

BIT PAGG LTDA.



DESENVOLVIMENTO DE PROXY EM NODE.JS

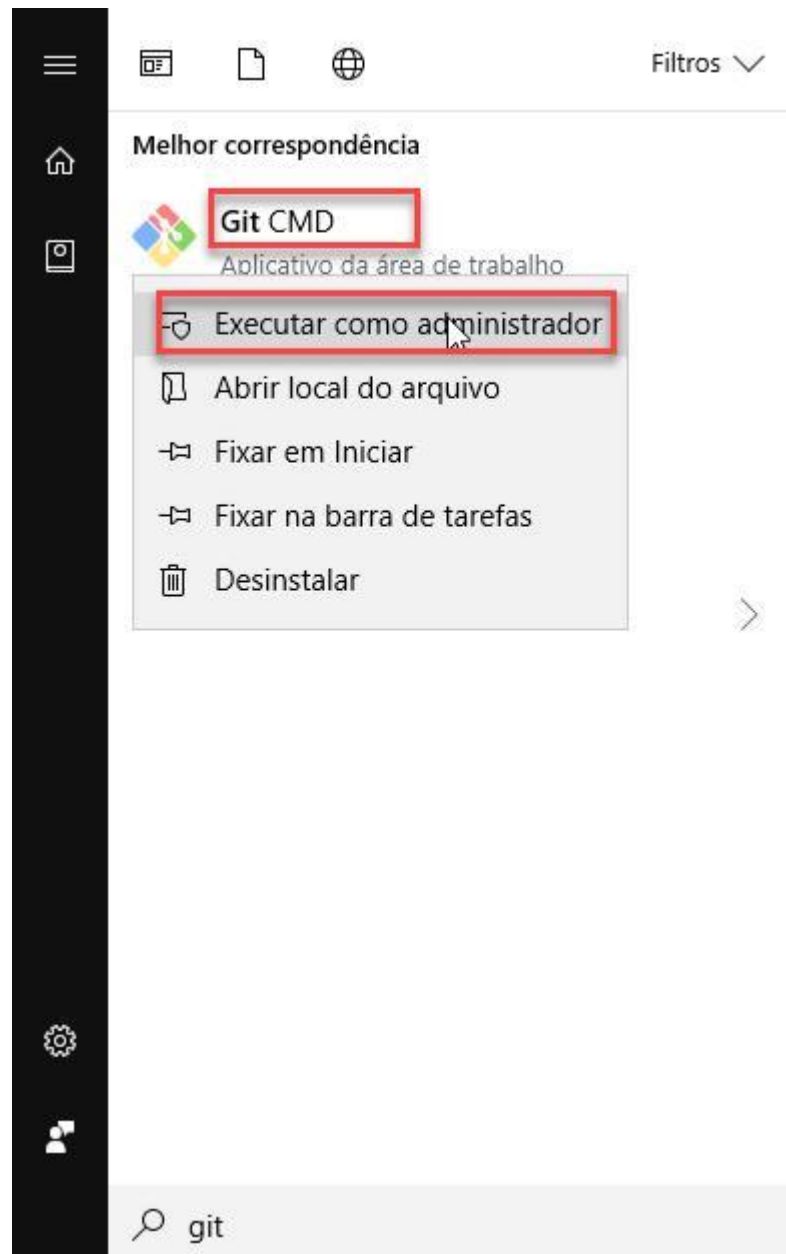
Araxá
2018

Sumário

1. Criação Proxy.....	3
2. Menu	6
2.1. Bibliotecas usadas	6
2.2. Criação arquivo Sever.js	7
2.3. Config.....	8
2.4. Routes.....	9
2.5. Métodos	10
2.5.1. Auth.js	10
2.5.2. Bitpagg_token.js.....	11
2.6. Controllers.....	12
2.6.1. Autenticador.js.....	12
2.6.2. Processo.js.....	13
3. Tratamento de erros.....	17
4. Criação Web.config.....	17
5. Possível problema Proxy Node.JS	18
5.1. Forma incorreta.....	18
5.2. Forma correta.....	18
5.3. Dados Header	18
6. Criação método Checkout.....	19
7. Observações	20

1. Criação Proxy

Para gerar a aplicação é necessária a abertura do **Git CMD** como administrador, pois iremos criar a aplicação por linhas de comando.



Ao abrir o Git em modo administrador, navegue até o diretório onde pretende salvar os arquivos que vamos criar, chegando no diretório digite o comando **“npm install”**, este comando irá criar o arquivo package.json, onde haverá informações sobre o node.js.

```

D:\Angular\RetornoMetodos\Node>npm install
npm WARN saveError ENOENT: no such file or directory, open 'D:\Angular\RetornoMetodos\Node\package.json'
npm WARN createLockfileAs package-lock.json. You should commit this file.
npm WARN enoent ENOENT: no such file or directory, open 'D:\Angular\RetornoMetodos\Node\package.json'
npm WARN Node No description
npm WARN Node No repository field.
npm WARN Node No README data
npm WARN Node No license field.

up to date in 6.176s
found 0 vulnerabilities

```

Em seguida digite o comando “**npm init**”, digitando este comando em seguida irá abrir algumas informações no prompt, essas informações são para efetuar alguma alteração nas configurações no **package-lock.json** que irá ser criado, aperte ENTER e então ele irá mudando de tópico, no tópico “description:” colocar uma descrição necessária em relação ao projeto que está sendo criado, no tópico “author:”, colocar o nome de quem está criando o projeto , na imagem abaixo e ilustrado os lugares onde terá a necessidade de apertar a tecla ENTER.

```

D:\Angular\RetornoMetodos\Node>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (node)
version: (1.0.0)
description:
git repository:
author:
license: (ISC)
About to write to D:\Angular\RetornoMetodos\Node\package.json:
{
  "name": "node",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.18.3",
    "config": "^2.0.1",
    "express": "^4.16.3",
    "helmet": "^3.13.0",
    "jsonwebtoken": "^8.3.0",
    "morgan": "^1.9.0",
    "uuid": "^3.3.2"
  },
  "devDependencies": {},
  "description": ""
}

Is this OK? (yes)

