Imersão no Node.JS

Desenvolvendo uma API RESTful em Node.JS e MySQL

Objetivos

- Iniciando um Projeto em Node.JS
- Criando a API
- Criando a listagem de clientes
- Criando a pesquisa de um cliente
- Excluindo um cliente
- Adicionando um cliente
- Atualizando um cliente

Iniciando um Projeto em Node.JS

Agora que você já tem o banco pronto, vamos estabelecer uma conexão do Node com o MySQL. Não pule esta etapa pois vamos fazer tudo isso usando Node.JS.

Crie uma pasta para guardar os arquivos do seu projeto Node.JS, você pode fazer isso pelo console se quiser, usaremos ele algumas vezes nesse tutorial. No exemplo abaixo, criei a pasta e depois entrei dentro dela.

```
> mkdir nodemysql

> cd nodemysql
```

Agora execute no console o comando **"npm init"** que o próprio NPM (gerenciador de pacotes do Node) vai te guiar para a construção do arquivo **package.json**, que é o arquivo de configuração do projeto.

```
nodemysql> npm init
```

```
package.json:
{
 "name": "nodemysql",
 "version": "1.0.0",
 "description": "Node com MySQL para o Curso de Desenvolvimento WEB da PUD",
 "main": "index.js",
 "scripts": {
  "test": "echo \\\"Error: no test specified\\\" && exit 1"
 },
 "keywords": [
```

```
  "Node",
  "MySQL",
  "JavScript",
  "Desenvolvimento",
  "Web",
  "PUD"
 ],
 "author": "Luis.Alvarenga",
 "license": "ISC"
 }
```

Com o arquivo de configurações criado, vá no console novamente, na pasta do projeto e digite o seguinte comando para instalar a extensão mysql, que permite usar Node com MySQL:

```
nodemysql> npm install -S mysql
```

A flag “-S” diz que é para salvar essa dependência no arquivo **packages.json**. Se você abrir o arquivo vai ver que tem coisa nova por lá.
Agora, crie um arquivo **popular-tabelas.js** dentro dessa pasta. Ele vai popular as tabelas que usaremos.

Vamos começar nosso **popular-tabelas.js** definindo uma constante para a String de conexão com o banco e uma constante para o objeto que vai carregar a extensão mysql.

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

const uriConnection = {
  host: 'localhost',
```

```
  port: 3306,
  user: 'root',
  password: '123456',
  database: 'LOJA'
};

const connection = mysql.createConnection(uriConnection);
```

Agora, usaremos esse objeto **connection** para fazer uma conexão e, em caso de sucesso, imprimir uma mensagem de sucesso. Caso contrário, se der erro, uma mensagem de falha:

```
connection.connect(function (err) {
  if (err)
    return console.log(err); console.log('conectou!');
});
```

Vamos entender o código. O objeto **connection** permite que façamos coisas no banco de dados, uma delas é a conexão (connect). No entanto, o Node.js trabalha de maneira assíncrona, o que quer dizer que **ele não espera pela conexão ser estabelecida**. Quando a conexão for estabelecida, a função de **callback,** passada por parâmetro, será executada e conterá, ou não, um objeto de erro.

Para executar esse arquivo, abra o console (se estiver usando **VS Code**, apenas aperta F5 com este arquivo aberto no editor) e na pasta do projeto digite:

