11

Integrações entre serviços

O estilo Arquitetural REST

A sigla REST vêm de ***Re****presentational* ***S****tate* ***T****ransfer* e surgiu da tese de doutorado de Roy Fielding, descrevendo as ideias que levaram à criação do protocolo HTTP. A Web é o maior exemplo de uso de uma arquitetura REST, onde os verbos são as operações disponíveis no protocolo (GET, POST, DELETE, PUT, OPTION...), os recursos são identificados pelas URLs e as representações podem ser definidas com o uso de *Mime Types*(texto, XML, JSON e outros).

Orientado ao recurso

REST não é uma especificação nem uma tecnologia, é um modelo arquitetural. Neste modelo, o pensamento da aplicação gira em torno dos recursos. Depois de definir os recursos, usamos os verbos disponíveis (no HTTP temos o GET, POST, PUT e outros) para manipular estes recursos.

Uma das ideias da arquitetura REST é aproveitar ao máximo o protocolo de comunicação (HTTP) usando-o direito. Nesse escopo, um dos princípios que o REST prega é a utilização das URIs de acordo com sua denominação, URI é a sigla para **U**niform **R**esource **I**dentifier (Identificador Uniforme de Recurso). Pensando voltado a recursos, podemos imaginar que cada recurso tem um identificador único perante os usúarios.

Casando com a ideia de manipular os recursos usando os verbos do protocolo, se cada recurso tem um único endereço, basta fazer uma requisição do tipo POST para criar um novo recurso ou fazer uma requisição GET para buscar o recurso. Tudo gira em torno do recurso.

Diferentes representações

Os recursos em geral estarão "guardados" ou gerados no servidor, então o cliente da aplicação não pode simplesmente pegá-lo, no máximo ele pode vizualizá-lo. Porém, essa visualização pode ocorrer de várias maneiras, por meio de uma página HTML, uma interface desktop, mobile e etc.

Parte da ideia REST foca que em cada recurso pode ter suas representações. Cada representação pode ter seu formato específico, por exemplo, via XML, HTML, JSON ou

Métodos HTTP

Ao desenhar aplicações REST, pensamos nos recursos a serem disponibilizados pela aplicação e em seus formatos, em vez de pensar nas operações.

As operações disponíveis para cada um dos recursos no protocolo HTTP são:

- **GET**: retorna uma representação do recurso
- **POST**: cria ou altera o recurso
- **PUT**: cria ou altera o recurso
- **DELETE**: remove o recurso
- outras menos comuns, como **HEAD** e **OPTIONS**

Os quatro principais verbos do protocolo HTTP são comumente associados às operações de CRUD em sistemas *Restful* (POST -> INSERT, GET -> SELECT, PUT -> UPDATE, DELETE -> DELETE). Há uma grande discussão dos motivos pelos quais usamos *POST* para criação (*INSERT*) e *PUT* para alteração (*UPDATE*). A razão principal é que o protocolo HTTP especifica que a operação PUT deve ser *idempotente*, já POST não.

> **Idempotência e SAFE**
>
> Operações idempotentes são operações que podem ser chamadas uma ou mais vezes, sempre com o mesmo resultado final.
>
> Uma operação é chamada SAFE se ela não altera nenhuma representação.
>
> Idempotência e SAFE são propriedades das operações e fundamentais para a escalabilidade da Web.

Rota para confirmar pagamentos

Utilizando então o método PUT, implementamos uma nova rota para confirmar pagamentos. Essa rota recebe uma requisição na mesma url que usamos para criar pagamentos, "/pagamentos/pagamento", só que dessa vez com o id do pagamento a ser confirmado também fazendo parte da url. Na execução dessa operação alteramos o status do pagamento para criado e persistimos esse objeto atualizado no banco.

```
app.put('/pagamentos/pagamento/:id', function(req, res){

    var pagamento = {};
    var id = req.params.id;

    pagamento.id = id;
    pagamento.status = 'CONFIRMADO';

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

    pagamentoDao.atualiza(pagamento, function(erro){
        if (erro){
          res.status(500).send(erro);
          return;
        }
        console.log('pagamento criado');
        res.send(pagamento);
    });

  });
```

