08

Implementando as funcionalidades básicas do Payfast

Implementando as funcionalidades básicas do PayFast

Até agora a aplicação PayFast já está funcionando e é capaz de atender uma rota, que é a rota de listar pagamentos, porém ainda não é possível sequer que se cadastre um novo pagamento. Essa e as outras funcionalidades básicas para que a *api* de pagamentos fique funcional serão desenvolvidas ao longo deste capítulo.

Recebendo dados na requisição

A primeira funcionalidade a ser implementada é a que disponibiliza uma rota para receber os dados de um novo pagamento. A forma de criar uma nova rota já é conhecida e não tem muita novidade. Basta definí-la em algum *controller* com sua *url* e o verbo *HTTP* aceito.

Como esta é mais uma funcionalidade fortemente atrelada à entidade 'Pagamento', faz bastante sentido que o *controller* a ser utilizado seja o `pagamentos.js`:

```
app.post("/pagamentos/pagamento",function(req, res) {
    res.send('ok');
});
```

Uma nova rota foi criada, porém ela está fazendo a mesma coisa que a rota que listava os pagamentos: simplesmente retornando um 'ok' para o usuário. A ideia é que agora as rotas passem a implementar suas reais funcionalidades. Pensando nisso, sabe-se que essa rota, por ser uma funcionalidade de cadastro, precisa necessariamente receber dados como entrada quando for consultada. Mas como fazer isso já que a aplicação PayFast é apenas uma *api* e não possui uma tela de interface com o usuário?

Enviando uma requisição HTTP com dados

A maneira mais comum de enviar dados em uma requisição *HTTP* é mandá-los no **body** da requisição. De preferência utilizando um método *HTTP* cuja semântica seja compatível com essa necessadidade. A nova rota já foi definida com o método **POST**, então está de acordo. Veja agora como ficaria um exemplo dessa requisição utilizando o *CURL*:

```
curl http://localhost:3000/pagamentos/pagamento \
-X POST \
-v \
-H "Content-type: application/json" \
-d '{
        "forma_de_pagamento": "payfast",
        "valor": "10.87",
        "moeda": "BRL",
        "descricao": "descrição do pagamento"
    }'
```

Na primeira linha foi definida a url de acordo com a rota implementada, em seguida o verbo *POST* através do parâmetro `-X`. O parâmetro `-v` indica que a operação deve ser mais 'verbosa' que o *default*, ou seja, deve imprimir uma maior quantidade de dados descritivos no console.

O parâmetro `-H` indica que está sendo passado um **header** *HTTP*, indicando que o tipo do conteudo enviado terá um formato `json`. A própria convenção do *HTTP* define um *header* específico para quando se quer passar esse tipo de informação, que é o `Content-type`, exatamente o que foi utilzado no código. O conteudo que ele recebeu também e uma convenção que define o tipo do dado como *json*: **application/json**.

O formato *json* foi definido por ser um formato que conversa muito bem com o `node.js` e várias tecnologias web em geral. Afinal ele nasceu exatamente com esse propósito e por isso é um dos campeões de uso em *apis*, como a do **PayFast**.

Por fim, o parâmetro `-d` indica o que se quer enviar de fato como corpo da requisição. No caso do exemplo em questão, o corpo é exatamente um *json* que possui os dados do pagamento a ser cadastrado.

Preparando a rota para receber os dados

Agora que já se sabe como enviar os dados, é preciso programar a rota para que o código saiba recebê-los e definir o que fazer com eles.

Dentro da função que implementa a rota, o objeto do **Request**, no caso representado pela variável **req** é quem conhece os dados da requisição, portanto é para ele que se deve pedir qualquer informação que se precise. Para receber o *body* inteiro basta acessar um atributo com o mesmo nome dentro do **req**:

```
app.post("/pagamentos/pagamento",function(req, res) {
    var pagamento = req.body;
    console.log(pagamento);
    res.send('ok');
});
```

Agora é preciso ensinar ao express que ele deve usar o **body-parser** para recuperar os parâmetros enviados na requisição e deixar disponível na propriedade **body**. Como isso é um código de configuração, o ideal é que fique no arquivo `config/custom-express.js`. O código deve vir antes do carregamento dos módulos, feito pelo *consign*.

```
npm install body-parser --save
```

```
var express = require('express');
var consign = require('consign');

var bodyParser = require('body-parser');

module.exports = function() {

  var app = express();

  app.use(bodyParser.urlencoded({extended: true}));
  app.use(bodyParser.json());

  consign()
   .include('controllers')
   .into(app);

}
```

Pronto! Agora se o mesmo comando *CURL* for executado novamente, é esperado que o pagamento enviado seja conrretamente impresso no terminal onde o *PayFast* está rodando.