```

Ao término deste procedimento o próximo passo é criar as pastas para criação do projeto, a primeira pasta a ser criada será a “**app**”, com o seguinte comando.

```
D:\Angular\RetornoMetodos\Node>mkdir app
```

Após este comando navegue para dentro da pasta “**app**”.

```
D:\Angular\RetornoMetodos\Node>cd app
```

Estando dentro da pasta “**app**”, você criará a pasta controllers digitando o comando, “**mkdir controllers**”.

```
D:\Angular\RetornoMetodos\Node\app>mkdir controllers
```

Após o término da criação da pasta crie outra pasta com o nome métodos, com o seguinte comando, “**mkdir metodos**”.

```
D:\Angular\RetornoMetodos\Node\app>mkdir metodos
```

Em seguida crie a pasta models, com o seguinte comando “**mkdir models**”.

```
D:\Angular\RetornoMetodos\Node\app>mkdir models
```

Novamente crie outra pasta com nome de routes, com o seguinte comando “**mkdir routes**”.

```
D:\Angular\RetornoMetodos\Node\app>mkdir routes
```

Após o procedimento de criação das pastas para o projeto, volte para a pasta node para que possamos instalar diretórios e bibliotecas node.js, em uma pasta separada. Para este procedimento é necessário o comando “**npm install express body-parser morgan uuid jsonwebtoken helmet config --save**”, para a criação da pasta.

```
D:\Angular\RetornoMetodos\Node\app>npm install express body-parser morgan uuid jsonwebtoken helmet config --save
npm WARN Node@1.0.0 No description
npm WARN Node@1.0.0 No repository field.
+ helmet@3.13.0
+ config@2.0.1
+ uuid@3.3.2
+ body-parser@1.18.3
+ express@4.16.3
+ jsonwebtoken@8.3.0
+ morgan@1.9.0
added 96 packages from 73 contributors and audited 196 packages in 23.939s
found 0 vulnerabilities
```

2. Menu

2.1. Bibliotecas usadas

Sever.js

```
var express = require("express");
var bodyParser = require("body-parser");
var helmet = require("helmet");
var morgan = require("morgan");
var cors = require('cors')
var rotas = require("../app/routes/rotas");

var app = express();
app.use(cors())
app.use(morgan("dev"));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(helmet.noCache());
app.use(helmet.noSniff());
app.use(helmet.xssFilter());
app.use(helmet.frameguard());
app.use(helmet.frameguard());
app.use(helmet.hidePoweredBy());

app.use("/node/ProxyBitpagg/", rotas);
app.listen(process.env.PORT || 3000);
```

```
JS rotas.js x
1 // var poc = require('../controllers/poc.controller');
2 var express = require('express');
3 var auth = require('../metodos/auth')
4 var autenticador = require('../controllers/autenticador')
5 var processo = require('../controllers/processo.js')
6
7 var router = express.Router();
8
```

```
JS auth.js x
1 var jwt = require('jsonwebtoken');
2 var config = require("../config/config.json");
3
```



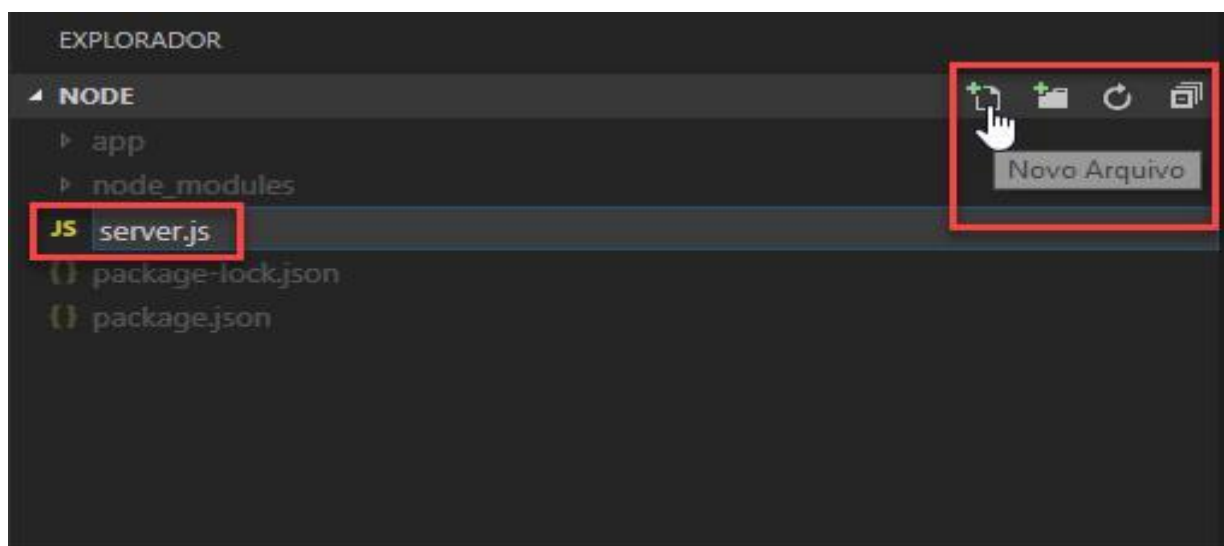
```
bitpagg_token.js x
1 var request = require("request-promise");
2 var config = require("../config/config.json");
3
```

```
autenticador.js x
1 var uuidv1 = require('uuid/v1');
2 var uuidv4 = require('uuid/v4');
3 var config = require("../config/config.json");
4 var jwt = require('jsonwebtoken');
```

```
processo.js x
1 var request = require("request-promise");
2 var config = require("../config/config.json");
3 var bitpagg = require("../metodos/bitpagg_token");
```

2.2. Criação arquivo Sever.js

Para a criação do arquivo **server.js**, vamos apenas em criar um novo arquivo na tela inicial do visual code, iremos criar dentro de nossa estrutura principal.



