

BIT PAGG LTDA.



DESENVOLVIMENTO DE PROXY EM NODE.JS

Sumário

1.	INTRODUÇÃO	2
2.	CRIAÇÃO DA APLICAÇÃO PROXY	3
3.	CONFIGURATIONS	5
3.1.	CONFIGURAÇÃO: AppFlow.json	5
3.2.	CONFIGURAÇÃO: appsettings.json	5
4.	CONTROLLERS	5
5.	METHODS	5
5.1.	<i>METHODS: Authentication.js</i>	5
5.1.1	<i>METHODS: GenerateToken</i>	6
5.1.2	<i>METHODS: CheckToken</i>	6
5.2	<i>METHODS: Cryptography.js</i>	6
5.2.1	<i>METHODS: CreateCipher</i>	7
5.2.2	<i>METHODS: CreateDecipher</i>	7
6	MODELS	7
7	ROUTES	7
8	SERVER.JS	7
9	WEB.CONFIG	8
10	CONSIDERAÇÕES GERAIS	9
1.	CONFIGURATIONS	10
a.	<i>AppFlow.json</i>	10
b.	<i>appsettings.json</i>	13
2.	CONTROLLERS	13
a.	<i>AuthenticationController.js</i>	13
b.	<i>ControllerExample</i>	15
3.	METHODS	17
a.	<i>Authentication.js</i>	17
I.	GenerateToken	17
II.	CheckToken	18
III.	validateFlow	19
b.	<i>Cryptography.js</i>	22
I.	createCipher	22
II.	createDecipher	22
4.	MODELS	23
5.	ROUTES	23
a.	<i>RoutesConfig.js</i>	23
6.	SERVER.JS	24
7.	Testes no postman	26
1.	INTRODUÇÃO	

Nesta documentação será apresentado o padrão de desenvolvimento da aplicação proxy, especificando como é estruturado o código desta aplicação assim como suas classes e funções.

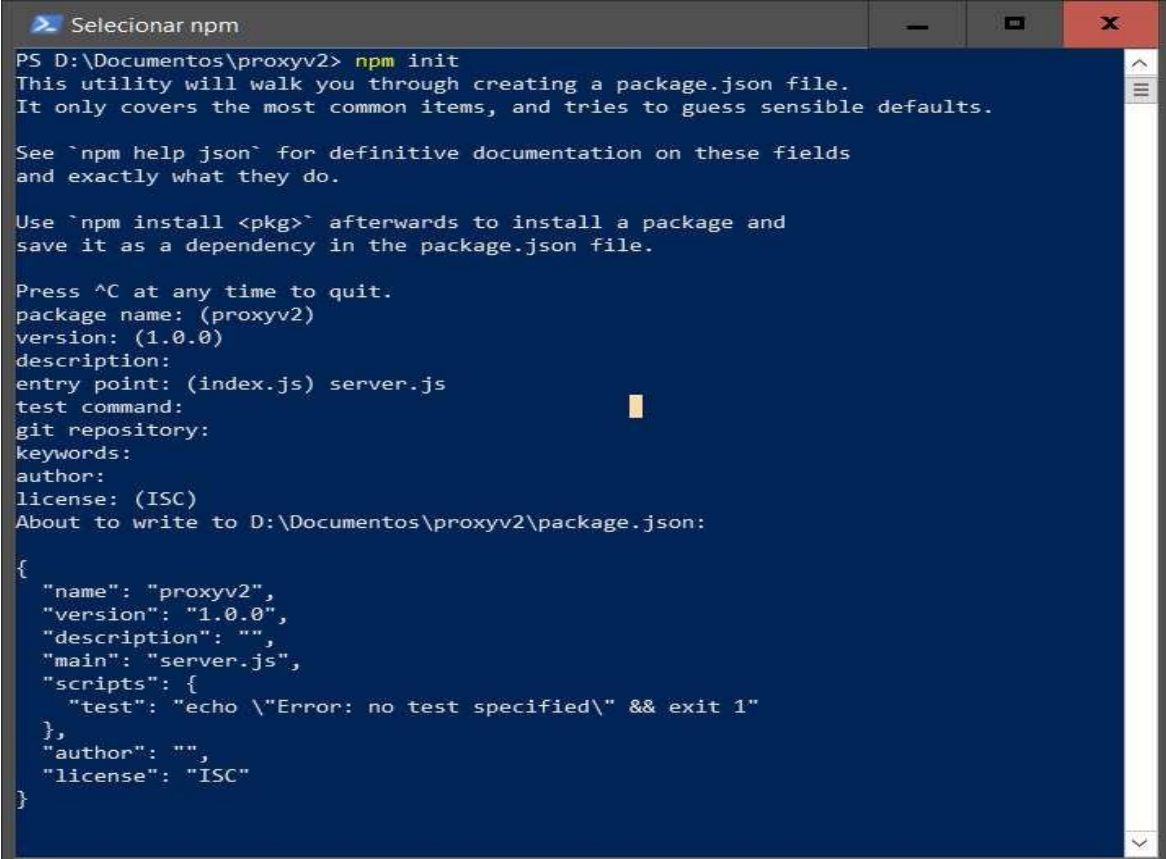
2. CRIAÇÃO DA APLICAÇÃO PROXY

A princípio crie um diretório para podermos iniciar a estruturação das pastas e instalações das bibliotecas que iremos utilizar. Feito isso, inicie o Visual Studio Code e vá ao prompt de comando, verificando se o diretório está apontando para a pasta que você criou para iniciar a aplicação.

