Navegando entre recursos com REST

Entendendo o HATEOAS e suas aplicações

Por que este artigo é útil:

Neste artigo serão apresentados os conceitos de REST dando uma ênfase especial em HATEOAS, uma abordagem utilizada para enriquecer a representação de recursos com links que permitem navegar entre recursos e mudar os estados da aplicação. Também será feita uma revisão das técnicas utilizadas ao longo dos anos para fazer a integração de sistemas. Para exemplificar os conceitos abordados será apresentada uma aplicação de compras online construída com o Spring HATEOAS, uma implementação de HATEOAS feita pela Spring Foundation.

A integração de sistemas é um dos grandes desafios para muitas empresas e seus desenvolvedores. Diversas razões fazem com que essa tarefa seja complexa, tais como: envolver aplicações construídas por equipes diferentes, na maioria das vezes em empresas diferentes e em certos casos com culturas diferentes. Além disso, existem questões técnicas envolvidas, como por exemplo: segurança, disponibilidade, desempenho, etc. Mas apesar desse cenário, as empresas se preocupam cada vez mais com integração para obter benefícios do tipo: reaproveitar funcionalidades já disponíveis nos sistemas que estão em produção e reduzir custos evitando que as informações sejam replicadas, levando à inconsistência e outros erros.

Atualmente, existem diversas formas de se fazer integração, mas a que talvez seja a mais conveniente se baseia no uso de serviço de software. De uma forma geral, um serviço consiste em basicamente disponibilizar uma funcionalidade implementada em um sistema para ser acessada por outros sistemas através de determinados protocolos e representações. Dessa forma, sempre que um sistema precisar de alguma informação mantida por outro sistema, basta que ele faça a chamada de um serviço no sistema apropriado e a resposta adequada será devolvida.

Atualmente, a WEB tem sido muito utilizada como meio de comunicação para fazer integrações. Isso se deve basicamente às próprias características da Web, tais como, ubiquidade e escalabilidade, além dela utilizar o protocolo HTTP que é muito simples e ao mesmo tempo versátil o suficiente para atender às demandas das empresas. Por trás de todos os conceitos da Web está o que se chama de estilo arquitetural, um conjunto de propriedades que definem as suas características. Nesse caso, estamos falando do REST (Representa tive State Transfer), um estilo apresentado por Roy Fielding na sua tese de doutorado em 2000. Apesar desse conceito ter ficado adormecido por alguns anos, ele é atualmente um dos principais estilos aplicado em sistemas para fazer integrações. O REST e suas propriedades são os principais assuntos desse artigo , como será visto nas seções seguintes.

Nesse artigo, serão apresentados os principais conceitos de REST e como ele pode ser utilizado para disponibilizar os serviços de uma aplicação. Será apresentado também o HATEOAS, uma das principais propriedades do REST para fazer a navegação entre recursos, que são representações dos modelos de negócio da aplicação e também para modificar o estado da aplicação. Ao final do artigo será apresentada uma aplicação de exemplo para demonstrar os conceitos apresentados, construída com o Spring HATEOS, um projeto do Spring Foundation. Para encerrar, será citado o HAL, *Hypermedia Application Language*, que define um padrão bem simples para a estrutura de representação de recursos e links do HATEOAS.

Evolução das tecnologias para integração de sistemas

A necessidade de integração entre sistemas não é nova. Durante muito tempo os desenvolvedores vêm buscando soluções para tratar desse problema, e nessa busca surgiram diversas abordagens, que vão desde integrações mais simples baseadas na troca de arquivos ou no uso de banco de dados centralizados e compartilhados por diversos sistemas, até as mais sofisticadas baseadas em camadas de serviço nas aplicações ou em ferramentas próprias de integração. Apesar de ainda serem muito usadas, essas abordagens simples são muito limitadas e trazem diversos problemas pois elas geralmente não são padronizadas, são de difícil manutenção, não são escaláveis e podem ser muito acopladas, como no caso de integração via banco de dados. Elas certamente não são adequadas para atender às demandas das aplicações mais modernas, devido à complexidade dessas aplicações e da necessidade de haver um grande número de sistemas interligados.

Considerando a forma de integração baseada em serviços, uma tecnologia importante que surgiu em 1976 foi o RPC (*Remote Procedure Call* – veja a referência *A High-Level Framework for Network-Based Resource Sharing* na seção **Links**). A ideia inicial do RPC era definir uma forma de compartilhar diversos recursos em uma rede, em particular, permitir o acesso remoto ao sistema de arquivos, mas ideia de RPC acabou se expandindo e tornando-se uma abordagem para executar funcionalidades diversas em sistemas remotos (o meio de acesso é feito através da interface de rede, tornando assim indiferente se o processo que estamos chamando está na mesma máquina ou em uma máquina remota), ou seja, uma forma de integrar sistemas. Algumas implementações importantes de RPC são: CORBA, DCom+, RMI, SOAP, entre outras. Apesar de funcionais, essas implementações possuem algumas limitações. DCom+ e RMI, por exemplo, são específicas de plataforma. DCom+ só permite a comunicação entre aplicações que utilizam tecnologia Microsoft. Já o RMI permite somente a comunicação entre aplicações desenvolvidas em Java. Por outro lado, CORBA possibilita a integração entre serviços implementados em diversas linguagens e que podem ser acessar através de um sistema intermediário, um middleware, chamado ORB (Object Request Broker), que é responsável por receber as requisições dos clientes, chamar o método do objeto destino e devolver o resultado. Apesar do CORBA ser muito flexível e poderoso, a dependência do middleware faz com que sua configuração e disponibilização se torne algo complexo para se desenvolver e manter. Posteriormente surgiu uma versão de RMI que era executada em cima do CORBA, chamado de RMI-IIOP, e que estava disponível em alguns servidores de aplicação, tal como o JBoss, que já vinha com uma versão específica de ORB, o JacORB (veja a seção **Links**).

