Atualizar” com o método de atualização `loadClient(cliente)`, passando como parâmetro o cliente corrente;

NOTA: O cliente a ser alterado é de fato alterado e não faz a criação de novo registro, pois no método `saveClient()`, criado anteriormente, já estava preparada para lidar tanto com “POST” – para inserir registro, como com “PUT”, para alterar registro.

3.2.2.13 Finalizando, configurando a limpeza do campo de formulário

Para que seja feita a limpeza do campo, bem como definir o posicionamento do cursor no campo nome, digite as seguintes instruções das **linhas 86 e 87** no método `getClients()`, conforme pode ser visto na imagem abaixo:



```
81      const getClients = () => {
82        $.ajax({
83          url: `${API}/clients`,
84          success: clients => {
85            renderRows(clients)
86            $('[name]').val('')
87            $('[name]').focus()
88          }
89        })
90      }
```

3.2.2.14 Última modificação, definindo o foco do cursor ao atualizar

É necessário também posicionar o cursor no campo nome, ao fazer a atualização de um registro. Então, no método `loadClient()`, digite o código de instrução da linha 67, como pode ser visto na imagem a seguir:

```
64      const loadClient = client => {
65        $('[name=id]').val(client._id)
66        $('[name=name]').val(client.name)
67        $('[name]').focus()
68      }
```

3.2.2.15 Tela do sistema de cadastro simples finalizado

  localhost

Cadastro Simples

Salvar

Nome	Ações
Ana Paula da Silva Santos	<div>Atualizar</div> <div>Remover</div>
Pedro Alterado	<div>Atualizar</div> <div>Remover</div>
Gabriela Alves de Oliveira	<div>Atualizar</div> <div>Remover</div>
Carla Peres	<div>Atualizar</div> <div>Remover</div>