Digite o comando `"npm install"` onde será criado o arquivo `"package-lock.json"`. Em seguida entre com o comando `"npm init"`, criando o arquivo `"package.json"`. Ao executar esta instrução serão feitas algumas perguntas referentes ao projeto como `"name"`, `"version"`, `"entry point"` (que deverá ser sempre: `"server.js"`), `"description"`, `"repositorio"`, dados com os quais o arquivo `"package.json"` será preenchido.

Logo em seguida iremos efetuar a instalação de algumas bibliotecas que não são instaladas por padrão:

`npm i express, body-parser, morgan, await, uuid, jsonwebtoken, helmet, config, crypto, url, cors, fs, fetch`



```
PS D:\Documentos\proxyv2> 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: (proxyv2)
version: (1.0.0)
description:
entry point: (index.js) server.js
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\Documentos\proxyv2\package.json:
{
  "name": "proxyv2",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Terminada a instalação, vamos criar nossa estrutura de pastas para o desenvolvimento desta aplicação. Crie uma pasta com o nome de `"app"` com o

comando “*mkdir app*”. Acesse a pasta com “*cd app*” e dentro de pasta “app” vamos criar mais cinco pastas com os nomes de “*Configurations*”, “*Controllers*”, “*Methods*”, “*Models*”, “*Routes*”, para isto, utilize o comando “*mkdir Configurations; mkdir Controller; mkdir Methods; mkdir Models; mkdir Routes*”

```
Windows PowerShell
"\"description\": \"\",
  \"main\": \"server.js\",
  \"scripts\": {
    \"test\": \"echo \\\"Error: no test specified\\\" && exit 1\"
  },
}
Is this OK? (yes)
PS D:\\Documentos\\proxyv2> mkdir app
}

Diretório: D:\\Documentos\\proxyv2
Is this OK? (yes) yes

Mode                LastWriteTime         Length Name
-----
d-----         14/04/2020    16:30                app

PS D:\\Documentos\\proxyv2> cd app
PS D:\\Documentos\\proxyv2\\app> mkdir Configurations; mkdir Controller; mkdir Methods;
mkdir Models; mkdir Routes

Diretório: D:\\Documentos\\proxyv2\\app

Mode                LastWriteTime         Length Name
-----
d-----         14/04/2020    16:30        Configurations
d-----         14/04/2020    16:30        Controller
d-----         14/04/2020    16:30        Methods
d-----         14/04/2020    16:30        Models
d-----         14/04/2020    16:30        Routes