Devido às dificuldades citadas anteriormente e, principalmente, pela necessidade de ambientes muito heterogêneos precisarem se comunicar, as implementações de RPC ainda não se mostravam adequadas e, na tentativa de superar essas dificuldades, a Microsoft desenvolveu em 1998 o SOAP, que inicialmente significava *Simple Object Access Protocol*, mas que com o tempo passou a ser mantido pela W3C, deixou de ser um acrônimo (veja a nota no final da Seção 1 da especificação [*SOAP Version 1.2 Part 1: Messaging Framework - Second Edition*](http://www.w3.org/TR/soap12-part1/#intro) a partir do endereço indicado na seção **Links**). O poder do SOAP estava no fato dele ser um protocolo relativamente simples em relação ao que existia até então, leve, baseado em XML, e que poderia utilizar qualquer meio de transporte (TCP, UDP, HTTP, SNMP, etc.) para fazer a comunicação entre sistemas, possibilitando a integração entre ambientes extremamente diferentes e até então totalmente isolados. E junto do SOAP surgiu uma série de outros serviços que definem um conjunto de características e formatos de comunicação entre aplicações, formando o que conhecemos como Web Services. Com o crescimento da web, essa abordagem se tornou uma das principais formas de integração entre sistemas até os dias de hoje.

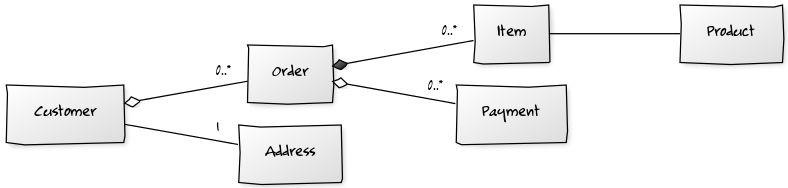
Apesar da grande evolução trazida pelo SOAP, alguns pontos ainda deveriam ser considerados, tais como o fato de ser um protocolo novo, que faz uso massivo de XML, necessitando de parses e transformadores de mensagens e que acabam tornando o processo de comunicação bem custoso.

Apesar de não estar diretamente relacionado à integração sistemas, Roy Fielding apresentou em 2000 na sua tese de doutorado o conceito de REST (REpresentative State Transfer), um estilo arquitetural que define as propriedades da própria web e é capaz de resolver a maioria dos problemas que as abordagens baseadas em RPC também tratam, porém, de uma forma bem mais simples e baseada em características do próprio protocolo HTTP. Na seção “REST: o estilo da Web”, vamos falar mais sobre o REST e suas propriedades. Mas antes, descreveremos brevemente a aplicação de uma loja online que será utilizada como exemplo no decorrer desse artigo.

A Macaxeira Online

A Macaxeira Online é uma loja que vende produtos pela Internet. Ela possui uma lista diversificada de produtos e muitos clientes. De forma simplificada, o cliente entra na loja, seleciona o que quer comprar, informa como será feito o pagamento e conclui a operação de compra. A partir desse momento é gerado um pedido que fica no estado de Aguardando Pagamento até que seja feita a confirmação do pagamento. Após essa confirmação, os produtos escolhidos são enviados para o endereço atual do cliente.

Com o crescimento da loja, vários sistemas internos deverão ser criados para dar suporte às operações. Visto que a empresa estima crescer muito nos próximos anos, foi feito um estudo e decidiram que a integração entre esses sistemas deveria ser feita na camada de aplicação, utilizando serviços no estilo REST. Basicamente, os serviços permitiriam o acesso e manipulação dos dados do sistema da loja. A **Figura 1** mostra o modelo de classes de negócio da loja Macaxeira Online.



**Figura 1.** Modelo de negócio da loja Macaxeira Online.

Esse exemplo da Macaxeira Online será usado no decorrer desse artigo para demonstrar os conceitos apresentados e também para mostrar uma solução para um problema de integração.

REST: o estilo da Web

É fato que a web está cada vez mais presente em nossas vidas. Tão fundamental para alguns tipos de serviço, que arriscamos dizer que se ela ficasse inacessível por somente uma dia já seria suficiente para causar um colapso. Apesar da sua importância, dificilmente pensamos em como é possível que a Web esteja presente em praticamente todos os lugares ao mesmo tempo, como ela se expande ou ainda, o que torna o seu acesso tão fácil a ponto de precisarmos somente abrir o browser e digitar uma URL.