Após a criação do arquivo **server.js**, iremos configurar nosso servidor declarando algumas variáveis como mostra na imagem abaixo dentro do quadrado vermelho, estas variáveis serão usadas para a chamada de alguns arquivos e biblioteca que iremos utilizar, como o arquivo de rotas que iremos utilizar para navegar em nossa API.

No quadrado amarelo estão alguns parâmetros que iremos utilizar para nossa aplicação, por exemplo na linha 13, e o parâmetro que é usado para que seja armazenado nada em cache, pois, para pegar essas informações dentro do cache não é nada complicado, e alguns outros comandos de ocultação de informação e de chamada de informação por arquivo json.

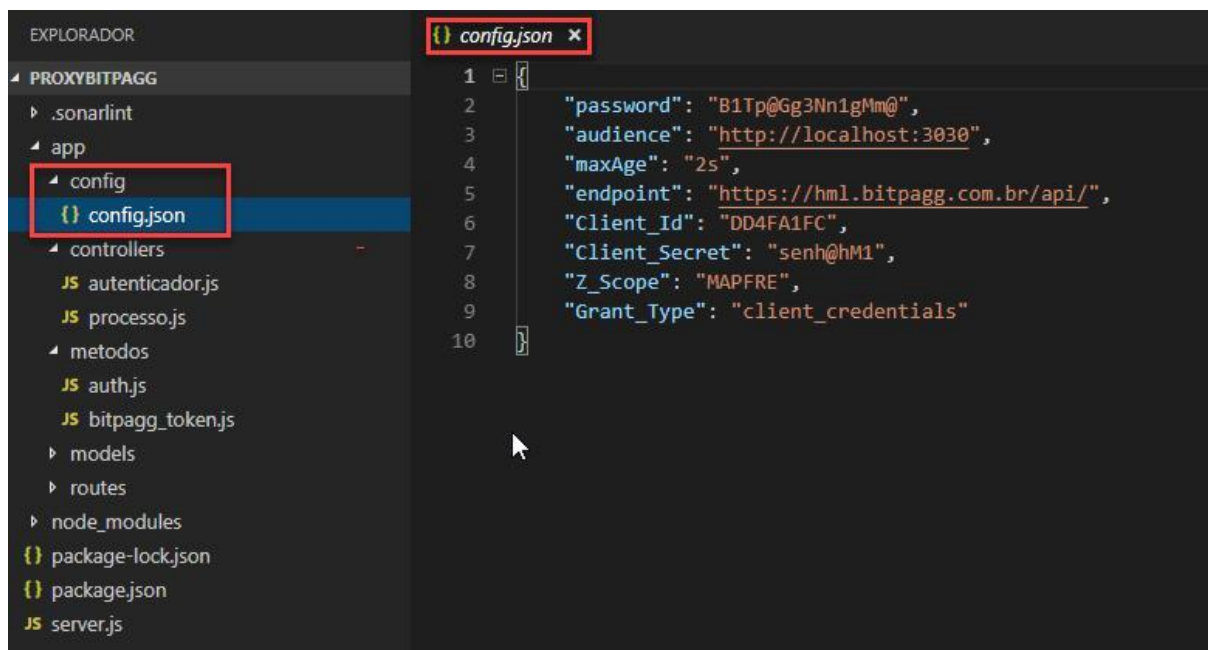
No quadrado verde mostra a parametrização que iremos utilizar para o acesso de nosso servidor, como setar a porta como padrão 3000, e declarar a rota de utilização de nosso servidor.

```
JS server.js x
1  var express = require('express');
2  var bodyParser = require('body-parser');
3  var helmet = require('helmet');
4  var morgan = require('morgan');
5  var rotas = require('./app/routes/rotas');
6
7
8  var app = express();
9
10 app.use(morgan('dev'));
11 app.use(bodyParser.urlencoded({ extended: false }));
12 app.use(bodyParser.json());
13 app.use(helmet.noCache());
14 app.use(helmet.noSniff());
15 app.use(helmet.xssFilter());
16 app.use(helmet.frameguard());
17 app.use(helmet.frameguard());
18 app.use(helmet.hidePoweredBy());
19
20
21 app.use('/node/ProxyBitpagg/', rotas);
22
23
24 app.listen(process.env.PORT || 3000);
25
26
27
```

2.3. Config

Nesta pasta será criado um arquivo “**config.json**”, que conterà todos os dados sobre o projeto, como password, endpoint, tempo de requisição e outros como

mostra a imagem abaixo. Estas informações serão usadas para a autenticação do token que será gerado para o acesso as informações via requisição http.