```
nodemysql> node popular-tabelas.js
```

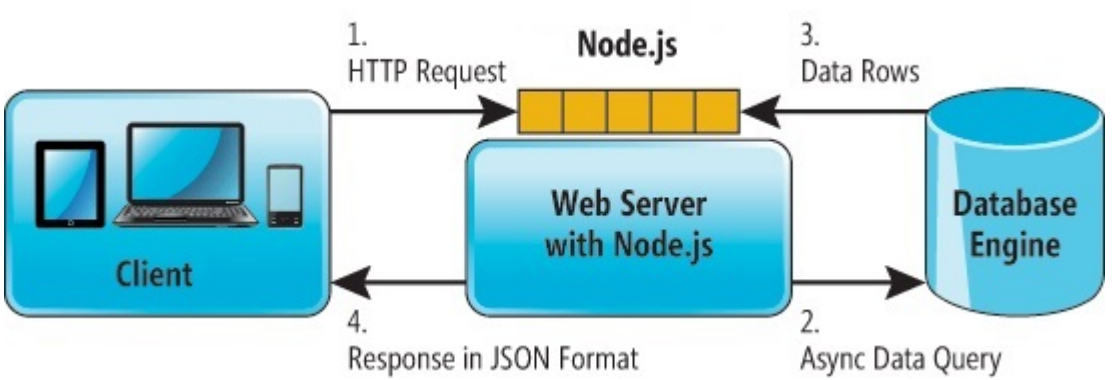
Agora que sabemos como conectar ao MySQL através de Node.js, é hora de executarmos o comando que vai popular as tabelas.

Coloque a chamada desta função após a conexão no banco, passando o objeto **conn** por parâmetro, conforme abaixo:

```
function addRows(conn) {
  const sql = "INSERT INTO usuarios(usuario, senha, status) VALUES ?";
  const values = [['teste1', '123456', '1'], ['teste2', '123456', '1'], ['teste3', '123456', '1']];

  conn.query(sql, [values],
    function (error, results, fields) {
      if (error)
        return console.log(error);

      console.log('adicionou registros!');
    }
  );
}

addRows(connection);

connection.end();
```

Execute o script novamente e verá que ele populará a sua tabela com sucesso.

Criando a API



Agora que já temos nosso banco de dados **MySQL** pronto e aprendemos como nos conectar a ele, vamos criar uma API básica usando **Express** para criarmos uma aplicação **CRUD** (create, read, update & delete).

Vamos começar adicionando a dependência do **Express** (framework web) e do **Body-Parser** (conversor de formatos de dados) no projeto via linha de comando. Na pasta do projeto, digite:

```
nodesqlserver> npm install -S express body-parser
```

Na sequência, vamos criar um arquivo **index.js** na pasta do projeto, onde criaremos o nosso servidor da **API** para o tratamento das requisições.

Vamos começar bem simples, apenas definindo as constantes locais que serão usadas à frente:

```
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const port = 3000;
const mysql = require('mysql');
```

Agora, vamos configurar nossa aplicação **express** (app) para usar o **body parser** que carregamos, permitindo que recebamos requisições nos formatos **URLEncoded** e **JSON**:

```
//configurando o body parser
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Na sequência, vamos criar um roteador e dentro dele definir uma regra inicial que exibe uma mensagem de sucesso quando o usuário requisitar a raiz da API (/) pelo metodo **GET**.

```
//definindo as rotas
const router = express.Router();

// rota raiz
router.get('/', (req, res) => res.json({ message: 'Funcionando!' }));
app.use('/', router);
```

Na última linha indicamos que as requisições que chegarem na raiz (“/”) devem ser tratadas pelo roteador.

Por fim, adicionamos as linhas abaixo que iniciam o servidor e o deixa aguarando requisições na porta 3000:

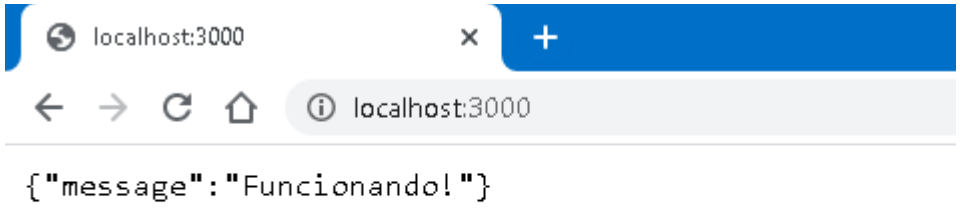
```
//inicia o servidor
app.listen(port);
console.log('Servidor iniciado e aguardando conexões na porta 3000');
```

Teste sua **API** executando via console o seu script **index.js**.