O fato é que Roy Fielding já pensava nessas propriedades quando participou da definição do protocolo HTTP/1.1 (veja a seção **Links**) em 1999 e deixou isso registrado na sua tese *Architectural Styles and the Design of Network-based Software Architectures*, defendida no ano de 2000, e que apresenta o conceito de REST. Segundo Fielding, REST é o conjunto de propriedades que definem a Web. Essas propriedades são [1]:

* **Recursos endereçáveis**. De forma simplificada, um recurso é um objeto, que pode ser uma pessoa, um produto, um endereço ou uma página web [2]. Segundo o conceito de REST, esse recurso deve possuir um identificador de recurso único, URI (Unique Resource Identifier), usado para referenciá-lo. Se pensarmos no exemplo da Macaxeira Online, poderíamos definir Pedido, Pagamento, Produto e Cliente como exemplos de recursos. Esses recursos devem estar acessíveis através de URLs, que são exemplos de URIs, para executar operações de consulta, inclusão, alteração, etc. Por exemplo, poderíamos usar as seguintes URLs para fazer referência a alguns dos recursos citados:
  + *http://macaxeiraonline.com/orders*, para fazer referência à lista de pedidos;
  + *http://macaxeiraonline.com/orders/1234*, para fazer referência ao pedido 1234;
  + *http://macaxeiraonline.com/orders/1234/items*, para fazer referência aos itens do pedido 1234;
  + *http://macaxeiraonline.com/orders/12/customer*, para fazer referência ao cliente do pedido 12.

Repare que foi utilizado um padrão para identificar o recurso de interesse. Por exemplo, */orders*, para identificar todos os pedidos, */orders/{id}*, para representar o pedido de número definido por *{id}*, */orders/{id}/items*, para fazer referência aos itens de um pedido em questão, e assim por diante. De uma forma geral, representamos qualquer recurso da seguinte forma: */<nome do recurso>/{id}/<nome do sub-recurso>*.

Note que esse padrão para identificar os recursos não é único. Também é comum se encontrar uma formato em que a primeira parte da URL identificando o recurso define o recurso manipulado, por exemplo, */customer/order/12,* poderia ser utilizado para acessar os dados do cliente do pedido 12, ao invés de */orders/12/customer*.

* **Conjunto restrito de operações**. Uma vez que foram definidos os recursos, é necessário haver uma forma de manipulá-los. A versão 1.0 do protocolo HTTP (veja a referência “Protocolo HTTP/1.0” na seção **Links**) já trazia diversos métodos que permitiam manipular recursos, então, do ponto de vista do REST, por que não padronizar esses mesmos métodos para realizar as operações necessárias? Assim, os seguintes métodos HTTP se tornaram padrão para manipular recursos:
  + **GET:** utilizado para consultar informações de um recurso. Dizemos que o GET é idempotente e seguro. Idempotente significa que toda vez que fizermos uma mesma consulta com GET devemos obter o mesmo resultado. Seguro significa que a consulta não deve fazer modificações no recurso consultado;
  + **POST:** cria um novo recurso baseado em uma representação informada. POST não é idempotente e nem seguro, pois ele altera o recurso e toda vez que é chamado ele deve devolver algo diferente. Normalmente, o POST devolve uma informação que permita acessar o recurso recém-criado;
  + **PUT:** permite fazer a alteração do recurso. O PUT é idempotente e não seguro. Ele é idempotente por que uma requisição com um mesmo conteúdo deve gerar o mesmo resultado. PUT também pode ser usado para criar um recurso, de forma semelhante ao POST;
  + **DELETE:** destrói a representação de um recurso. Se for feita uma consulta ao recurso após um DELETE, deverá ser devolvido um código de erro HTTP 404, informando que o recurso não foi encontrado. DELETE é idempotente e não seguro;
  + **HEAD:** devolve somente os cabeçalhos associados ao recurso e códigos de erro. Ele pode ser utilizado para obter certas informações do recurso, como por exemplo, se o recurso teve seu tamanho alterado, a data da sua última modificação, entre outros;
  + **OPTIONS:** utilizado para verificar se um recurso está acessível ou não e os métodos permitidos para acessar o recurso. Por exemplo, se o recurso em questão existe e está acessível, deverá ser devolvido uma resposta do tipo:

200 OK

Alo: HEAD,GET,PUT,DELETE,OPTIONS

* + **PATCH:** permite fazer a alteração parcial de um recurso. O método PATCH surgiu somente em 2010 com a aprovação da RFC 5789 (veja a seção **Links**). O PATCH é idempotente e não seguro.

**Nota:** É importante perceber que quando um serviço é chamado via SOAP ou CORBA, a requisição enviada par executá-lo contém o nome de uma operação específica que será executada, por exemplo, um serviço para consultar os dados de um determinado cliente pode ter o nome consultaCliente, que recebe como parâmetro o Id desse cliente. No caso do REST, isso é diferente, pois as operações são padrões (GET, PUT, POST, etc.) e elas sempre serão executadas em relação a um recurso, que nesse caso poderia ser o recurso que se encontra no local /costões.

Ao se pensar nesse fato de haver um conjunto restrito de operações, o leitor pode se perguntar se isso não pode gerar algum tipo de limitação na construção dos serviços de integração. Porém, é fácil fazer uma analogia com os bancos de dados que utilizam sentenças SQL para executar a suas operações que, da mesma forma, define um conjunto restrito de operações: SELECT, INSERT, UPDATE e DELETE para fazer tudo o que for necessário e mesmo assim as possiblidades parecem ser infinitas.