O próprio módulo do *bodyParser* já possui algumas funções representando os tipos de dados mais comuns de serem utilizados em projetos web, como o PayFast. Cabe ao programador somente identificar aqueles que são interessantes apra a aplicação e fazer suas devidas chamadas atribuindo-os finalmente ao objeto do *express*. Isso foi feito no código àcima para os tipos *json* e *urlencoded* que é o tipo *default* de dados trafegado em requisições *HTTP*.

Persistindo o novo pagamento no banco de dados

O *PayFast* já está consegue receber um pagamento vindo de uma requisição do cliente, mas até então não sabe fazer nada com esse pagamento. O objetivo que essa informação fique persistida em uma base dados e em seguida o *PayFast* dê uma resposta para o cliente sobre o que aconteceu com a sua requisição. Esse são os próximos passos a ser implmentados.

O banco a ser utilizado será o MySQL. Então a primeira tarefa é justamente criar a tabela para armazenar os pagamentos. Abaixo o *script* que cria a tabela desejada:

```
CREATE TABLE `pagamentos` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
`forma_de_pagamento` varchar(255) NOT NULL,
`valor` decimal(10,2) NOT NULL,
`moeda` varchar(3) NOT NULL,
`status` varchar(255) NOT NULL,
`data` DATE,
`descricao` text,
 PRIMARY KEY (id)
);
```

Agora que já temos a tabela criada e alguns produtos adicionados, chegou a hora de realizarmos a query. Assim como em outras linguagens, não queremos ter que lidar com detalhes do protocolo do banco de dados, independende de qual seja. Para resolver este problema a comunidade Javascript, em volta do Node.js, já desenvolveu um driver de acesso ao MySQL. Assim como foi com o *express* e todas as outras libs utilizadas até agora, precisamos adicioná-lo ao nosso projeto para conseguir usá-lo.

```
npm install --save mysql
```

Integrando a rota com a camada de persistência

Para ter um código bem organizado e fácil de manter é interessante isolar a camada de persistência e, de preferência, utilizar *patterns* já conhecidos para acessá-la. O PayFast aproveitará do mesmo modelo já visto no curso anterior de Node.js e utilizará o *patter* **DAO** para implementar a classe que terá toda essa responsabilidade de acesso à banco de dados. E o *pattern* **Factory** para a classe que será responsável por criar as conexões com o banco.

Para garantir a organização será criada uma pasta chamada `persistencia` e estes novos arquivos ficarão dentro dela.

Primeiro veja como fica o *DAO* que será nomeado `PagamentoDao.js` :

```
function PagamentoDao(connection) {
  this._connection = connection;
```

```
}

PagamentoDao.prototype.salva = function(pagamento,callback) {
    this._connection.query('INSERT INTO pagamentos SET ?', pagamento, callback);
}

PagamentoDao.prototype.lista = function(callback) {
    this._connection.query('select * from pagamentos',callback);
}

PagamentoDao.prototype.buscaPorId = function (id,callback) {
    this._connection.query("select * from pagamentos where id = ?",[id],callback);
}

module.exports = function(){
    return PagamentoDao;
};
```

O arquivo que implementa a *factory* responsável pela conexões será a `connectionFactory.js` :

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

function createDBConnection(){
  return mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'payfast'
  });
}

module.exports = function() {
  return createDBConnection;
}
```

Pronto! Agora que o acesso ao banco, já está devidamente implementado, basta impementar as chamadas à essa nova camada nas rotas que tenham essa necessidade:

```
...
app.post("/pagamentos/pagamento",function(req, res) {
  var pagamento = req.body;
  console.log('processando pagamento...');

  var connection = app.persistencia.connectionFactory();
  var pagamentoDao = new app.persistencia.PagamentoDao(connection);

  pagamentoDao.salva(pagamento, function(exception, result){
    console.log('pagamento criado: ' + result);
    res.json(pagamento);
  });
});
...
```

A rota que insere um pagamento através de uma requisição *POST* agora deve primeiramente pedir uma conexão para a *connectionFactory* e em seguida instanciar um novo objeto de *PagamentoDao* para, a partir dele, conseguir invocar o método *salva*.

Note que tanto a *connectionFactory* quanto o *PagamentoDao* foram obtidos dentro do *controller* sem a necesidade de utilizar um **require**. Bastou que eles fossem identificados pelo caminho completo de seus diretórios.

Essa é uma estratégia muito bacana, mas para que funcione de fato, é preciso que o objeto do **express** passe a conhecer essa nova estrutura de diretórios que foi criada. Essa informação deve ser passada no aquivo de configurações do **express**, o `custom-express.js` , informando ao **consign** que ele deve conhecer mais um diretório no seu carregamento:

```
...
module.exports = function() {

  ...

  consign()
   .include('controllers')
   .then('persistencia')
   .into(app);
  return app;
}
```

Com toda essa configuração pronta, basta agora fazer as alterações necessárias no objeto pagamento antes que ele seja persistido. Voltando ao *controller* de pagamentos, a regra de negócio pede que cada novo pagamento seja marcado com o status *CRIADO* e que também tenha a informação da data em que foi criado:

```
...
app.post("/pagamentos/pagamento",function(req, res) {
  var pagamento = req.body;
  console.log('processando pagamento...');

  var connection = app.persistencia.connectionFactory();
  var pagamentoDao = new app.persistencia.PagamentoDao(connection);

  pagamento.status = "CRIADO";
  pagamento.data = new Date;

  pagamentoDao.salva(pagamento, function(exception, result){
    console.log('pagamento criado: ' + result);
    res.json(pagamento);
  });
});
...
```

Agora é só enviar novamente o *curl* que informa um novo pagamento e ele deverá ser inserido no MySQL conforme esperado.

Validação de dados com express-validator

O PayFast já é capaz de receber um novo pagamento e salvá-lo no banco conforme desejado, mas o sistema ainda peca em um aspecto bem básico: não existe uma validação para os dados enviados. Quando se implementa uma *api* é muito

importante que a entrada de dados seja bem tratada e que seja dado um retorno adequando ao cliente em caso de alguma inconsistência.

Para facilitar a vida, ao invés de ficar gastando tempo implementando código simples de validação, vamos usar um outro módulo construído para o *express*, chamado **express-validator**. Como é de praxe, a primeira coisa que precisamos fazer é instalar o módulo na aplicação.

```
npm install express-validator --save
```

Também precisamos carregá-lo, dentro da nossa aplicação. Então vamos alterar o arquivo *custom-express.js*.

```
var express = require('express');
...
var expressValidator = require('express-validator');

module.exports = function() {
  ...
  app.use(bodyParser.urlencoded({extended: true}));
  app.use(bodyParser.json());

  //Obrigatoriamente logo apos o bodyParser
  app.use(expressValidator());
  ...
  return app;
}
```

A mudança aqui é bem básica: somente informar ao objeto do **express** que ele deve saber usar mais um novo módulo.

Com tudo configurado, resta somente implementar a validação desejada de fato. Para a rota de cadastro de pagamento é importante que o dado informado contenha todas as informações obrigatórias e com seus respectivos formatos de dados adequados.

- Forma de pagamento deve ser obrigatória.
- Valor deve ser obrigatório e um número real.
- Moeda também deve ser obrigatória e conter exatamente 3 dígitos.

```
...
app.post("/pagamentos/pagamento",function(req, res) {
    var pagamento = req.body;

    req.assert("forma_de_pagamento", "Forma de pagamento é obrigatória.").notEmpty();
    req.assert("valor", "Valor é obrigatório e deve ser um decimal.").notEmpty().isFloat();
    req.assert("moeda", "Moeda é obrigatória e deve ter 3 caracteres").notEmpty().len(3,3);

    var errors = req.validationErrors();

    if (errors){
        console.log("Erros de validação encontrados");
        res.status(400).send(errors);
        return;
    }
```