Para que essa execução seja completada corretamente, foi preciso também criar a função que atualiza o status de um pagamento no banco no arquivo PagamentoDao:

```
PagamentoDao.prototype.atualiza = function(pagamento,callback) {
    this._connection.query('UPDATE pagamentos SET status = ? where id = ?', [pagamento.status, pagar
}
```

Rota para cancelar pagamentos

Analogamente à rota de confirmar um pagamento, foi também criada a rota para cancelar um pagamento. Ela tem sua operação exposta pela mesma URL que a anterior, com a diferença que o verbo atendido aqui é o DELETE.

```
app.delete('/pagamentos/pagamento/:id', function(req, res){
    var pagamento = {};
    var id = req.params.id;

    pagamento.id = id;
    pagamento.status = 'CANCELADO';

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

    pagamentoDao.atualiza(pagamento, function(erro){
        if (erro){
          res.status(500).send(erro);
          return;
        }
        console.log('pagamento cancelado');
        res.status(204).send(pagamento);
    });
  });
```

Nesse caso tivemos uma outra novidade que é o uso de um novo Status Code HTTP: o 204 indicando que o servidor conclui a requisição com sucesso, mas que agora não tem nenhum conteudo para retornar. O que faz sentido visto que o conteudo acessado nessa url por esse id acaba de ser desabilitado mesmo.

Hipermídia

Os recursos serão apresentados por meio de representações. Seguindo os princípios RESTful, representações devem ser interligadas umas com a outras. Isso é chamado **hipermídia** e conhecido na Web através de *hyperlinks*. No nosso exemplo, a representações de um livro poderia conter a URI dos autores. Como resultado disso, é possível navegar entre os recursos.

Mais sobre hipermídia no blog da Caelum:

http://blog.caelum.com.br/hipermidia-e-contratos-dinamicos-menor-acoplamento/ (http://blog.caelum.com.br/hipermidia-e-contratos-dinamicos-menor-acoplamento/)

> **RESTful**
>
> Qualquer sistema que aplique as ideias do estilo arquitetural REST, pode ser chamado de RESTful. Existe uma intensa discussão na comunidade sobre quando um sistema pode ser considerado RESTful ou não, porém, na maioria dos casos, basta apenas implantar uma parte do REST (em especial pensar em recursos, verbos fixos e ligações entre apresentações) para ser chamado de RESTfull

Vantagens Restful

Por quê (ou quando) usar uma arquitetura REST? A primeira coisa que deveríamos saber é quais são as vantagens do REST: http://en.wikipedia.org/wiki/Representational_State_Transfer#Claimed_benefits (http://en.wikipedia.org/wiki/Representational_State_Transfer#Claimed_benefits)

- Protocolos menos complexos: Não precisamos de tantos protocolos para enviar e receber informações, basta que as duas partes estejam de acordo com os recursos e representações (formatos) disponíveis.
- Mais poder e flexibilidade: Por não precisarmos nos preocupar com dezenas de protocolos, temos maior liberdade na hora de devolver um recurso, por exemplo, se usarmos SOAP precisamos necessariamente devolver um recurso no formato de XML, já com a ideia do REST, podemos devolver um objeto JSON direto para uma página na Web.
- Arquitetura amplamente disponível: Em linhas gerais, a arquitetura REST é mais simples, essa simplicidade permite que sua adoção seja mais fácil, com uma curva de aprendizado menor. Com isso é mais fácil disponibilizar seus serviços através do REST.
- Menos overhead de protocolo: As requisições em uma arquitetura REST são menores, pois trabalhamos com menos protocolos, por exemplo, não precisamos enviar todas informações definidas no protocolo SOAP, basta padronizar com o cliente o XML que ele receberá para os recursos.