* **Orientado a representação**. Para acessar um recurso é necessário que seja feita a troca de informações entre um cliente e um servidor. Via de regra, o cliente deve especificar os tipos de representação que ele é capaz de interpretar, ou então, qual a representação de sua preferência e o servidor deve gerar uma resposta de acordo. No caso das requisições que utilizam o protocolo HTTP, um atributo **Acep.** enviado no cabeçalho da mensagem é usado para informar a preferência do cliente a respeito da representação e o **Content-type** informa o tipo que realmente está sendo recebido tanto do lado cliente quanto do lado servidor. Esse atributo usa o formato conhecido por Multipurpose Internet Mail Extension, ou simplesmente *MIME type*. A informação trazida pelo MIME é fundamental nesse tipo de comunicação pois um determinado recurso pode ter diversas representações e sem ele não seria possível fazer a interpretação do conteúdo devolvido no corpo da mensagem. Uma das necessidades de haver múltiplas representações é que o estado de um mesmo recurso pode ser acessado por plataformas diferentes, por exemplo, se for um browser que estiver fazendo a requisição, é conveniente que a representação seja em HTML, caso seja uma chamada AJAX, uma representação conveniente seria JSON, e assim por diante.

Para exemplificar esses conceitos de conjunto restrito de operações e as representações de estado de um recursos, suponha que deverá feita uma consulta do pedido 1234 na Macaxeira Online e o conteúdo devolvido deverá ser no formato JSON. Para isso, será utilizada a URL onde se encontra esse recurso, no caso *http://macaxeiraonline.com/orders/1234*. A seguinte requisição pode ser feita para obter o resultado desejado:

GET /orders/1234 HTTP/1.1

Host: macaxeiraonline.com

Accept: application/json

Na primeira linha da requisição é informado o método GET para informar que o recurso que se encontra em */orders/1234* será consultado. Além disso, também é informado que o protocolo da requisição é *HTTP/1.1*. Na segunda linha é informado o servidor para o qual a requisição foi encaminhada. Na terceira linha aparece o cabeçalho *Accept*, informando que o cliente da requisição espera receber um conteúdo no formato JSON. Caso não ocorram erros na solicitação dessa requisição, a resposta apresenta na **Listagem 1** será devolvida.

**Listagem 1.** Resposta da nossa requisição exemplo.

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: application/json;charset=UTF-8

Date: Tue, 20 May 2014 00:55:55 GMT

{

"orderId":1234,

"createdAt":1400546614231,

"status":"WAITING\_FOR\_PAYMENT",

"value":315.27

}

Essa resposta mostra o status *200* da requisição, informando que teve sucesso no processo da requisição, o nome do servidor que atendeu à requisição na segunda linha, o *Content-type* informando que o conteúdo no corpo da mensagem está no formato JSON e por fim, a data ocorreu a resposta.

* **Comunicação sem estado (stateless)**. Uma comunicação sem estado implica que toda requisição é tratada de forma isolada e, portanto, precisa conter todos os dados necessários para concluir a requisição. Isso significa que o protocolo em si não mantém informações de sessão e, caso necessário, o cliente deve prover algum mecanismo para fazer esse tratamento. Uma das maiores vantagens de se operar dessa forma é a escalabilidade, pois caso seja necessário mais processamento, basta adicionar mais máquinas. Uma forma simples e eficiente;
* **Hipermídia como mecanismo de transição de estados da aplicação (HATEOAS - Hypermedia as The Engine of Application State).** Essa propriedade basicamente indica que toda representação de um recurso deve possuir links que permitem fazer a mudança de estado da aplicação. Por exemplo, um pedido pode conter links para efetuar o seu cancelamento, para listar os seus itens, os seus pagamentos, entre outros. Devido à importância desse item, vamos discuti-lo em mais detalhes na seção “HATEOAS: navegando entre recursos”.

As propriedades supracitadas caracterizam o estilo arquitetural REST, sendo a própria Web um exemplo de aplicação desse estilo. Assim, sempre que falamos que uma aplicação segue o estilo REST, devemos subentender que essas propriedades estão presentes nessa aplicação e, consequentemente, que ela tem “o estilo da Web”.

HATEOAS: navegando entre recursos

Hipermídia é um conceito que surgiu em 1965, definido por Ted Nelson [3], mas que ficou mundialmente conhecido apenas com o surgimento da Web. Basicamente, hipermídia é um meio de transmissão de conteúdo não linear, envolvendo diversos tipos de mídia, tais como texto, imagem, vídeo, etc. A Web é o melhor exemplo de hipermídia, onde o conteúdo é apresentado no formato de páginas e com navegação feita através de links.

Em uma aplicação hipermídia, os estados da aplicação são apresentados através de representações dos recursos acessados via URIs. Por exemplo, para acessar os dados do cliente 10, pode ser utilizado */customers/10*. Quando isso é feito, o que se obtém são as informações atuais do cliente 10 naquele momento, ou seja, o seu estado atual. Caso esse cliente tenha alguma informação atualizada ele terá um estado diferente daquele obtido na consulta anterior.