PS D:\\Documentos\\proxyv2\\app>
```

3. CONFIGURATIONS

Nesta pasta serão criados os arquivos de configuração json, utilizados no decorrer do processo de validação do token e para utilização nas requisições para as API's bitpagg. A especificação dos nomes dos arquivos e o que estará contido dentro deles será explicado logo abaixo.

OBS: É de caráter obrigatório, que TODOS os arquivos em questão sejam criados de acordo com a especificação.

3.1. CONFIGURAÇÃO: AppFlow.json

O arquivo AppFlow será utilizado e editado para a criação de parâmetros fixos que serão utilizados na requisição para as API's bitpagg. Como padrão, neste arquivo conterão dados da aplicação, como o nome, url (endpoint) e o fluxo de chamadas de cada tela da aplicação em questão. É imprescindível a criação deste fluxo para a validação e segurança do token da aplicação que irá consumir a proxy.

3.2. CONFIGURAÇÃO: appsettings.json:

O appsettings será utilizado para criação de constantes usadas no processo de autenticação e criação do token. Este arquivo será imutável, portanto o padrão inicial estipulado não deve ser alterado, seja removendo ou acrescentando novos parâmetros relacionado ao projeto que utilizará a proxy.

4. CONTROLLERS

Nesta pasta estará contidas as controllers que efetuarão as requisições para nossas API's Bitpagg, adotado o conceito de micro serviços para as requisições criadas. Ou seja, deve ser criada mais de uma controller para a descentralização das requisições, de acordo com o fluxo e utilização das requisições.

5. METHODS

Nesta pasta estará contido as funções e classes que serão utilizadas para a geração e autenticação do token e suas devidas restrições na utilização dos métodos de acordo com o fluxo indicado no "AppFlow.json". **Estas funções não serão editáveis.** Nesta pasta existem dois arquivos, são esses:

5.1. METHODS: Authentication.js

Neste arquivo estará contido a classe "AuthenticationMethods", e dentro desta classe está contido as funções "GenerateToken" e "CheckToken".

5.1.1 METHODS: GenerateToken

Esta função é utilizada para a geração do token. Para o funcionamento desta função é necessário a passagem de dois parâmetros para a inclusão nas variáveis que compõem o token. As outras variáveis serão geradas utilizando a biblioteca “*uuid*”, onde são gerados valores aleatórios.

Ao reunir todas as variáveis de composição do token, será inicialmente criptografado utilizando a função “*createCipher*” dentro da classe “*Cryptography*”, onde eles são transformados em uma string “*aes256*” com uma chave de verificação. Em seguida inicia o próximo processo de encriptação convertendo esta string em um “*base64*”. Em seguida são criadas mais cinco variáveis que estão contidas nos arquivos json na pasta “*Configurations*”. Estas variáveis serão utilizadas com chaves para a geração do JWT. Ao gerar o JWT e retornado o mesmo para a origem da requisição.

5.1.2 METHODS: CheckToken

Esta função será utilizada para a autenticação do token. Inicialmente é pego o token na requisição passada, na header da requisição é extraído o token que está no atributo “*x-access-token*”, em seguida é realizado o processo inverso de geração do token, primeiramente é efetuado a verificação do JWT com suas chaves localizadas nos arquivos JSON da aplicação, ao termino da verificação é efetuada a decodificação da string “*base64*” e colocada no padrão “*ascii*”, feito a decodificação será chamada à função “*createDecipher*”, que efetua a decriptação da string no padrão “*aes256*” com sua referente chave para verificação do token.

Em seguida são feitas as verificações das chaves de segurança que foram recebidas pela requisição ao gerar o token e em outra requisição qualquer. É passado o token criado com as informações da aplicação, e efetuado o filtro do objeto para validação de permissões desta chamada, verificando se o método e a url estão liberadas para efetuar tal requisição. Caso nenhuma destas validações estejam OK será retornado a mensagem “*Unauthorized access attempt!*”, caso esteja tudo OK o fluxo do método continua e é efetuada a requisição destinada.

5.2 METHODS: Cryptography.js

Este arquivo é criado para conter a classe “*Cryptography*”, onde estão localizadas as funções para encriptação e decriptação do token.

5.2.1 METHODS: CreateCipher

Nesta função é feita a encriptação da string recebida. Primeiramente é efetuado a criptografia da string no padrão “aes256” com uma chave de segurança, em seguida é feita uma transformação da string de “utf8” para “hex”. Feito isso a string é retornada.

5.2.2 METHODS: CreateDecipher

Nesta função é feita a deciptação da string recebida. Primeiramente é feita deciptografia utilizando o padrão “aes256” e a chave de segurança para a validação da string e em seguida esta string é transformada de “hex” para “utf8” e retornada.

6 MODELS

Esta pasta será utilizada para a criação de objetos de requisição, que serão utilizados nas controllers para comunicação com as API’s bitpagg. Sendo criadas em padrão POO, sendo assim criar classes diferentes para cada objeto diferente.

7 ROUTES

Nesta pasta estará contido o arquivo “*RoutesConfig.js*”, sendo responsável pela criação das rotas.

Neste arquivo, serão criado as rotas dos métodos utilizados pelo cliente. Em cada rota criada deve-se ser feita a autenticação, seja ela em qualquer tipo de chamada: GET, POST, PUT, entre outros. A pasta “Controller” será como referência para o direcionamento dos métodos, contendo todas as requisições para as API’s bitpagg. Para efetuar a validação do token que está sendo enviado, deve ser utilizado a função “*CheckToken*” na classe “*AuthenticationMethods*”.

8 SERVER.JS

Este arquivo deve ser criado na raiz do projeto, fora da pasta “app”. Este arquivo contém parâmetros que serão utilizados pelo servidor onde estará localizada a aplicação. Neste arquivo serão utilizadas algumas bibliotecas. Segue bibliotecas e funcionalidade delas:

- ❑ express: Criação de um servidor.
- ❑ cors: Libera portas em ambiente DSV para testes.
- ❑ helmet.frameguard: Utilizado para negar ou permitir o enquadramento de sua aplicação.

- ❑ `helmet.hsts`: Informa aos navegadores que aderem ao HTTPS e nunca visitam a versão HTTP insegura.
- ❑ `helmet.noSniff`: Impede que o navegador realize um sniff nos MIME TYPES.
- ❑ `helmet.noCache`: Previne gravação de cache pelo navegador.
- ❑ `helmet.xssFilter`: Previne cross site scripting.
- ❑ `helmet.hidePoweredBy`: Esconde informações referentes a tecnologia do sistema.
- ❑ `Morgan`: Aplica configurações para dentro do servidor express, adicionando middlewares (`body-parser`, `morgan`, `cors`).
- ❑ `bodyParse.urlencoded`: Permite escolher entre analisar os dados codificados em URL com a query string biblioteca (quando `false`) ou a qs biblioteca (quando `true`).
- ❑ `bodyParser.json`: Verifica os objetos enviados pelo corpo da requisição.
- ❑ `helmet.dnsPrefetchControl`: Esse middleware permite desativar a pré-busca de DNS dos navegadores, definindo o X-DNS-Prefetch-Control cabeçalho.

9 WEB.CONFIG

Para deploy em ambiente de homologação deve-se criar o arquivo com o nome “*web.config*” na raiz do projeto. 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="proxyv2">
          <match url="/*" />
          <action type="Rewrite" url="server.js" />
        </rule>
      </rules>
    </rewrite>
```



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

Neste código deve-se especificar o nome da aplicação em que está sendo criada. Exemplo: `<rule name="easytaxi">` .

A tag `<httpErrors existingResponse="PassThrough" />` terá que ser implementada 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.