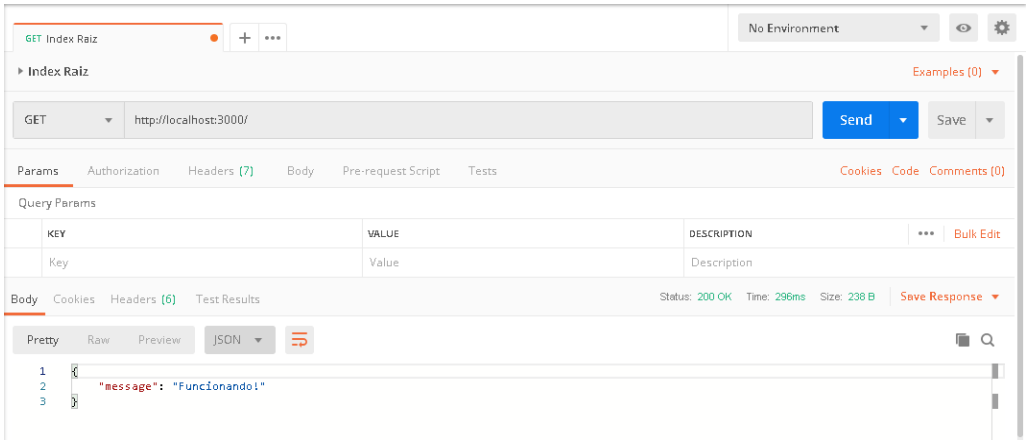
```
nodesqlserver> node index.js
```

Você deve ver a mensagem de **'Servidor iniciado e aguardando conexões na porta 3000'** no console.

Acesse o endereço **http://localhost:3000/** no navegador, que retornará uma mensagem em formato **JSON** que deixamos na rota raiz.



Ou acesso o endereço pelo **postman (*):**



(*) https://www.getpostman.com/product/api-client

Criando a listagem de clientes

Agora que temos uma API funcionando, vamos adicionar uma rota **/clientes** que listará todos os clientes do banco de dados.

Para isso, vamos criar uma função que executará as consultas **SQL** no banco de dados usando uma conexão que será criada a cada uso.

Existem técnicas mais avançadas (como o pool de conexões) que oferecem maior performance, mas por ora, isso resolve satisfatoriamente, como abaixo:

```
var uriConnection = {
  host: 'localhost',
  port: 3306,
  user: 'root',
  password: '123456',
  database: 'LOJA'
};

function execSQLQuery(sqlQry, values, res) {

  const connection = mysql.createConnection(uriConnection);

  connection.query(sqlQry, values,
    function (error, results, fields) {
      if (error)
```