2.4. Routes

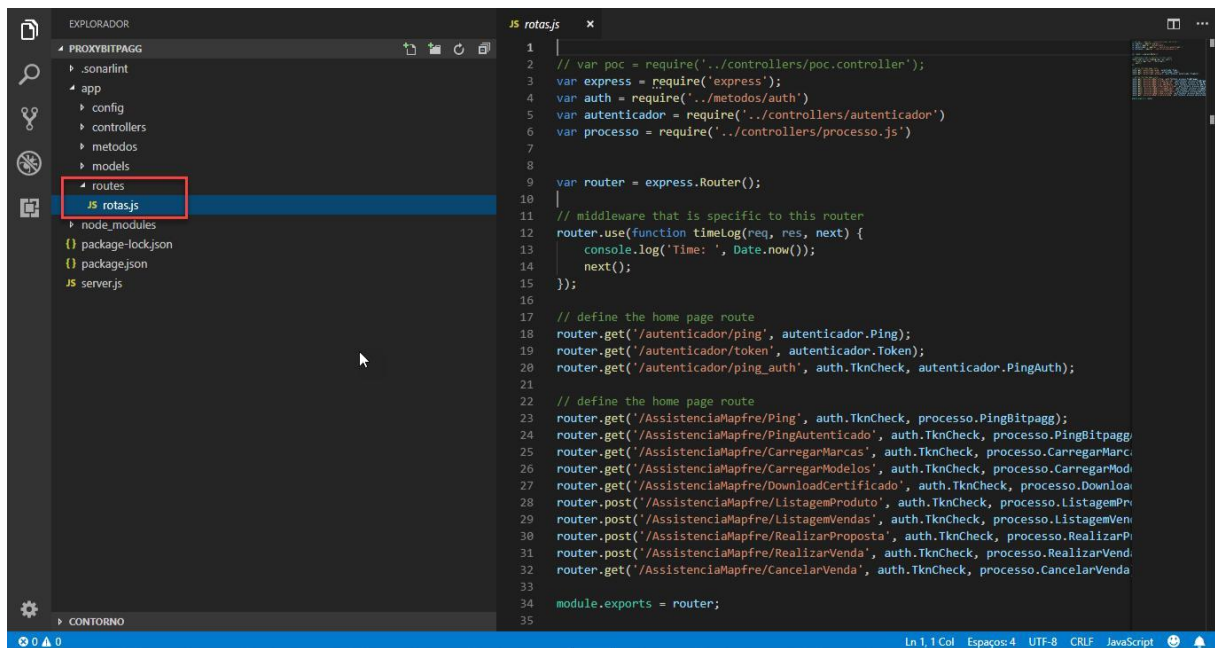
Dentro da pasta routes será criado um arquivo com o nome de **“rotas.js”**, neste arquivo conterà todas as rotas necessárias para a chamada dos métodos, no início do arquivo vemos algumas variáveis declaradas para que possamos acessar determinadas pasta como a controllers para que possamos utilizar dos métodos que estão dentro das mesmas.

Em seguida no código vemos uma função de tempo para que possamos acompanhar via console o tempo gasto para efetuar uma operação http.

Após esta função estão todos as chamadas https, de cada retorno específico da API, seja GET, POST, PUT e DELETE.

Nesta API como utilizaremos a mesma apenas para adquirir informação, não usaremos o DELETE, pode-se ver o início das chamadas sempre deve-se efetuar um GET de **“ping”**, para verificar a comunicação entre ambas as partes, se retornado **“ping”**, ok podemos prosseguir com as outras chamadas.

Nas demais chamadas estão o token que iremos implementa-lo na controller, e os outros métodos que iremos implementar na pasta models.



Na linha de chamada de uma função http, tem-se o tipo da chamada se GET ou POST, em seguida a rota onde está declarado o método de chamada, em azul está o método de autenticação e o processo de chamada de cada método.

2.5. Métodos

A pasta métodos terá a composição de dois arquivos o auth.js e o bitpagg_token.js.

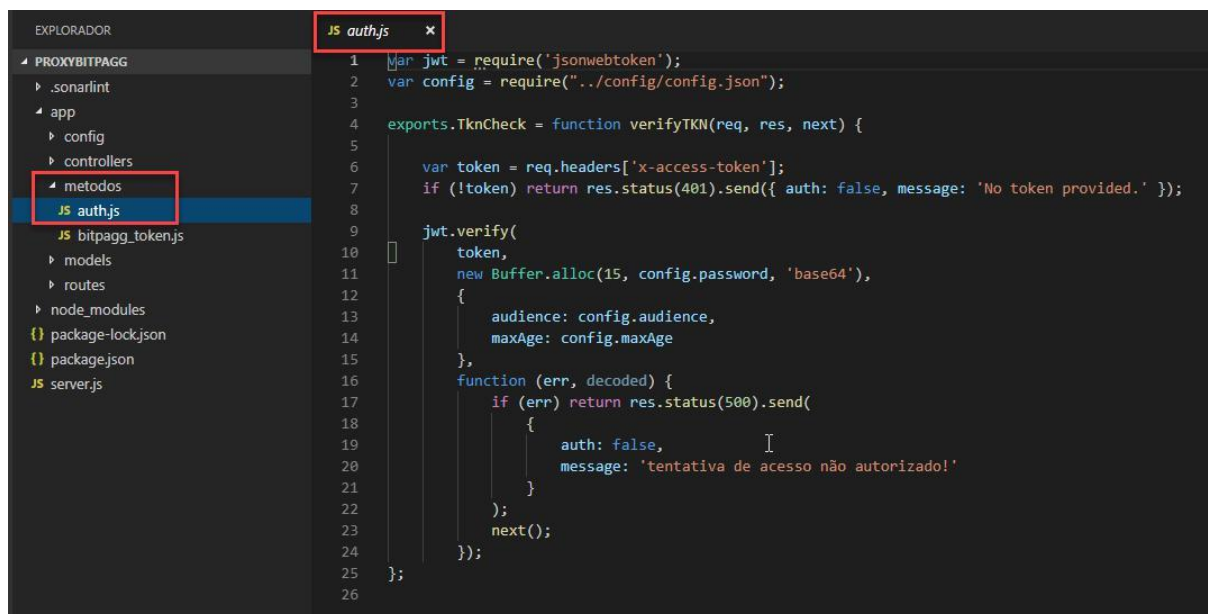
2.5.1. Auth.js

Este arquivo contém a parte de autenticação do token, para a autenticação e necessário primeiro importar a biblioteca “**jwt**”, para a utilizarmos na validação do token, em seguida importar a “**config.json**”, que contém os acessos que teremos para a autenticação do token.