Coreografia de serviços com HATEOAS

Um pagamento *nasce* a partir de uma transação com estado *CRIADO* e pode ser *CONFIRMADO* ou *CANCELADO* pelo cliente. *Confirmar* representa um "próximo passo" na vida do pagamento e pode ou não ser seguido pelo cliente. A ideia principal é que um recurso informe ao cliente quais os próximos passos ou relacionamentos, e atrás de cada relacionamento há um serviço transformador de dados.

Uma vez criado um pagamento vamos então receber os dados dele e também os relacionamentos:

```
{"id":3,"status":"CRIADO","valor":29.9,
  "links":[
    {"rel":"confirmar","uri":"/pagamentos/pagamento/3","method":"PUT"},
    {"rel":"cancelar","uri":"/pagamentos/pagamento/3","method":"DELETE"}
  ]
}
```

No entanto, um pagamento no estado *CONFIRMADO*, podemos pedir apenas informações sobre o pagamento. Assim, a apresentação do recurso, além dos dados, também leva sempre as informações sobre o que é permitido executar:

```
{"id":3,"status":"CONFIRMADO","valor":29.9,
  "links":[
    {"rel":"self","uri":"/pagamentos/pagamento/3","method":"GET"}
  ]
}
```

Essa forma de juntar os dados às ações é conhecida como *hypermedia* e é a essência do **HATEOAS** - ***H**ypermedia **a**s **t**he **E**ngine **o**f **A**pplication **S**tate*.

Para que seja aplicado o **HATEOAS** ao PayFast basta que cada rota passe a adicionar no *body* do seu *response* as informações com os próximos links possíveis a serem seguidos após a requisição que foi feita.

Implementando um cliente RESTfull com restify

A primeira integração demandada ao PayFast é que ele passe validar dados de pagamento via cartão cartão de crédito em uma *api* REST de uma operadora de cartões de crédito.

Essa *api* poderia ter sido tranquilamente escrita em Node.js utilizando tudo que já vimos até aqui. Veja abaixo um exemplo simples de um arquivo que implementa a funcionalidade mais básica desse serviço, que é autorizar um pagamento.

```
module.exports = function(app) {
    app.post("/cartoes/autoriza",function(req, res) {
      console.log('processando pagamento com cartão');

        var cartao = req.body;

        req.assert("numero", "Número é obrigatório e deve ter 16 caracteres.").notEmpty().len(16
        req.assert("bandeira", "Bandeira do cartão é obrigatória.").notEmpty();
        req.assert("ano_de_expiracao", "Ano de expiração é obrigatório e deve ter 4 caracteres.'
        req.assert("mes_de_expiracao", "Mês de expiração é obrigatório e deve ter 2 caracteres")
        req.assert("cvv", "CVV é obrigatório 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;
        }
        cartao.status = 'AUTORIZADO';

        var response = {
          dados_do_cartao: cartao,
        }

        res.status(201).json(response);
        return;
    });
  }
```

Agora o desafio é consumir esse serviço. Já é sabido que vamos usar um módulo externo, **restify**, portanto a primeira coisa a fazer é sua instalação via npm.

```
npm install --save restify
```

O próximo passo é implementar um arquivo que funcione como um cliente para consumir o serviço de cartões. Para manter o requisito de isolar códigos semelhantes no projeto, vamos criar o diretório `servicos` na raiz do projeto e dentro dele o arquivo `CartoesClient.js` , o primeiro cliente para serviços externos do *PayFast*.

A primeira coisa a se fazer nesse serviço é, obviamente, fazer o require da lib **restify**:

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

O **restify** possui alguns métodos já prontos para prover um objeto cliente de acordo com o serviço que se deseja consumir. Para consumir o serviço de cartões por exemplo, que já sabemos que retorna um json, é possível utilizar a a sua função específica para criação de um cliente *json*. Basta que ela seja invocada a partir do objeto onde foi carregado o módulo:

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

var client = restify.createJsonClient({
    url: 'http://localhost:3000',
    version: '~1.0'
  });
```

A função `createJsonClient` recebe como parâmetro um json, onde podemos passar todas as informações relevantes para consumir o serviço. A principal delas é a url do *endpoint* do serviço, mas é possível passar inúmeras outras, como a versão que deve ser consumida, por exemplo.

O retorno dessa função é um objeto a partir do qual é possível fazer a chamada dos serviços, simplesmente invocando a função que tenha o mesmo nome do verbo HTTP desejado. Veja um exemplo com POST:

```
client.post('/cartoes/autoriza', {cartao: info}, function() {
    console.log('consumindo serviço de cartões');
});
```

Cada uma das funções para invocação recebe sempre como parâmetro a uri do serviço desejado e, possivelmente outros parâmetros também. No exemplo àcima, foram passados também um objeto de **dados** que no nosso casoserá a representação dos dados do cartão e uma função de *callback*, onde pode ser definido o que será feito no código ao final dessa execução.

Claro que para tornar esse comportamento reaproveitável para ser utilizado pelos arquivos e classes que consomem a `CartoesClient.js` é preciso que esses parâmetros sejam definidos como variáveis e a possibilidade de executar um POST seja isolada em uma função específica:

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

function CartoesClient() {
  this._client = restify.createJsonClient({
    url: 'http://localhost:3000',
    version: '~1.0'
  });
}

CartoesClient.prototype.autoriza = function(cartao, callback) {
  this._client.post('/cartoes/autoriza', cartao, callback);
}

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

No exemplo, àcima utilizamos a mesma estratégia já conhecida de criar um construtor, onde foi executada a construção do cliente e seu retorno ficou guardado no atributo privado `_client` .

Esse construtor foi exposto a partir do `module.exports` dessa classe e por fim foi criado um método especialista em fazer POST no serviço de autorização do cartão de crédito, chamado `autoriza`, que recebe como parâmetros os dados do cartão e uma função de *callback*.

Agora ficou simples consumir este serviço a partir do *controller* de `pagamentos.js`. A primeira parte é receber uma nova instância do serviço:

```
...
  var cartoesClient = new app.servicos.CartoesClient();
...
```

Veja que a instância também foi feita à semelhança do `PagamentosDao`, identificando o arquivo através do seu caminho completo. Para que isso seja possível, já sabemos que é preciso que express-load conheça essa nova pasta que foi criada.

```
...
load('controllers')
  .then('infra')
  .then('servicos')
  .into(app);
return app;
...
```

Agora sim é só invocar o método `autoriza`:

```
...

  var cartoesClient = new app.servicos.CartoesClient();

  cartoesClient.autoriza(body['cartao'], function (err, request, response, retorno) {
    if (err){
      console.log("Erro ao consultar serviço de cartões.");
      res.status(400).send(err);
      return;
    }

    console.log('Retorno do servico de cartoes: %j', retorno);
```

Veja que a função passada como *calback* recebe 4 parâmetros: err, request, response e retorno.