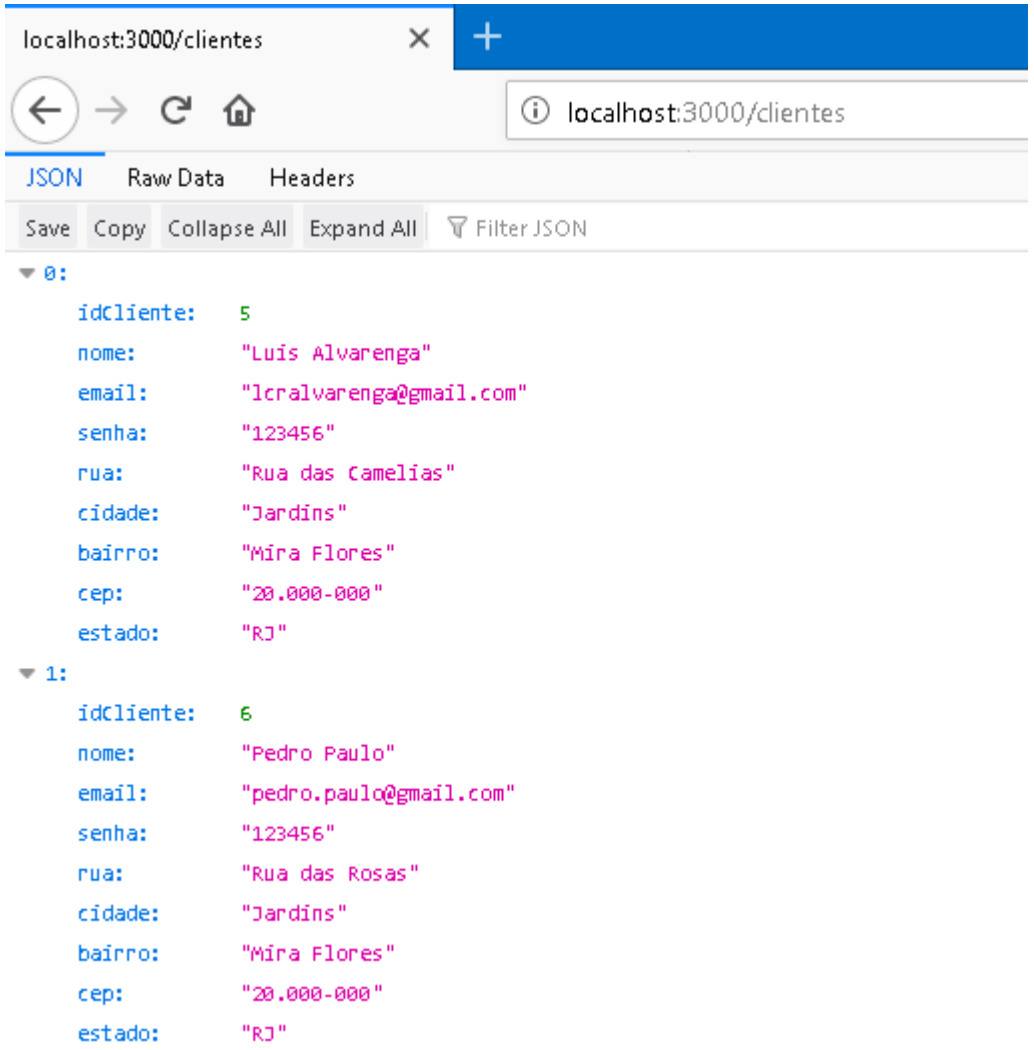
```
        res.json(error);
      else
        res.json(results);
      connection.end();
      console.log('Executou: [SQL = ' + sqlQry + '] - [Values = ' + values + ']');
    }
  );
}
```

Esta função pode ficar no final do seu arquivo **index.js** e nós a usaremos para fazer todas as operações de banco de dados da nossa API.

Começaremos criando a rota **/clientes** logo abaixo da rota **/** (raiz):

```
// rota clientes
router.get('/clientes',
      (req, res) => {
            execSQLQuery('SELECT * FROM Clientes', res);
      }
);
```

Agora, ao executarmos novamente nosso projeto e ao acessarmos a URL **http://localhost:3000/clientes**, veremos todos os clientes cadastrados no banco de dados:



E com isso finalizamos a listagem de todos clientes na nossa API!