A partir da linha 4 teremos a função para efetuar a autenticação, na linha 4 declaramos a função para que possamos exporta-la e usa-la em outro lugar com o nome de “**TknCheck**”, em seguida na linha 6 temos a variável “**token**” que receberá o “**x-access-token**” onde contém o corpo do token com suas informações, em seguida há um “**if**” para a verificação do conteúdo do token, pois se for retornado um erro a autenticação já para por aqui mesmo e retorna uma mensagem de erro “**No token provided**”.

Passados estes primeiros passos para a autenticação, iremos pegar a variável **“token”** onde está o corpo do token de entrada, e iremos aloca-lo em **“base64”** e confirmar os dados que estão no conteúdo do token.

Em seguida criaremos uma função de erro caso haja alguma informação entro do token que esteja inválida, onde será retornado para o usuário a mensagem, **“tentativa de acesso não autorizado”**.



```
1 var jwt = require('jsonwebtoken');
2 var config = require("../config/config.json");
3
4 exports.TknCheck = function verifyTKN(req, res, next) {
5
6     var token = req.headers['x-access-token'];
7     if (!token) return res.status(401).send({ auth: false, message: 'No token provided.' });
8
9     jwt.verify(
10         token,
11         new Buffer.alloc(15, config.password, 'base64'),
12         {
13             audience: config.audience,
14             maxAge: config.maxAge
15         },
16         function (err, decoded) {
17             if (err) return res.status(500).send(
18                 {
19                     auth: false,
20                     message: 'tentativa de acesso não autorizado!'
21                 }
22             );
23             next();
24         });
25 };
26
```

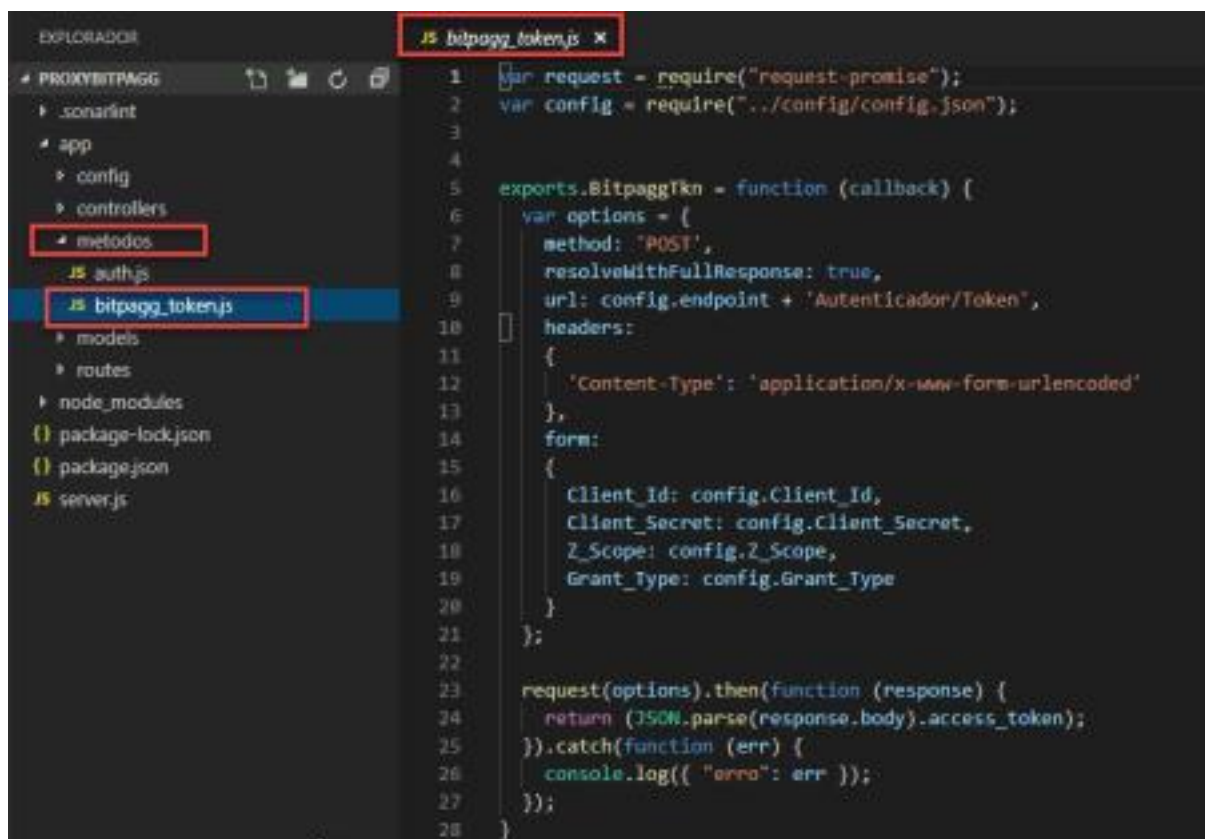
2.5.2. Bitpagg_token.js

Este arquivo contém a parte de criação do **“bitpagg_token”**, para a autenticação e necessário primeiro importar a biblioteca **“jwt”**, para a utilizarmos na validação do token, em seguida importar a **“config.json”**, que contém os acessos que teremos para a autenticação do token.

Após efetuar as chamadas que iremos precisar, criamos uma função com o nome de **“BitpaggTkn”**, que usaremos em outros arquivos, para criarmos o token devemos primeiro declarar o método http que utilizaremos que será o **“POST”** em seguida setar como **“true”** o **“resolveWhithFullResponse”**, em seguida declaramos a url por onde será chamado o token, onde a primeira parte está o endpoint padrão que está do arquivo **“config.json”** concatenado com a rota que será buscado.