Além do estado do recurso, a representação pode conter informações que permitem navegar para outros recursos associados ao recurso em questão e também executar operações que pode causar a mudança de estado do próprio recurso. Por exemplo, um pedido contém itens. Ao se fazer a consulta de um determinado pedido, o resultado obtido pode trazer links que permitem acessar a lista de itens. Além disso, um pedido pode ser cancelado e, dessa forma, o resultado da consulta de pedidos também pode trazer um link para executar essa operação. A **Listagem 2** mostra a representação JSON para o pedido 1234. Nessa listagem são apresentados os atributos do pedido *orderId*, *createdAt*, *status* e *value* e uma lista de links sendo o primeiro link uma referência para o próprio recurso, o segundo link que pode ser utilizado para cancelar o pedido atual e o terceiro link para acessar os itens de pedido.

**Listagem 2.** Representação JSON do pedido 1234.

{

"orderId": 1234,

"createdAt": 1400795943914,

"status": "WAITING\_FOR\_PAYMENT",

"value": 95.5,

"links": [

{

"rel": "self",

"href": "http://macaxeiraonline.com/orders/1234"

},

{

"rel": "Order Cancelation",

"href": "http://macaxeiraonline.com/orders/1234/cancel"

},

{

"rel": "Items",

"href": "http://macaxeiraonline.com/orders/1234/items"

}

]

}

Essa abordagem de trazer na representação do recurso os links que permitem a mudança de estados da aplicação é o que define o HATEOAS. Ao utilizarmos essa abordagem nos nossos serviços, obtemos os seguintes benefícios:

* **Transparência na Localização:** Cada link devolvido na representação do recurso deve ser acompanhado de um nome lógico, um *alias*, que serve para identificar o novo estado que a aplicação assumirá ao seguir por aquele link. Dessa forma, é possível fazer a navegação entre os estados sem a necessidade de saber de antemão a localização dos recursos. Esse conceito é análogo ao JNDI (*Java Name and Directory Interface*), utilizado em aplicações Java EE, que permite acessar EJBs sem que se conheça a localização exata deles.

Os nomes lógicos podem ser definidos conforme a necessidade de cada aplicação e podem ter significados variados. Via de regra, é interessante que eles tenham significados relacionados ao domínio de negócio da aplicação, como por exemplo: Itens do pedido, Pagamentos do Pedido, etc.;

* **Detalhes de interações desacoplados:** Quando um recurso é definido, deve-se garantir que certos detalhes irrelevantes para o cliente a respeito da interação não estejam expostos. Por exemplo, suponha que seja necessário acessar a lista de pedidos do cliente 48923849. A URL para acessar esse recurso seria: *http://macaxeiraonline.com/customer/48923849/orders*. Mas note que pode haver um problema com essa requisição. Caso esse cliente tenha muitos pedidos, essa requisição irá demorar muito para ser atendida e muitos pedidos precisarão ser devolvidos como resposta. Nessa situação, seria melhor se o serviço devolvesse uma lista paginada de pedidos contendo, por exemplo, no máximo cinco pedidos por página. Isso resolveria o problema de trafegar muitos dados, porém, seria necessário prover um mecanismo para fazer a navegação entre as páginas. Ao invés de delegarmos essa função para o cliente, o serviço poderia incluir na lista de links as URLs para as páginas anterior e posterior, que quando consumidos, poderiam gerar a informação da nova página. Por exemplo, a segunda página de resultados da lista de pedidos poderia ter o conteúdo indicado na **Listagem 3**.

**Listagem 3.** Exemplo de segunda página de resultados da lista de pedidos.

{

orders: [

{

id: 12,

...

links:[ ... ]

},

{

id: 1094,

...

links:[ ... ]

},

{

id: 438,

...

links:[ ... ]

},

{

id: 1829,

...

links:[ ... ]

},

{

id: 34,

...

links:[ ... ]

}

],

links:[

{

rel: "next",

link: "http://macaxeiraonline.com/customer/48923849&start=11&size=5"

},

{

rel: "previous",

link: "http://macaxeiraonline.com/customer/48923849&start=1&size=5"

}]

}

Note que os links *next* e *previous* possuem parâmetros para informar o primeiro elemento da página em questão (*start*) e a quantidade de elementos por página (*size*). Dessa forma, o cliente não precisa armazenar qual a página atual ou mesmo fazer contas para descobrir qual o primeiro elemento das páginas anterior e posterior.

* **Redução dos Erros nas Transições de Estado:** Baseado no HATEOAS, as representações dos recursos deveriam trazer as informações necessárias para fazer a mudanças de seu estado. Dessa forma, a aplicação que acessa o recurso não precisa ter conhecimento prévio a respeito das URLs envolvidas para executar as operações, somente saber qual o alias que é usado para identificar a URL. Fazendo isso, a aplicação que disponibiliza os serviços tem um controle maior sobre quais operações podem ser executadas nos recursos e nos estados corretos. Por exemplo, suponha que você efetuou uma compra na Macaxeira Online mas após um tempo você decide efetuar o cancelamento dessa compra. Se for feita a consulta desse pedido antes de efetuar o cancelamento, a sua representação deve trazer o link para o cancelamento. Mas após ter feito o cancelamento, não será possível mais executar essa operação e consequentemente, isso deverá ser refletido na representação, omitindo o link para essa operação. A **Listagem 4** traz uma representação de exemplo de um pedido que pode ser cancelado. Note que o link de cancelamento aparece com o alias “cancele”.