Criando a pesquisa de um cliente

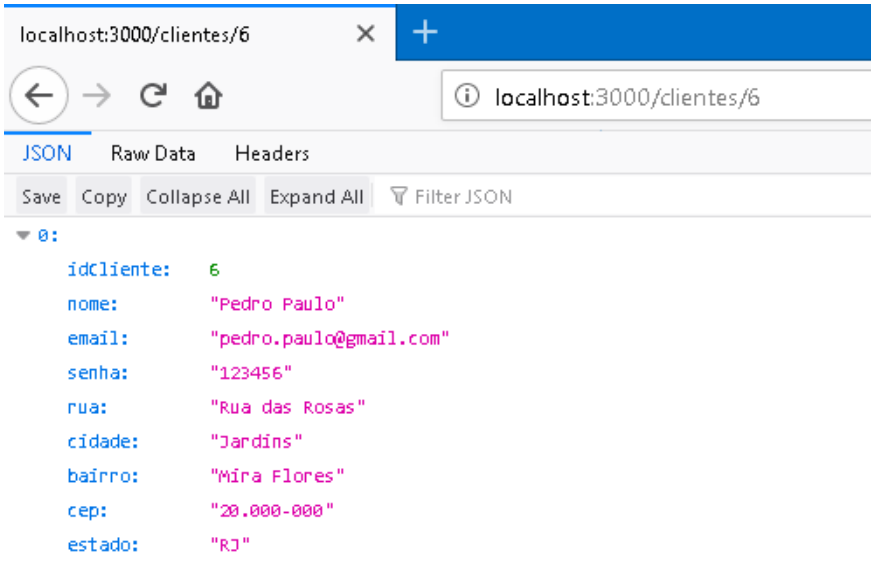
Agora, se o usuário quiser ver apenas um cliente, ele deverá passar o **ID** (identificador do clientes – coluna: **loja.clientes.idCliente**) na **URL**, logo após a rota **/clientes**.

Vamos modificar a rota **/clientes** criada no passo anterior para **aceitar um parâmetro opcional ID**. No processamento da rota, se o ID estiver preenchido, devemos fazer uma consulta diferente da anterior, como mostra o código abaixo:

```
// rota clientes
router.get('/clientes/:id?',
  (req, res) => {
    let sqlQry = 'SELECT * FROM Clientes ';
    let values = [];

    if (req.params.id) {
      sqlQry += 'WHERE idCliente = ?';
      values = [parseInt(req.params.id)];
    }

    execSQLQuery(sqlQry, values, res);

  }
);
```

O **parseInt** é uma proteção contra **SQL Injection**, uma vez que neste caso o **ID** deve ser um número inteiro válido.

Ao executarmos o projeto e ao acessarmos a URL **http:///localhost:3000/clientes/6**, veremos o cliente cadastrado no banco de dados com o ID informado:



E com isso terminamos a pesquisa por cliente.

Excluindo um cliente

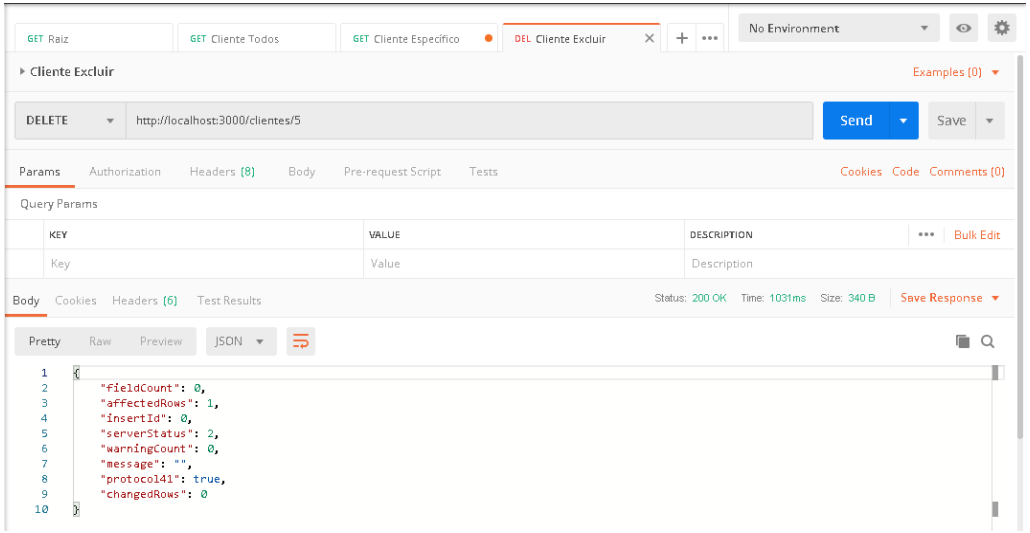
Para excluir um cliente vamos fazer um processo parecido com o de consultar, no entanto, mudaremos o método (também chamado de verbo) **HTTP** de **GET** para **DELETE**, como manda o protocolo.

```
router.delete('/clientes/:id',
  (req, res) => {
    const sqlQry = 'DELETE FROM Clientes WHERE idCliente = ?';
    const values = [parseInt(req.params.id)];

    execSQLQuery(sqlQry, values, res);
  }
);
```