No **“form”** estará as informações do cliente para a requisição de determinado método, após feito a parametrização necessária para o conteúdo do token ele irá

retornar um arquivo json com o token, caso aconteça algum erro ele irá escrever uma mensagem de erro no console especificando o tipo do erro.



```
1 var request = require("request-promise");
2 var config = require("../config/config.json");
3
4
5 exports.BitpaggTkn = function (callback) {
6   var options = {
7     method: 'POST',
8     resolveWithFullResponse: true,
9     url: config.endpoint + 'Autenticador/Token',
10    headers:
11      {
12        'Content-Type': 'application/x-www-form-urlencoded'
13      },
14    form:
15      {
16        Client_Id: config.Client_Id,
17        Client_Secret: config.Client_Secret,
18        Z_Scope: config.Z_Scope,
19        Grant_Type: config.Grant_Type
20      }
21  };
22
23  request(options).then(function (response) {
24    return (JSON.parse(response.body).access_token);
25  }).catch(function (err) {
26    console.log({ "erro": err });
27  });
28 }
```

2.6. Controllers

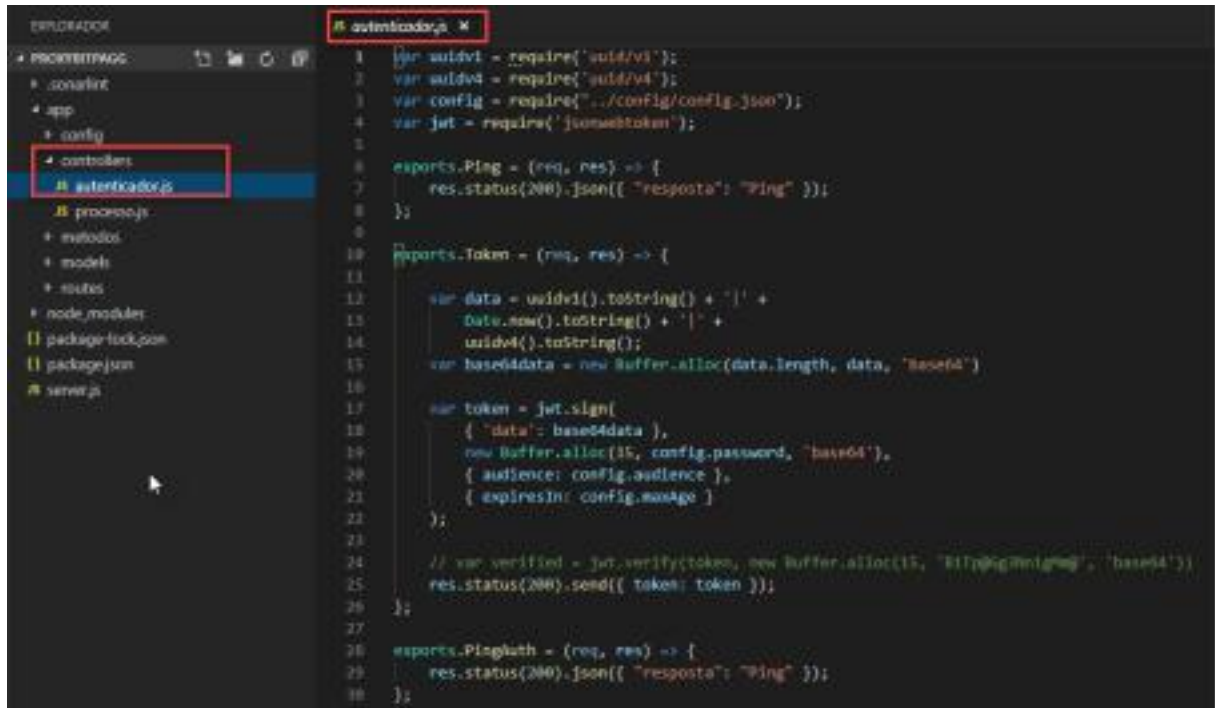
2.6.1. Autenticador.js

Neste arquivo inicialmente declaramos as bibliotecas “**uuidv1** e **uuidv4**”, que serão utilizadas para gerar parâmetros aleatórios, a rota de onde está o arquivo “**config.json**” para utilizarmos parâmetros de acesso, e a biblioteca “**jwt**” para auxiliar na autenticação.

Na primeira função iremos requerer um “**ping**”, para confirmar a comunicação.

Em seguida para autenticação do token, iremos gerar três parâmetros aleatórios e alocá-los em “**base64**” com o nome de data, para que possamos concatena-los com outros parâmetros dentro da variável “**token**”, estes outros parâmetros serão pegos no arquivo “**config.json**” para que seja padrão de ambos os lados, com a concatenação dessas variáveis criamos um json que é utilizado para a autenticação dos tokens gerados.

Em seguida feito um “**ping**” autenticado que utiliza da função anterior para efetuar o teste de autenticação.



```
1 var uuidv1 = require('uuid/v1');
2 var uuidv4 = require('uuid/v4');
3 var config = require('../config/config.json');
4 var jwt = require('jsonwebtoken');
5
6 exports.Ping = (req, res) => {
7   res.status(200).json({ "resposta": "Ping" });
8 };
9
10 exports.Token = (req, res) => {
11
12   var data = uuidv1().toString() + '|' +
13     Date.now().toString() + '|' +
14     uuidv4().toString();
15   var base64data = new Buffer.alloc(data.length, data, 'base64');
16
17   var token = jwt.sign(
18     { 'data': base64data },
19     new Buffer.alloc(15, config.password, 'base64'),
20     { audience: config.audience },
21     { expiresIn: config.maxAge }
22   );
23
24   // var verified = jwt.verify(token, new Buffer.alloc(15, 'Bitpagg@innnng', 'base64'));
25   res.status(200).send({ token: token });
26 };
27
28 exports.Pingauth = (req, res) => {
29   res.status(200).json({ "resposta": "Ping" });
30 };
```

2.6.2. Processo.js

No arquivo de processo contém todas as nossas chamadas de métodos http, para as informações necessárias para o frontend.