10 CONSIDERAÇÕES GERAIS

- ☐ Nomenclatura de classes e funções deverá seguir o padrão CamelCase.
- ☐ Não será admitido o uso de `"console.log()"` dentro da aplicação em ambientes de homologação e/ou superiores.
- ☐ Não será admitido o uso de cors em ambientes superiores.
- ☐ As funções deverão ser autenticadas com o padrão apresentado, não sendo admitidos desvios desse padrão por **NENHUM** motivo.
- ☐ No arquivo `"appsettings.json"` a equipe de desenvolvimento poderá alterar APENAS as urls do host, subject e audience.
- ☐ Não deverá, em hipótese alguma, existir algum tipo de parâmetro não estabelecido pela equipe de desenvolvimento do front, dentro do arquivo `"appsettings.json"`.
- ☐ É estritamente proibido que alguma rota que consuma qualquer uma das api's Bitpagg, esteja sem autenticação por qualquer motivo que seja.
- ☐ É obrigatório que o fluxo da aplicação descrito em appflow, venha a refletir exatamente o fluxo de utilização das apis.
- ☐ É obrigatório que o desenvolvedor verifique se o arquivo web.config existe na aplicação e que está como especificado nessa documentação.
- ☐ **Aplicações que não estejam no padrão descrito neste documento não terão garantia de suporte.**

Metodologia de configuração e construção da PROXY – Guiado usuário iniciante

1. CONFIGURATIONS

Nos item a seguir será explicado a parametrização dos arquivos e no fim terá um exemplo de uso QUE DEVERÁ SER EDITADO PARA UTILIZAÇÃO.

a. AppFlow.json

O arquivo começa com o paramentro “grantAuth” que define o tipo de autenticação.

"grantAuth": "password"

Em seguida temos “grantCode” que recebe o tipo de código para autenticação.

"grantCode": "authorization_code"

grantType que é o tipo de credencial (token) que vai ser recebido.

"grantType": "client_credentials"

grantTypePassword é a forma de obtenção para autenticação.

"grantTypePassword": "password"

zCompany é utilizado para dar nome para o projeto dedicado. No exemplo será utilizado o MAPF.

"zCompany": "MAPF"

O apiEndPoint é onde a aplicação vai buscar as funções a serem utilizadas.

Trocar sempre que for testar outro endpoint.

"apiEndPoint": "https://hml.bitpagg.com.br/api/"

Audience é o end point da aplicação no caso

"audience": https://hml.bitpagg.com.br/api/

O zScope, clientId, clientSecret, campanha, intGrupo e CodigoModulo é definido de acordo com o projeto e é descrito no arquivo de configuração do projeto

**"zScope": "MAPFRE",
"clientId": "7E11F293",**

```
"clientSecret": "senh@hM1", "campanha": 102,  
"intGrupo": 49,  
"CodigoModulo": 10,
```

O appFlowClient abre chaves para receber um conjunto de parametros que serão descritos a seguir.

“application”: “Nome_da_aplicação”

O nome da aplicação deve ser inserido nessa propriedade. Exemplo:
"application": "proxyv2"

“applicationUrl”: “http://localhost:4200”

A url da aplicação deve ser colocada nessa propriedade. Olocalhost na porta 4200 é utilizado como padrão para o ambiente dev, este url deve ser configurado novamente ao levar a aplicação para homologação ou produção.

Monta-se então o fluxo de aplicação por meio da propriedade appFlow.

Abaixo se encontra um exemplo do código com comentários em cada ramo:

```
“appFlow”: [{
```

```
“step”: 0,
```

No campo “step”, se adiciona o número do passo que se encontra como um incremento. O número de step aumenta de acordo com o número de rotas da sua aplicação contando com a home “/”. Por exemplo {url_da_aplicação}/step0/step1/step2 e ai por diante.

```
“stepName”: “Home”
```

Neste campo deve-se colocar um nome no qual representa o destino de sua rota e sua função. Por exemplo, para a rota raiz, “/” será a home de nosso projeto.

```
“stepPath”: “/”,
```

Neste campo defina a rota adotada na aplicação, porém com uma barra anterior, conforme a configuração definida em app-routing.module.ts na aplicação. Por exemplo, o roteamento para a página “step1” deverá ficar como “/step1” no campo “stepPath”.

A propriedade “allowFunctions” irá permitir os métodos de chamada API para a sua página especifica definida em “stepPath”.

“allowFunctions”:

[{ “method”: “GET”

Defina o método de chamada API neste campo. Esse método é referente ao método dentro da aplicação. Se for uma chamada de get na aplicação, aqui também deverá ser get. Caso não seja igual, a proxy barra e não deixa executar a solicitação “resource”:

“/ + url da chamada”

Neste campo contém o “/” concatenado com a url da chamada de API contida nos métodos dentro de app.service.ts na aplicação. Por exemplo, a rota adotada na aplicação para o token de autenticação “Authentication/Token” passa a ser “/Authentication/Token”.

Obs: se a mesma página constituir de diversos métodos, os mesmos deverão ser adicionados abrindo uma nova {} e adicionando “method” e “resource” específicos para a página. Assim para cada página adicionada e a cada novo roteamento deverá ser feita a configuração nos moldes acima, sempre incrementando o “step”.