Adicione a nova rota logo após as demais.

Note que **desta vez o parâmetro ID na URL não é opcional (uso da ? após :id)**.

Para testar essa rota você tem duas alternativas, ou usa o postman para simular o método DELETE do HTTP, como abaixo:



Ou via console usando cURL:

```
> curl -X DELETE http://localhost:3000/clientes/5
```

Em ambos os casos você deve obter uma resposta 200 OK, em caso de sucesso.

Adicionando um cliente

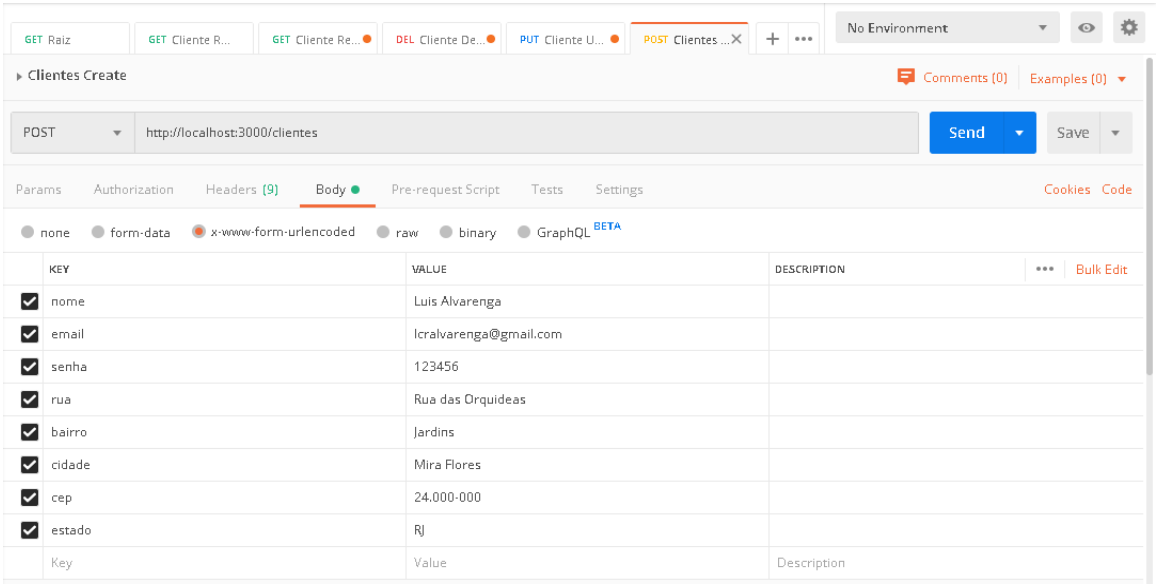
Agora vamos adicionar um novo cliente com um **POST** na rota **/clientes**. Adicione esta nova rota logo abaixo das anteriores.

```
router.post('/clientes',
  (req, res) => {

    const nome = req.body.nome;
    const email = req.body.email;
    const senha = req.body.senha;
    const rua = req.body.rua;
    const bairro = req.body.bairro;
    const cidade = req.body.cidade;
    const cep = req.body.cep;
    const estado = req.body.estado;

    const sqlQry = 'INSERT INTO loja.clientes (nome, email, senha, rua, bairro,
      cidade, cep, estado) VALUES (?)';
    const values = [[nome, email, senha, rua, bairro, cidade, cep, estado]];

    execSQLQuery(sqlQry, values, res);
  }
);
```

Recebemos as variáveis que vêm com o método POST e depois executamos o comando de INSERT no banco de dados.