As chamadas https que utilizaremos serão o GET, POST e PUT, será explicado uma chamada de cada pois a chamada em si serão todas iguais apenas mudarão os métodos que serão chamados e as variáveis que serão usadas como parâmetro para o retorno adequado.

No arquivo para efetuar as chamadas dos métodos e necessário efetuar algumas declarações para conseguirmos adquirir parâmetros necessários para as requisições http. Declarando a biblioteca para auxílio na requisição, e rotas dos arquivos internos para que possamos utilizar algumas variáveis necessária, como o “**BitpaggTkn**” e o arquivo “**config**” onde fica as especificações do nosso acesso.


```

JS processo.js x
1  var request = require("request-promise");
2  var config = require("../config/config.json");
3  var bitpagg = require("../metodos/bitpagg_token")
4

```

Vamos começar pelo GET, para exemplificar iremos utilizar o método de Download certificado, primeiro criamos a função com o nome do método que utilizaremos, após criada a função dentro da mesma iremos requisitar o “BitpaggTkn”, este token é usado para a autenticação da requisição do método, em seguida dentro de “options” será declarado o método GET, a url de requisição do serviço onde a primeira parte da url é composta pelo endpoint que está dentro do arquivo “confi.js” concatenado com a rota do serviço que neste exemplo será “AssistenciaMapfre/DownloadCertificado”, em seguida no “qs” e passado os parâmetros necessários para a filtragem e especificação da informação na qual foi requerido.

Em seguida em “headers” será passado a string “bearer” concatenada com o token e outra string especificando o tipo de arquivo que no caso será tipo json.

Em seguida as declarações de parâmetros está o tratamento de erro, onde caso seja retornado à informação sem nenhum erro a mesma será exibida em “body” no formato json, usado para comparação o status “200”, caso contrário será apresentado o status de erro no “body”, usado para comparação o status “500”.

```

exports.DownloadCertificado = function DownloadCertificado(req, res) {
  let initializePromise = BitpaggTkn();
  initializePromise.then(function (result) {
    var options = {
      method: 'GET',
      url: config.endpoint + 'AssistenciaMapfre/DownloadCertificado',
      qs: { OCP_IdAlternativo: req.query.OCP_IdAlternativo },
      headers: {
        'Authorization': 'bearer '.concat(result),
        'Content-Type': 'application/json'
      }
    };
    return request(options, function (error, response, body) {
      if (error) throw new Error(error);
      res.status(200).json(JSON.parse(body));
    }); // console.log(result)
  }, function (err) {
    res.status(500).json(JSON.parse(body));
  });
}

```


Agora vamos usar o POST, para exemplificar iremos utilizar o método de Listagem de produtos, primeiro criamos a função com o nome do método que utilizaremos, após criada a função dentro da mesma iremos requisitar o “**BitpaggTkn**”, este token é usado para a autenticação da requisição do método, em seguida dentro de “**options**” será declarado o método POST, a url de requisição do serviço onde a primeira parte da url é composta pelo endpoint que está dentro do arquivo “**confi.js**” concatenado com a rota do serviço que neste exemplo será “**AssistenciaMapfre/ListagemProduto**”.

Em seguida em “**headers**” será passado a string “**bearer**” concatenada com o token e outra string especificando o tipo de arquivo que no caso será tipo json.

Após a “**headers**” será especificado onde estão os parâmetros necessários para o retorno da informação necessária e especificando que os parâmetros fornecidos estarão em formato json.

Em seguida as declarações de parâmetros está o tratamento de erro, onde caso seja retornado à informação sem nenhum erro a mesma será exibida em “**body**” no formato json, usado para comparação o status “**200**”, caso contrário será apresentado o status de erro no “**body**”, usado para comparação o status “**500**”.

```
exports.ListagemProduto = function ListagemProduto(req, res) {
  let initializePromise = BitpaggTkn();
  initializePromise.then(function (result) {
    var options = {
      method: 'POST',
      url: config.endpoint + 'AssistenciaMapfre/ListagemProduto',
      headers: {
        'Authorization': 'bearer '.concat(result),
        'Content-Type': 'application/json'
      },
      body: req.body,
      json: true
    };
    return request(options, function (error, response, body) {
      if (error) throw new Error(error);
      res.status(200).json(JSON.parse(body));
    }); // console.log(result)
  }, function (err) {
    res.status(500).json(JSON.parse(body));
  });
});
```

A utilização do PUT, para exemplificar iremos utilizar o método de Cancelamento de venda, primeiro criamos a função com o nome do método que utilizaremos, após criada a função dentro da mesma iremos requisitar o “**BitpaggTkn**”, este token é usado para a autenticação da requisição do método, em seguida dentro de “**options**” será declarado o método PUT, na url de requisição do serviço onde a primeira parte é composta pelo endpoint que está dentro do arquivo “**config.js**” concatenado com a rota do serviço que neste exemplo será “**AssistenciaMapfre/CancelarVenda**”, em seguida no “**qs**” é passado os parâmetros necessários para concretizar o método.

Em seguida em “**headers**” será passado a string “**bearer**” concatenada com o token e outra string especificando o tipo de arquivo que no caso será tipo json.