Modelo de um appflow.json

```
{
  "zScope": "MAPFRE",
  "clientId": "7E11F293",
  "clientSecret": "senh@hM1",
  "grantAuth": "@cLboSc#",
  "grantCode":
    "authorization_code",
  "grantType": "client_credentials",
  "grantTypePassword": "password",
  "nomeAplicacao": "Aplicação de exemplo",
  "audience": "https://hml.bitpagg.com.br/api/",
  "apiEndPoint": "https://hml.bitpagg.com.br/api/",
  "campanha": 102,
  "intGrupo": 49,
  "CodigoModulo": 10,
  "appFlowClient": {
    "application":
      "proxyv2",
    "applicationUrl": "http://localhost:4200",
    "appFlow": [
      {
        "step": 0,
        "stepName":
          "Home",
        "stepPath": "/", "allowFunctions":
          [
            {
              "method": "GET",
              "resource": "/Authentication/Ping"
            },
            {
              "method": "GET",
              "resource": "/Authentication/Token"
            }
          ]
      }
    ]
  }
}
```

```

    },
    {
      "method": "GET",
      "resource": "/Authentication/PingAuth"
    }
  ]
}
}
}
}
}

```

b. **appsettings.json**

O arquivo appsettings.json, contém configurações da aplicação para com o proxy, neste não deverá ser adicionado nenhum novo parâmetro, o nome da aplicação deve ser adicionado ao “nodeAppPath” e para o host utiliza-se o localhost:3000 para dev, o mesmo parâmetro deve ser modificado em “subject” e “audience”.

Parâmetros como cryptographyAlgorithm, cryptographyPassword, JwtPassword, issuer, expiresIn não devem ser alterados de forma alguma, a alteração nesses parâmetros irá quebrar a autenticação do proxy, assim como não deve ser adicionado nenhum parâmetro além dos já existentes.

Veja o código abaixo:

```

{
  "nodeAppPath": "/node/MapfreResidencialServer",
  "nodeAppPort": 3000,
  "cryptographyAlgorithm": "aes256",
  "cryptographyPassword": "@XTu49cT@qjhq7Hbxj",
  "host": "http://localhost:3000",
  "JwtPassword": "JwtPassword",
  "issuer": "Bitpagg",
  "expiresIn": "60000s",
  "subject": "http://localhost:3000",
  "audience": "http://localhost:3000",
  "password": "B1tP4G6Pr0xY"
}

```

2. CONTROLLERS

a. Authenticator.js

O controller de autenticação será padrão e imutável. Este controller coordena a criação do token para as chamadas de API. O controlador de autenticação possui dois métodos, o de ping, para testes de resposta, e o de token, para a criação de token.

O método de Ping possui request e response, para ele utilizamos somente a response de um ping, retornando um json, com uma resposta Ping, este método pode ser utilizado de forma autenticada ou não. Veja o código abaixo:

```
apiMethods.Ping = (req, res) => {  
  try {  
    let ping = {};  
  
    ping.result = "Ping Bitpagg";  
  
    ping.requestHeaders = req.headers;  
  
    res.status(200).json(ping);  
  
  } catch (error) {  
    res.status(500).json(ping);  
  }  
}
```

Em seguida fica o função de geração o ticket da proxy:

```
apiMethods.Ticket = (req, res) => {  
  try {  
  
    let parameters = {  
      rota: req.route,  
      host: req.hostname,  
      origin: req.headers.origin,  
      methodCalled: req.url,  
      referer: req.headers.referer,  
      userRequest: req.socket.remoteAddress  
    };  
  
    res.status(200).json({  
      Ticket: authenticationMethods.GenerateToken(parameters)  
    });  
  
  } catch (error) {  
    res.status(500).json(error);  
  }  
}
```

b. ControllerExample

Esta sessão se trata de um exemplo de controller para requisições específicas da aplicação, diferentemente do controller de autenticação, este e outros serão criados para o projeto em específico.

Em nossa controller criamos uma função onde é estanciado inicialmente a função de geração do token bitpagg, em seguida encadeada no retorno desta função e construída a requisição para a API de dados da bitpagg, exemplo:

```
apiMethods.ListarGrupoCodigoVendedor = (req, res) => {  
  try {  
    let finalStatusCode = 0;  
    let Z_Scope = appFlow.zScope;  
    let Client_Id = appFlow.clientId;  
    let Grant_Type = appFlow.grantType;  
    let Client_Secret = appFlow.clientSecret;  
    let api = bitpaggAuthenticator.Methods();  
  
    api.Token(Client_Id, Client_Secret, Z_Scope, Grant_Type)  
  
    .then((result) => {  
      let returnToken = {};  
      returnToken.code = result.code;  
      returnToken.content = `${result.body.token_type} ${result.body.access_token}`;  
      return returnToken;  
    })  
    .then(result => {  
      let requestData = {};  
      requestData.url =  
`${appFlow.bitpaggApiUrlHost}/api/NegocioComum/Comum/ListarGrupoCodigoVendedor/${req.body.c  
od_pesquisa}`;  
      requestData.options = {  
        method: "GET",  
        headers: {  
          Authorization: result.content,  
          "Content-Type": "application/json",  
          origin: req.headers.origin,  
          referer: req.headers.referer  
        },  
        json: true  
      };  
      return requestData;  
    })  
    .then(result => fetch(result.url, result.options))  
    .then(result => {  
      finalStatusCode = result.status;  
      return result.json();  
    })  
    .then(result => res.status(finalStatusCode).json(result))  
    .catch((err) => {  
      console.log(err);  
    });  
  
  } catch (error) {  
    res.status(500).json(error);  
  }  
}
```