```
        console.log('processando pagamento...');
    ...
```

A validação pode ser implementada diretamente na rota. A função `assert` também faz parte do módulo **express-validator**, mas como foi informado ao **express** que ele deve saber utilizar este novo módulo e ambos estão pronto para essa integração, é possível invocá-la a partir do próprio parâmetro `req`, que é um objeto do express e não necessariamente do validator.

A função `assert` retorna um objeto cujo o tipo é **ValidatorChain**. Este objeto possui as funções de validação que desejamos aplicar na propriedade especificada. Por debaixo do pano, o *express-validator* usa um outro módulo chamado *validator*, que fornece muitas funções de validação. Basicamente ele é a cola entre esse módulo e o *express*. A função *errors* é adicionada no request para possibilitar que recuperemos os possíveis erros gerados.

Veja também que cada tipo de validação já possui uma função específica no **express-validator** que a implementa e que elas podem ser invocadas de maneira incremental:

- notEmpty(): indica que o campo não deve ser vazio.
- isFloat(): verifica se o dado realmente é um número real válido.
- len(): indicada a quantidade mínima e máxima de caracteres aceitos na String.

Existem diversas outras funções já prontas no express-validator para diferentes tipos de situações, além de ser possível criar as próprias validações e adicioná-las ao projeto.

Usando melhor os recursos do HTTP

No momento em que erros, foram encontrados na validação, passamos a retornar para o cliente estes erros e uma informação específica do HTTP, que foi o **status code 400**:

```
...
  if (errors){
        console.log("Erros de validação encontrados");
        res.status(400).send(errors);
        return;
    }
...
```

Esse código indica que a requisição veio com dados inválidos. E essa é uma parte muito importante numa integração via REST. O cliente da aplicação vai se apoiar justamente nesse status para saber o que fazer com a resposta do servidor. Por exemplo, caso você retorne 200, ele pode achar que deu tudo certo na requisição, mesmo que o servidor tenha enviado um JSON com os erros.

Existem diversos outros **status code HTTP** e conhecê-los e utilizá-los corretamente é extremamente importância para o bom desenho de uma **api REST**. Veja abaixo os principais códigos e seus significados:

- 100 Continue: o servidor recebeu a solicitação e o cliente pode continuar a comunicação.
- 200 Ok: tudo ocorreu como esperado.
- 201 Created: um novo recurso foi criado no servidor.
- 301 Moved: a url solicitada foi movida.
- 400 Bad Request: problemas na requisição do cliente.
- 404 Not Found: a url solicitada não foi encontrada.

- 500 Internal Server Error: algo inesperado aconteceu do lado do servidor.

Além destes, existem diversos outros que valem muito a pena ser conhecidos e estudados.

Note que cada centena corresponde à uma categoria específica de informação. A família do **100** indica uma **conexão continuada**. A família do **200** indica **sucesso**. **300** significa **redirecionamento**. **400** é para **erro do cliente** e finalmente a família do **500** é usada para informar **outros erros**, em sua maioria do lado do servidor.

O bom uso dos **status code HTTP**, portanto não se restringe somente às situações de falha. Na verdade eles devem ser explorados ao máximo da melhor maneira possível sempre que a *api* precise passar qualquer informação para o cliente.

Quando ocorrer tudo bem no cadastro de um novo pagamento, por exemplo, é importante que o cliente saiba que um novo recurso foi **criado** do lado do servidor e qual é a nova **localização** que foi gerada para que esse recurso possa ser acessado. No caso do PayFast, a nova localização é a url para consulta de pagamentos acompanhadas do novo **id** gerado pelo banco.

Em HTTP a forma de identificar essa nova localização é através do atributo **location**. Essa alteração pode ser feita, portanto na rota que cadastra o pagamento:

```
pagamentoDao.salva(pagamento, function(exception, result){
    console.log('pagamento criado: ' + result);

    res.location('/pagamentos/pagamento/' + result.insertId);

    pagamento.id = result.insertId;

    res.status(201).json(pagamento);
});
```

O objeto **result** retornado pelo módulo *mysql* ***já traz consigo a informação do novo id gerado. Sendo assim basta pegar essa informação e concatenar com a url 'pagamentos' para montar a nova*** *location**.

Agora ao executar o *curl* que insere um pagamento, teremos uma saída semelhante à que segue abaixo:

```
> POST /pagamentos/pagamento HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.43.0
> Accept: */*
> Content-type: application/json
> Content-Length: 104

...

< HTTP/1.1 201 Created
< Location: pagamentos/1
```