Em seguida as declarações de parâmetros está o tratamento de erro, onde caso seja retornado à informação sem nenhum erro a mesma será exibida em “**body**” no formato json, usado para comparação o status “**200**”, caso contrário será apresentado o status de erro no “**body**”, usado para comparação o status “**500**”.

```
exports.CancelarVenda = function CancelarVenda(req, res) {
  console.log(req.query.OCP_IdAlternativo);
  let initializePromise = BitpaggTkn();
  initializePromise.then(function (result) {
    var options = {
      method: 'PUT',
      url: config.endpoint + 'AssistenciaMapfre/CancelarVenda/',
      qs: { OCP_IdAlternativo: req.query.OCP_IdAlternativo },
      headers: {
        'Authorization': 'bearer '.concat(result),
        'Content-Type': 'application/json'
      }
    };
    return request(options, function (error, response, body) {
      if (error) throw new Error(error);
      res.status(200).json(body);
    }); // console.log(result)
  }, function (err) {
    res.status(500).json(JSON.parse(body));
  });
}
```

3. Tratamento de erros

Para o tratamento de erros nos métodos criados dentro de "processo.js", deve-se fazer da seguinte maneira:

```
return request(options, function(error, response, body) {  
  res.status(response.statusCode).json(body)  
});  
},  
function(err) {  
  res.status(500).json(body);  
})
```

Efetuar o retorno da proxy com os status que a API Bitpagg retornar, o conteúdo deve ser passado de forma direta como JSON.

E na função erros retornar o status 500 e corpo da mensagem deve ser passado também em JSON.

Caso não esteja retornando um JSON da API Bitpagg deve-se verificar com o desenvolvedor da mesma para retornar a resposta em formato JSON.

4. Criação Web.config

A criar a aplicação para deploy em ambiente de homologação deve-se criar o arquivo com o nome web.config, neste arquivo deve conter o seguinte código:

```
<configuration>  
  <system.webServer>  
    <httpErrors existingResponse="PassThrough"  
  /> <handlers>  
    <add name="iisnode" path="server.js" verb="*" modules="iisnode" />  
  </handlers>  
  <rewrite>  
    <rules>  
      <rule name="(NOME DA APLICAÇÃO)">  
        <match url="/*" />  
        <action type="Rewrite" url="server.js" />  
      </rule>
```

```
</rules>
</rewrite>
</system.webServer>
</configuration>
```

Neste código deve ser feita a seguinte alteração, na tag “rule” deve-se especificar o nome da aplicação em que está sendo criada. exemplo: <rule name=**"easytaxi"**>

A tag **<httpErrors existingResponse="PassThrough" />** terá que ser implementa para evitar problemas, como foi observado no retorno da mensagem **“Bad request”**, esta tag tem a função de forçar o retorno do Json vindo da API Bitpagg.

5. Possível problema Proxy Node.JS

Métodos não autenticados. Para correção do método favor instanciar o tempo de vida do token em 1s, sendo indicado a utilização do mesmo com o intervalo de 0.75 segundos.

5.1. Forma incorreta

```
router.get('/AssistenciaMapfre/PingAutenticado',processo.PingBitpaggAuth);
```

5.2. Forma correta

```
router.get('/AssistenciaMapfre/PingAutenticado',auth.TknCheck,
processo.PingBitpaggAuth);
```

5.3. Dados Header

Para o envio de referer e origin é necessário realizar a seguinte alteração no header das chamadas feitas para API's, na pagina de **“processos.js”** dentro da proxy:

```
headers:
{
  Authorization: "bearer ".concat(result),
  origin: req.headers.origin,
  referer: req.headers.referer
```

6. Criação método Checkout

Este método será criado somente para o consumo de informações de pagamento relacionados a Cyber Source, onde será usado uma tela de captura desenvolvida pela BitPagg, onde como padrão adotamos passar tais informações pelo proxy/node.js.


Em relação ao proxy deve-se manter os padrões exigidos para os demais métodos, para este método específico iremos seguir do seguinte padrão:

- **app/serve.js**: `app.use("/node/" + NomeProjeto + "/", rotas);`
- **app/config/config.json**: `{"RetornoOk" : "Link Página de Sucesso", "RetornoErr" : "Link Página de Erro"}`
- **app/routes/rotas.js**: `router.post('/CyberSource/Checkout', processo.Checkout);`
- **app/controllers/processos.js**:

```
let respostaCheckout;
let resposta01;
let resposta02;
exports.Checkout = function Checkout(req, res) {
  try {
    respostaCheckout = req.body;

    if (respostaCheckout["StatusCode"] == 200 ||
        respostaCheckout["StatusCode"] == 201)
    {
      resposta01 = respostaCheckout["Content"];
      res.status(200).json(config.RetornoOk);
    }
    else {
      resposta01 = respostaCheckout["Content"];
      if(resposta01 == null){
        resposta02 = respostaCheckout["Message"];
      }

      res.status(500).json(config.RetornoErr);
    }
  }
  catch {
    res.status(500).json(config.RetornoErr);
  }
};
```



Este método base tem a função de receber um json de retorno na variável **“req”**, onde será feito a validação da descrição do status que está descrito na **“req”**, caso o **“StatusCode”** seja **“200”** ou **“201”** será retornado uma url de sucesso **“config.RetornoOK”** e status **“200”**, caso seja diferente de **“200”** ou **“201”** não existir o **“StatusDescription”** no json será retornado um url de Erro **“config.RetornoErr”** e status **“500”**, que estará localizada na config.json, caso não ocorra nenhum desses casos.

O **“req,body”** será gravado em uma variável global **“respostaCheckout”**, onde poderá ser usada da melhor forma possível, de acordo com o fluxo dos dados na aplicação.

Lembrando que este método acima é um exemplo de como será feito a validação para retorno da checkout. Caso haja necessidade podem ser feitas alterações de acordo com a necessidade do projeto, apenas deve ser fixo o retorno das url's de erro e de sucesso para que seja efetuado o redirecionamento da página.

7. Observações

Nas requisições REST feitas pela proxy para as API's bitpagg que demoram mais do que 180 segundos serão barradas pelo servidor, e retornado erro de timeout **“504”**. Pois está sendo usado por padrão este tempo para retorno da requisição, esta configuração feita pelo lado do servidor.