3. METHODS

a. Authenticator/index.js

Neste arquivo esta localizado a geração e autenticação do ticket gerado pela proxy

I. GenerateToken:

O método GenerateToken irá gerar o token e criptografá-lo utilizando métodos do arquivo cryptography, este método recebe parâmetros e payload, que são chamados no controller "AuthenticationController.js", desta forma ele utiliza de uuidv1 e uuidv4 concatenados com a data, utiliza todos os parâmetros recebidos para gerar uma keyToCipher, logo após esta key é criptografada e convertida em base64, assim o método utiliza a key criptografada mais o password definido na configurations mais as options também definidas no arquivo Configurations. Com todos os parâmetros definidos, utiliza-os em um jwt.sign, e retorna o ticket/Token. Veja o código abaixo:

```
static GenerateToken(parameters) {  
  
    let dynamicKey = parameters;  
    let privateKey = parameters.referer;  
    let dynamicKeyRandomBasedv1 = uuidv1().toString() + Date.now();  
    let dynamicKeyRandomBasedv4 = uuidv4().toString() + Date.now();  
    let dynamicKeyTimeBased = Date.now();  
  
    let keyToCipher = JSON.stringify({  
        dynamicKeyRandomBasedv1,  
        dynamicKey,  
        dynamicKeyRandomBasedv4,  
        dynamicKeyTimeBased,  
        privateKey  
    });  
  
    let returnCipher = Cryptography.createCipher(keyToCipher);  
  
    let buff = new Buffer(returnCipher);  
  
    let base64Data = buff.toString('base64');  
  
    let password = Configurations.JwTPassword;  
  
    let verifyOptions = {  
        issuer: Configurations.issuer,  
        subject: Configurations.subject,  
        audience: Configurations.audience,  
        expiresIn: Configurations.expiresIn  
    };  
  
    let unsignedTicket = {  
        'Ticket': base64Data  
    };  
  
    let Tiket = Jwt.sign(unsignedTicket, password, verifyOptions);  
  
    return Tiket;  
}
```


II. CheckToken

O método CheckToken é utilizado para fazer chamadas autenticadas, este método verifica primeiramente a existência de um token , e em seguida, de acordo com todas as configurações ele verifica todos os parâmetros citados anteriormente para a criação do token, além disso ele decriptografa o base64 do token recebido, logo após ele compara se o keyOfDecipher é valido ou não , através de um else if , verifica também se o url contido no AppFlow.json é igual ao contido no token de criptografado, em seguida utiliza- se o uso do terceiro método FlowApp para verificar o fluxo da aplicação, do qual será abordado logo em seguida.

Se a verificação de um if ou else if dentro do CheckToken for true, ele retornará uma response contendo a seguinte mensagem 'Unauthorized access attempt!', que significará falha na autenticação do token em chamadas autenticadas. Veja o código abaixo:

```
static CheckToken(req, res, next) {
  let Tiket = req.headers['x-access-token'];

  if (!Tiket) {
    return res.status(401).send({
      auth: false,
      message: 'Unauthorized access attempt! First authentication.'
    });
  }
  let verifyOptions = {
    issuer: Configurations.issuer,
    subject: Configurations.subject,
    audience: Configurations.audience,
    expiresIn: Configurations.expiresIn
  };
  let password = Configurations.JwtPassword;
  var legit = Jwt.verify(Tiket, password, verifyOptions);
  let Ticket = legit.Ticket;
  let buff = new Buffer(Ticket, 'base64');
  let data64Base = buff.toString('ascii');
  let returnDecipher = Cryptography.createDecipher(data64Base);
  let keyOfDecipher = JSON.parse(returnDecipher);
  if (!keyOfDecipher.dynamicKey.referer) {
    return res.status(401).send({
      auth: false,
      message: 'Unauthorized access attempt! Second authentication.'
    });
  } else if (FlowApp.appFlowClient.applicationUrl != keyOfDecipher.dynamicKey.origin) {
    return res.status(401).send({
      auth: false,
      message: 'Unauthorized access attempt! Third authentication.'
    });
  } else if (!AuthenticationMethods.validateFlow(FlowApp.appFlowClient.appFlow,
    keyOfDecipher, req.route.path, req.body, req.method)) {
    return res.status(401).send({
      auth: false,
      message: 'Unauthorized access attempt! Fourth authentication.'
    });
  }
  next();
}
```

```
}  
}
```

III. **validateFlow**

Este método valida se as configurações de AppFlow do token de criptografado estão contidas no AppFlow do proxy, ele valida se existe os steps, assim como foi pedido sua declaração anteriormente no tópico AppFlow.js, ele também valida se os métodos conferem, tudo isso ele faz através do método filter do js, desta forma o método verifica se os valores do token decriptografado condizem com os valores declarados em AppFlow.js .

O método ValidateFlow é utilizado como descrito acima no CheckToken, para verificar se os fluxos do token conferem com o AppFlow.json do proxy em questão, ele é utilizado na cadeia de else if para retornar ou não 'Unauthorized access attempt!'.