```
cartoesClient.autoriza(body['cartao'], function (err, request, response, retorno)
```

Esse formato é definido pelo **restify**. Os nomes dos parâmetros são de livre escolha, mas sua ordem é muito importante:

- **err**: armazena um possível erro na execução e como de praxe no mundo Node.js é o primeiro parâmetro da função de callback, para assegurar que o erro, caso ocorra, seja sempre identificado.
- **resquest** e **response**: semelhante aos parâmetros recebidos pelas rotas do express, eles representam a requisição e resposta da chamada do serviço REST, já que ela nada mais é do que uma chamada HTTP. Um detalhe importante é tomar cuidado para que eles não fiquem com os mesmos nomes utilizados para identificar a req e res da rota onde estão inseridos, pois isso pode causar conflitos e comportamentos inesperados.

- **retorno**: variável que armazena o retorno da requisição, ou seja, o que o serviço de cartões enviou ao final de sua execução. Com esse informação em mãos, pode-se usar para informar ao cliente qual o status atual da solicitação do lado do serviço de cartões, por exemplo.

O **restify** simplifica bastante a vida para consumir serviços REST, mas lembre sempre que serviços REST nada mais são do que integração via HTTP, então qualquer chamada via request HTTP normal também deveria funcionar normalmente.

Web Services SOAP

A maneira de integração mais difundida hoje em dia está no uso de **Web Services**. Disso já sabemos. E existem várias maneiras de se implementar um Web Service, mas apesar de ser um termo genérico, existe algo muito bem especificado pela W3C:

Um dos quesitos primordiais durante a elaboração dessa especificação era que precisaríamos aproveitar toda a plataforma, arquitetura e protocolos já existentes a fim de minimizar o impacto de integrar sistemas. Criar um novo protocolo do zero era fora de cogitação .

Por esses motivos o Web Service do W3C é baseado em HTTP e XML, duas tecnologias onipresentes e que a maioria das linguagens sabe trabalhar muito bem.

Apesar de XML não ser uma tecnologia tão presente assim no mundo Node.js, existem muitos sistemas que utilizam desse recurso e que expõem Web Services SOAP, portanto o PayFast precisa estar preparado para realizar essa integração caso seja necessário.

Implementando o cliente SOAP

Como de costume, existe um módulo desenvolvido pela comunidade que abstrai grane parte da dificuldade em trabalhar com SOAP, XML e seus detalhes mais profundos. O módulo recebeu o singelo nome de **soap** e sua instalação no projeto é feita da forma que já conhecemos:

```
npm install --save soap
```

A utilização do pacote **soap** é muito semelhante à do **restify** e praticamente tão simples quanto. O primeiro passo é obviamente carregar o pacote e, em seguida, criar o clinte:

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

soap.createClient(url_do_wsdl, function(err, client) {
        ...
});
```

No momento da criação do cliente, deve ser informada a url do **WSDL** do Web Service SOAP e uma função de *callback*. Essa função de *callback* recebe como parâmetro uma variável para armazenar possíveis erros e um objeto que armazenará o retorno da criação do client.

Este é um ponto muito interessante da lib **soap** do Node, pois como a criação do cliente é feita a partir da leitura do WSDL, ou seja, do descritor dos serviços, o cliente retornado é um objeto que já possui disponíveis para uso funções que tem exatamente o mesmo nome dos serviços expostos pelo WebService.

O serviço que iremos consumir é o WebService dos Correios que possui um serviço chamado `CalcPrazo`, que serve para calcular prazos de entrega. Isso significa que o **client** dentro da função de *callback* é capaz de invocar diretamente uma função chamada `CalcPrazo` passando os parâmetros que o serviço espera e este será invocado normalmente:

var soap = require('soap');

function CorreiosSOAPClient(url) { this._url = url; }

CorreiosSOAPClient.prototype.calculaPrazo = function(args, callback) { soap.createClient(this._url, function(err, client) { client.CalcPrazo(args, callback); }); }

Os parâmetros que o serviço espera e que no exemplo àcima chamamos de **args** podem ser passados em formato json e é possível também passar uma função de *callback* para tratar o retorno, como de praxe.

O legal dessa lib é exatamente isso! Ela abstrai totalmente as conversões para XML e permite que a gente trabalhe somente com JSON. Todas as conversões necessárias são realizadas de maneira encapsulada e nem precisamos ficar sabendo como elas acontecem!

Por fim, seria legal isolar estes comportamentos numa classe especialista em consumir Web Services SOAP dos correios e deixá-la na pasta reservada para serviços.

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

function CorreiosSOAPClient(url) {
  this._url = url;
}

CorreiosSOAPClient.prototype.calculaPrazo = function(args, callback) {
  soap.createClient(this._url, function(err, client) {
        client.CalcPrazo(args, callback);
  });
}

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

A estrutura também é semelhante às anteriores. Com um construtor exposto pelo `module.exports` e métodos expostos no `prototype`.

Mais uma vez, toda esta implementação poderia ser feita com requests HTTP e conversões de dados JSON em XML feitos manualmente, mas a lib ajuda a ganhar um tempo monstruoso no desenvolvimento encapsulando esses requisitos de infra-estrutura e expondo interfaces de uso bem simples.