**Listagem 4.** Representação de um pedido com link para cancelamento.

{

id: 74,

status: AGUARDANDO\_PAGAMENTO,

price: 250.00,

created\_at: 1289731982,

links:[

{

rel: "cancel",

link: "http://macaxeiraonline.com/orders/74/cancel"

},

...

]

}

Concluída a fundamentação do nosso estudo, na próxima seção mostraremos na prática o Spring HATEOAS, uma implementação de HATEOAS feito pela Spring Foundation, e descreveremos em detalhes como ele pode ser usado para resolver o problema da Macaxeira Online.

[CHECKPOINT]

RESTful API com Spring HATEOAS

Em 2013 a Spring Foundation lançou o projeto Spring HATEOAS com o objetivo de facilitar a criação de APIs RESTful em aplicações que já utilizam o Spring MVC. Entre as funcionalidades disponibilizadas estão: classes para ajudar na criação de representações de recursos, classes para auxiliar na criação de links, além de helpers para ajudar a descobrir os links em recursos HATEOAS.

A seguir será apresentado como aplicar os conceitos abordados até o momento no projeto do Macaxeira Online. A API da loja será mostrada, como torná-la aderente ao HATEOAS, além de detalhar a utilização do Spring HATEOAS. O modelo apresentado na **Figura 1** descreve o relacionamento das classes de negócio utilizadas na loja virtual e que será usado como base para a implementação.

O primeiro passo da implementação consiste em mapear os recursos, para isso o framework oferece a classe **ResourceSupport**. Ela dá suporte para a criação dos links entre recursos, além de ajudar a definir os atributos da classe de negócio que serão expostos. A classe **Order** será usada para exemplificar, como apresentado na **Listagem 1**.

**Listagem 1**. Classe de negócio representando um pedido.

**public class** Order **implements** Serializable {

**private** Long id;

**private** Customer customer;

**private** Set<Payment> payments;

**private** Set<Item> items;

**private** Date createdAt = new Date();

**private** OrderStatus status = OrderStatus.WAITING\_FOR\_PAYMENT;

**private** BigDecimal value = BigDecimal.ZERO;

**public** Order() {}

...

}

A **Listagem 2** mostra a classe **OrderResource,** que estende de **ResourceSupport** e define os atributos da entidade **Order** que serão usados na representação do recurso. Foi acrescentado o método **copyAttributesFrom()** para ajudar no processo de cópia dos valores de **Order** para **OrderResource.**

**Listagem 2**. Classe de recurso representando um produto.

**public** **class** OrderResource **extends** ResourceSupport {

**private** Long orderId;

**private** Date createdAt;

**private** String status;

**private** BigDecimal value;

...

**public** **void** copyAttributesFrom(Order order) {

**this**.orderId = order.getId();

**this**.createdAt = order.getCreatedAt();

**this**.status = order.getStatus().toString();

**this**.value = order.getValue();

}

}

Um mesmo recurso pode ser requisitado em diversos pontos distintos na aplicação, tornando o seu processo de *build* repetitivo. Imagine os seguintes cenários: a consulta de um determinado pedido e a consulta de todos os pedidos de um determinado cliente. Em ambos os cenários é necessário fazer a criação do recurso. Uma boa abordagem é centralizar esse processo em um único ponto. A classe **ResourceAssemblerSupport** é utilizada para esse propósito. Para isso, é necessário estendê-la e implementar o método **toResource(...)**,responsável por devolver uma instância do recurso. Caso seja necessário criar uma lista de recursos, a classe sobrecarrega o método **toResource()** para receber como parâmetro lista. Um exemplo prático do *build* pode ser visto na **Listagem 3**. Nessa listagem é apresentada implementação da classe **OrderResourceAssembler**. No método **toResource()** é construído o objeto representando o recurso e também são definidos alguns links que permitem a navegação entre outros recursos.

**Listagem 3**. Assembler de pedido (Order).

@Componente

**public** **class** OrderResourceAssembler **extends** ResourceAssemblerSupport<Order, OrderResource> {

@Autowired

**private** ItemResourceAssembler itemRessourceAssembler;

**public** OrderResourceAssembler() {

**super**(OrderController.class, OrderResource.class);

}

@Override

**public** OrderResource toResource(Order order) {

OrderResource orderResource = createResourceWithId(order.getId(), order);

orderResource.copyAttributesFrom(order);

...

orderResource.add(linkTo(methodOn(OrderController.class)

.items(order.getId())).withRel("Items"));

**return** orderResource;

}

}

No exemplo do *controller* na **Listagem 4**, pode ser visto é esse processo da criação do recurso que será exposto. Imagine que um usuário pesquisou pelo pedido ***1234*** através da seguinte URL [*http://macaxeiraonline.com/orders/1234*](http://macaxeiraonline.com/orders/1234). Ao efetuar a requisição, o *controller* direciona para o método *order(...)*, executando uma chamada para o serviço, que recupera o pedido e o encaminha para o **OrderResourceAssembler** que se encarrega em transformar o pedido no recurso do tipo OrderResource. A **Listagem 5** mostra como seria uma representação desse recurso no formato JSON.