Além de verificar os parâmetros de AppFlow, o método validateFlow valida o payload enviado no post, ele compara se o payload ao criptografar o token e o mesmo do payload na hora de checkar o token em uma chamada autenticada. Veja o código abaixo:

```
static validateFlow(appFlow, keyOfDecipher, path, body, method) {  
    if (!appFlow) {  
        return false;  
    }  
  
    let stepFlowResult = appFlow.filter(x => (FlowApp.appFlowClient.applicationUrl + x.stepPath) ==  
keyOfDecipher.dynamicKey.referer.replace("/#", ""));  
  
    if (!stepFlowResult || stepFlowResult.length < 1) {  
        return false;  
    }  
  
    let stepFlowMethodResult = stepFlowResult[0].allowFunctions.filter(x => x.resource == path);  
  
    if (!stepFlowMethodResult || stepFlowMethodResult.length < 1) {  
        return false;  
    }  
  
    if (stepFlowMethodResult[0].method != method.toUpperCase()) {  
        return false;  
    }  
  
    return true;  
}
```

b. Cryptography.js

O arquivo de criptografia contém uma classe chamada `Cryptography`, nela temos dois métodos, são eles `createCipher` e `CreateDecipher`.

I. createCipher

O método `createCipher`, recebe uma `stringKey`. Logo após ele a criptografa utilizando o “crypto” assim como ilustrado abaixo:

```
var mykey = crypto.createCipher(configurations.cryptographyAlgorithm,
configurations.cryptographyPassword);
```

O `mykey` leva como parâmetros, o `algorithm` e `password default`, dos quais não devem ser alterados.

Após a criptografia para o algoritmo `aes256`, o método transforma a string recebida de `utf8` para o `hex` utilizando o comando `update` e passando a string como parâmetro, pode se ver abaixo:

```
var mystr = mykey.update(stringKey, 'utf8', 'hex');
mystr += mykey.final('hex');
return mystr;
```

II. createDecipher

Assim como o método anterior ele recebe uma string, porém com o intuito de descriptografar-la, desta forma utiliza-se o `crypto` para descriptografar do `aes256` a string recebida:

```
var mykey = crypto.createDecipher(configurations.cryptographyAlgorithm,
configurations.cryptographyPassword);
```

Da maneira como foi feita a encriptação, realiza-se o procedimento contrário para a descriptografia, convertendo de “hex” para “utf8” e em seguida utilizando o

método `.final` para usar o output encoding “utf8” e retornar a string :

```
var mystr = mykey.update(stringKey, 'hex', 'utf8')
    mystr += mykey.final('utf8');
    return mystr;
```

4. ROUTES

a. index.js

O arquivo de configuração de rotas conterá todas as rotas previstas nos controllers utilizados para fazer as requisições na aplicação, todas as requisições para API's da BitPagg deverão ser autenticadas, para utilizar desta autenticação é importado por default no modelo do proxy a classe “AuthenticationMethods” do arquivo de Authentication descrito na seção 4.1, para fazer a autenticação de uma requisição utilizaremos o método de CheckToken, este método deve ser chamado dentro da declaração da rota, veremos logo a seguir.

Previamente antes de declarar nossas rotas criamos um objeto Router do express:

```
const router = express.Router();
```

Agora para declararmos as rotas utiliza-se a propriedade do Router de acordo com a chamada que se deseja fazer para aquela rota, por exemplo get, post , delete e etc, desta forma passamos como primeiro parâmetro a rota da requisição por exemplo “/seguradora/CotacaoSeguro” , logo após passamos o parâmetro de nosso CheckToken, “AuthenticationMethods.CheckToken” e por último o controller que foi criado em específico para esta chamada. Veja abaixo um exemplo:

```
router.post('/Seguradora/CotacaoSeguro',AuthenticationMethods.CheckToken,Seguradora/Cotacao);
```

Nota: A rota apontada acima, deve fazer referência a mesma rota criada na aplicação angular em `app.service.ts`, assim como o método deve ser o mesmo.

5. **SERVER.JS**

O arquivo Server.js é o arquivo de entrada do proxy, é nele que criamos um servidor Express, que é utilizado nas chamadas API e na criação do proxy em si, o express possibilita a utilização de bibliotecas de segurança para o proxy, desta forma é utilizado a biblioteca helmet, Morgan, cors e body parse.

Alem disso é feito um teste de caixa preta para a verificação da origem da

requisição. Esse teste deve sempre ser realizado após a declaração `“const app = express();”`

Utilizamos a biblioteca helmet, com os seguintes atributos: helmet.frameguard: Utilizado para negar ou permitir o enquadramento de sua

aplicação helmet.hsts : Navegadores devem aderir ao HTTPS e nunca visitar com o protocolo HTTP. helmet.noSniff: Impede que o navegador realize um sniff nos MIME TYPES. helmet.noCache: Previne a gravação de cache pelo navegador.

helmet.xssFilter: Previne cross site scripting

helmet.hidePoweredBy: Esconde informações referentes a tecnologia do sistema.

helmet.dnsPrefetchControl: Outro middleware a ser adicionado no server express, ele permite desativar a pré-busca de DNS dos navegadores.

Utiliza-se o Morgan para adicionar middlewares de segurança em nosso server express.

bodyParse.urlencoded: Permite escolher entre analisar os dados codificados em url com a query string biblioteca (quando false) ou a qs biblioteca (quando true).

bodyParser.json: Verifica os objetos enviados pelo corpo da requisição.