Em uma aplicação profissional é necessário fazer as validações do tipo e conteúdo das variáveis antes de executar o comando INSERT no banco de dados.

Vamos utilizar o postman para testar este método:



Se quiser fazer via cURL:

```
> curl -X POST -d "Luis Alvarenga&email=lcralvarenga@gmail.com&senha=123456&rua=Rua das Orquideas&bairro=Jardins&cidade=Mira Flores&cep=24.000-000&estado=RJ" http://localhost:3000/clientes
```

Se testar agora vai ver que é possível inserir novos registros no banco de dados através de requisições POST.

Atualizando um cliente

E para finalizar o CRUD, vamos ver como podemos atualizar um cliente no banco de dados através da API.

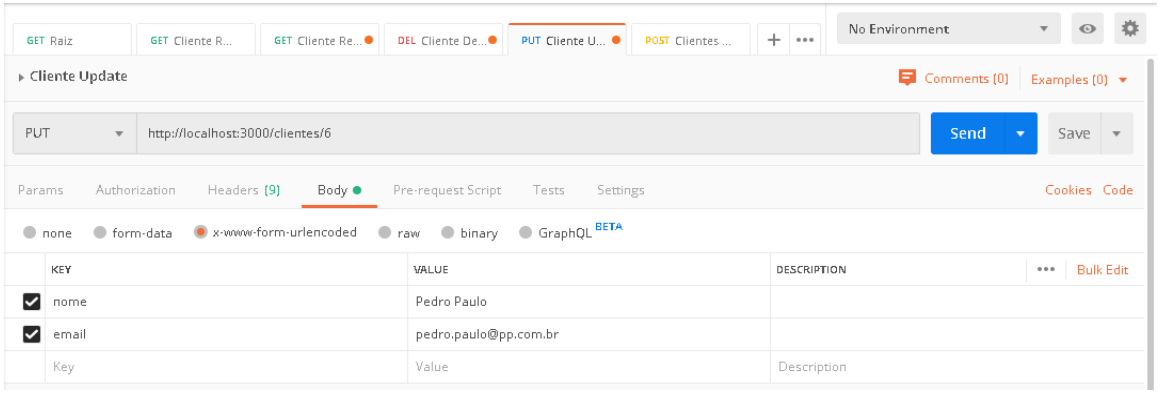
Para fazer updates podemos usar os verbos **PUT** ou **PATCH**. O protocolo diz que devemos usar **PUT** se pretendemos passar todos os parâmetros da entidade que está sendo atualizada, mas não vamos alterar jamais o ID, então usaremos **PATCH** na API.

Crie uma rota **PATCH** em **/clientes** esperando o **ID** do cliente a ser alterado.

```
router.patch('/clientes/:id',
  (req, res) => {

    const sqlQry = 'UPDATE Clientes SET nome = ?, email = ? WHERE idCliente = ?';
    const values = [req.body.nome, req.body.email, parseInt(req.params.id)];

    execSQLQuery(sqlQry, values, res);
  }
);
```

No código acima, pegamos o **ID** que veio na **URL** e as demais informações que vieram no corpo da requisição. Depois executamos o comando UPDATE no banco de dados.

Para testar uma requisição PATCH, você pode usar o postman:



Ou o cURL

```
> cURL -X PATCH -d "nome=Pedro Paulo da Silva&email=pedro.paulo@pp.com" http://localhost:3000/clientes/4
```

Note que se o ID não existir ocasionará um erro de banco de dados que será apresentado no corpo da resposta.

Assim, finalizamos o CRUD da nossa API Node.js que usa MySQL como persistência de dados.