**Listagem 4**. Controller utilizado para devolver a representação de um pedido.

@RestController

@RequestMapping("/orders")

**public** **class** OrderController {

@Autowired

**private** OrderService orderService;

@Autowired

**private** OrderResourceAssembler orderResourceAssembler;

@RequestMapping("/{orderId}")

**public** OrderResource order(@PathVariable Long orderId) {

Order order = orderService.get(orderId);

**return** orderResourceAssembler.toResource(order);

}

...

@RequestMapping(method = RequestMethod.GET)

**public** List<OrderResource> getOrders() {

List<Order> orders = orderService.getAll();

**return** orderResourceAssembler.toResources(orders);

}

...

}

**Listagem 5.** Exemplo de JSON retornado.

{

"createdAt": 1400638921610,

"links": [ ... ],

"orderId": 1234,

"status": "WAITING\_FOR\_PAYMENT",

"value": 95.5

}

Quando se fala em HATEOAS, os links passam a desempenhar um papel importante, visto que são os responsáveis por garantir a transição de estado da aplicação. O Spring HATEOAS oferece mais de um mecanismo para a sua criação. A seguir serão apresentadas algumas formas para criar esses links.

A forma mais simples de se criar um link é através da própria class **Link**, passando em seu construtor a URI que se deseja referenciar e, através do método **withRel(...)**, pode-se definir um *alias*. Uma vez que o link foi criado, pode-se então associá-lo a um determinado recurso. A classe **ResourceSupport** provê os métodos **add(Link link)**  e **add(Iterable<Link> links)**, para auxiliar nesse processo.

O problema de se adotar esta abordagem de definir os links diretamente é que o desenvolvedor precisa conhecer previamente o protocolo utilizado, *HTTP* ou *HTTPS*, o hostname da aplicação e o número da porta quando necessário, tornando os links muito sensíveis a possíveis mudanças no sistema e no ambiente.

**Listagem 6**. Construção de links para fazer a navegação entre recursos

@RequestMapping("/{orderId}")

**public** OrderResource get(@PathVariable Long orderId) {

Order order = orderService.get(orderId);

OrderResource resource = orderResourceAssembler.toResource(order);

Link selfLink = **new** Link("http://macaxeiraonline.com/orders/74");

Link custormerLink = **new** Link("http://macaxeiraonline.com/customer/83985/orders", "my orders");

resource.add(selfLink);

resource.add(customerLink);

return orderResource;

}

A classe **ControllerLinkBuilder** vem justamente para ajudar a contornar esses problemas. Ela utiliza a classe **ServletUriComponentsBuilder**, que analisa a requisição feita e recupera as informações referente ao protocolo, hostname, porta, etc.

Como o próprio nome já diz, **ControllerLinkBuilder** tem um relacionamento muito estreito com os *controllers*, permitindo a criação de links, tomando-os como base. Ela fornece o método estático **linkTo(...)** que recebe como parâmetro um *controller*, sendo necessário definir apenas o restante do *path* para compor a URI. Pode-se observar na **Listagem 7** o exemplo da construção do link para o recurso *customer* do pedido.

**Listagem 7**. Exemplo de construção de links com ControllerLinkBuilder.

**import** **static** org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;

@Component

**public** **class** OrderResourceAssembler **extends** ResourceAssemblerSupport<Order, OrderResource> {

@Override

**public** OrderResource toResource(Order order) {

OrderResource orderResource = createResourceWithId(order.getId(), order);

Link customerLink = linkTo(OrderController.class).slash(order.getId())

.slash("customer").withRel("Customer Detail");

orderResource.add(customerLink);

...

}

O **ControllerLinkBuilder** disponibiliza ainda outra estratégia, a criação de links apontando diretamente para métodos de um determinado *controller*. Essa estratégia é interessante quando deseja-se acessar serviços da própria API. Essa abordagem talvez seja a mais produtiva, além de ser menos suscetível a *bugs* devido à mudança de caminhos nos *controllers*. A seguir está um exemplo de como fazer essa construção.

Link customerLink = linkTo(methodOn(OrderController.class).customer(order.getId())).withRel("Customer Detail");

Vimos até aqui como montar uma aplicação REST com o Spring HATEOAS. Apresentamos como é possível a criação de recursos e links, considerando diversas abordagens para a construção dos links. A seguir, discutiremos um pouco sobre o Hypermedia Application Language, ou HAL, que é um MIME melhor estruturado e mais apropriado para representar recursos.

HAL: Um padrão para a definição de links entre recursos

Seguindo o conceito de HATEOAS, os links para os recursos devem ser disponibilizados nas respostas devolvidas pelos próprios serviços. Uma das questões que surge quando são construídos esses links é a padronização e, via de regra, cada API define o seu padrão. Dessa forma, sempre que uma aplicação precisa acessar os serviços disponibilizados por uma API, será necessário que os desenvolvedores aprendam o padrão utilizado por essa API em questão, o que pode ser trabalhoso e desagradável.

Para colaborar na solução desse problema, foi proposto em 2013 um formato que define como os links entre recursos devem ser devolvidos nas representações com JSON, chamado de HAL (*Hypermedia Application Language*). A ideia básica do HAL é definir um conjunto de convenções que permitem construir e documentar as transições entre recursos de uma API, impondo um mínimo de estrutura e de uma forma simples.