Após todas as configurações do servidor express, definimos a configuração de rota do mesmo, adicionamos ao servidor o objeto de rotas que aponta para nosso RoutesConfig.js, e também o endpoint da aplicação com o prefixo node desta forma `${root_configuration.nodeAppPath}`, o nodeAppPath seria o nome de sua aplicação definido previamente no appsettings.json , um exemplo seria o

host + \${root_configuration.nodeAppPath} , o endpoint dentro do arquivo da aplicação ficaria `http://localhost:3000/node/aplicação/`.

Por fim para iniciar o servidor utilizaremos `app.listen` , para ambiente em desenvolvimento utilizará por padrão a porta 3000 , e ao levar a aplicação para homologação ou produção , será utilizado uma variável dentro do processo de deploy para mudar o ambiente da proxy.

```
app.listen(process.env.PORT || root_configuration.nodeAppPort, () => {  
  console.log(`Express started at  
http://localhost:${root_configuration.nodeAppPort}${root_configuration.nodeAppPath}`)  
});
```

O exemplo final ficaria da seguinte forma:

```
const bodyParser = require('body-parser');  
const express = require('express');  
const morgan = require('morgan');  
const helmet = require('helmet');  
  
const routes = require('./app/Routes')  
const root_configuration = require('./app/Configuration/app.settings.json');  
  
const app = express();  
  
##region body-parser  
app.use(bodyParser.json());  
app.use(bodyParser.urlencoded({ extended: true }));  
##endregion  
  
##region helmet  
app.use(helmet());  
app.use(helmet.hsts());  
app.use(helmet.noSniff());  
app.use(helmet.xssFilter());  
app.use(helmet.hidePoweredBy());  
app.use(helmet.dnsPrefetchControl());  
##endregion  
  
##region Log node server  
app.use(morgan('combined'));  
##endregion
```

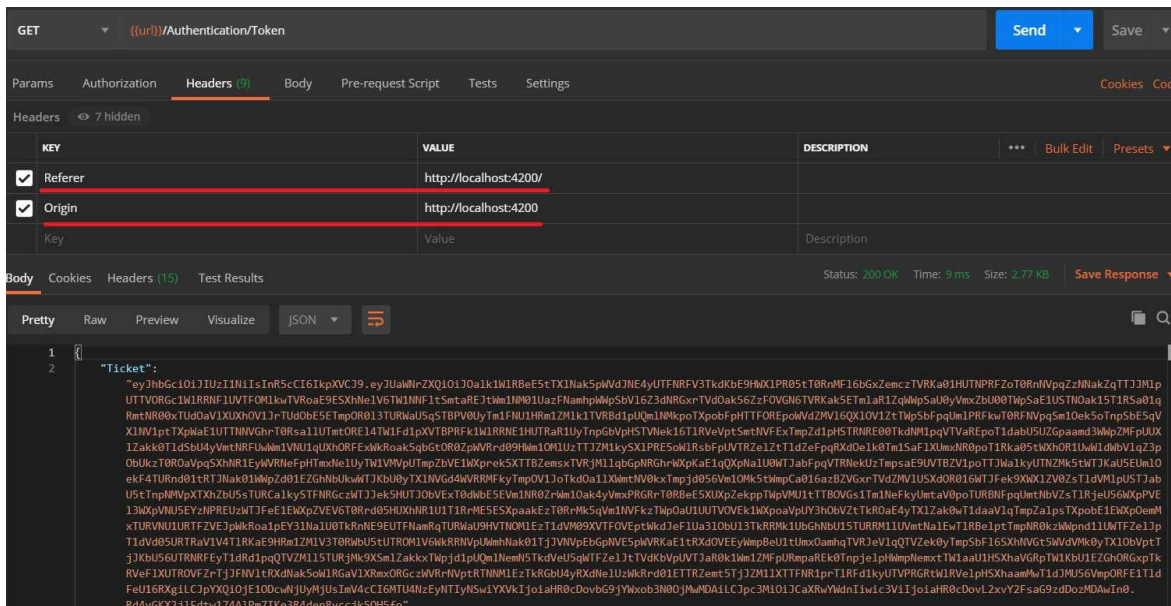
```
app.use(root_configuration.nodeAppPath, routes);

app.listen(process.env.PORT || root_configuration.nodeAppPort, () => {
  console.log(`Express started at
http://localhost:${root_configuration.nodeAppPort}${root_configuration.nodeAppPath}`)
});
```

6. Testes no postman

Os testes realizados no postman precisam de alguns parametros para que funcionem da maneira correta. Para a geração do ping não é necessário realizar modificações. Para o ping autenticado é necessário passar o token gerado, e para gerar o token é necessário passar alguns parametros.

Para gerar o token é necessário declarar nos HEADERS a chaves “Referer” e “Origin” que são referentes a rota da aplicação, declarada no arquivo appFlow.json no campo “applicationUrl”



Com o ticket gerado, copie ele, pois será necessário utiliza-lo no ping autenticado e nas demais requisições. Os campos para o ping autenticado são mantido os “Referer” e “origin” e tambem adicionados o campo de “content-Type” e “x-access- token”. No content-type o valor é “application/json” e no x-access-tokken é o ticket gerado