Apesar de ser um formato relativamente novo, o Spring HATEOAS já tem suporte para utilizar o HAL. Para tanto, basta especificar a anotação **@EnableHypermediaSupport(type = HypermediaType.HAL)** na classe de inicialização da aplicação para criar toda a infraestrutura necessária para a construção de links nesse formato.

A **Listagem 9** mostra como fica o JSON devolvido como resultado da chamada de */orders/1234* utilizando o formato HAL.

**Listagem 9.** Exemplo de JSON utilizando o formato HAL.

{

"\_links": {

"Customer Detail": {

"href": "http://macaxeiraonline.com/orders/1234/customer"

},

"Items": {

"href": "http://macaxeiraonline.com/orders/1234/items"

},

"Order Cancelation": {

"href": "http://macaxeiraonline.com/orders/1234/cancel"

},

"Payment Details": {

"href": "http://macaxeiraonline.com/orders/1234/payments"

},

"self": {

"href": "http://macaxeiraonline.com/orders/1234"

}

},

"createdAt": 1400724229897,

"orderId": 1234,

"status": "WAITING\_FOR\_PAYMENT",

"value": 95.5

}

Repare que os links agora ficam agrupados em “\_links”. Cada link fica agora debaixo de um objeto definido pelo alias do link e não existe mais o atributo **rel.**

A principal vantagem de se adotar um padrão como o HAL é que a navegação entre os recursos é feito de um forma mais intuitiva e padronizada. Dessa forma, fica muito mais fácil entender a função de cada link no momento da construção das aplicações que usam os serviços.

Nessa seção apresentamos apenas algumas características do HAL. Caso o leitor queira obter mais detalhes, pode ser consultada a página JSON Hypertext Application Language (veja o endereço na seção **Links**). Também é importante citar que

o HAL ainda está em draft no IETF e poderão ocorrer mudanças até seja feita a versão final especificação.

Conclusão

Apesar desse artigo ter dado uma visão abrangente de REST e das formas de integração entre aplicações, vale citar algumas abordagens específicas para se obter resultados semelhantes. O próprio Spring WEB pode ser utilizado para esse fim, pois ele conta com todo o ferramental necessário para criar representações de objetos. O fator negativo dessa abordagem é que ele por si só não traz grandes facilidades para a construção de links e nem para gerar um modelo de representações desacoplado do modelo de negócio, algo que é facilmente obtido com o Spring HATEOAS através das implementações de *ResourceSupport* e *ResourceAssemblerSupport*.

Uma outra abordagem útil consiste em utilizar o Spring Data para acessar os dados a aplicação e disponibilizá-los através de serviços utilizando o Spring Data REST. O Spring Data define uma camada de abstração responsável por acessar os dados em diferentes tecnologias, tais como: bancos de dados relacionais, Nos, entre outros. Com o Spring Data REST é possível expor as operações de CRUD através de uma camada de serviços. Esses serviços são gerados com o próprio Spring HATEOAS para construir os links de relacionamentos. Apesar de haver essa relação entre bibliotecas, vale citar que o Spring HATEOAS não se limita à geração de links somente para CRUDs e pode ser usado independente do Spring Data, como foi feito no exemplo aqui apresentado.

Saindo desse mundo Spring, também é possível construir aplicações REST utilizando implementações do JAX-RS. O JAX-RS é uma especificação de serviços em Java que seguem o estilo REST. Algumas das principais implementações dessa API são: RESTEasy, Restlet e Jersey (veja mais informações na seção **Links**). Todas essas implementações são open source e muito maduras. Caso se queira suporte para a criação de links seguindo HATEOAS, deverão ser utilizadas as implementações que seguem a especificação 2.0 do JAX-RS, que atualmente são o RESTEasy e o Jersey. Uma versão do Restlet com esse suporte está prevista para Janeiro de 2015, segundo o rampa do projeto.

Links

Macaxeira Online (código fonte)

https://github.com/antonioreuter/hateoas-oms

Site do projeto JacORB.

http://www.jacorb.org/

SOAP Version 1.2 Part 1: Messaging Framework - Second Edition.

http://www.w3.org/TR/soap12-part1/#intro

JSON Hypertext Application Language.

http://tools.ietf.org/html/draft-kelly-json-hal-06

Protocolo HTTP/1.0.

http://www.rfc-editor.org/in-notes/rfc1945.txt

Protocolo HTTP/1.1.

http://www.ietf.org/rfc/rfc2616.txt

Patch method for HTTP.

http://tools.ietf.org/html/rfc5789

A High-Level Framework for Network-Based Resource Sharing.

http://tools.ietf.org/html/rfc707

Architectural Styles and the Design of Network-based Software Architectures, PhD Thesis, Roy Fielding.

https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding\_dissertation.pdf

JSON Hypertext Application Language.

http://tools.ietf.org/html/draft-kelly-json-hal-06

RESTlet

http://restlet.com

RESTEasy

http://resteasy.jboss.org/

Jersey

https://jersey.java.net/

Referências

[1] RESTful Java with JAX-RS 2.0, 2nd Edition, Bill Burke.

[2] Richardson L., Amundson M., RESTful Web APIs, ISBN: 978-1-449-35806-8.

[3] T. H. Nelson. 1965. Complex information processing: a file structure for the complex, the changing and the indeterminate. In Proceedings of the 1965 20th national conference (ACM '65), Lewis Winner (Ed.). ACM, New York, NY, USA.