artigo

RESTful Web Services e a API JAX-RS

Conheça o poder dos serviços REST e a implementação de Referência da Sun, o Jersey.

Com o surgimento de vários padrões para comunicação entre aplicações com Web Services, aparece uma solução simples e inovadora chamada REST. Seguindo os conceitos de praticidade do protocolo HTTP, REST surge como uma nova opção para problemas de integração nos projetos de software, tornando-se uma opção viável aos Web Services baseados em SOAP, WSDL e WS-. Neste artigo vamos abordar o que é REST, apresentar a API JAX-RS para geração de web services baseados em REST e, por fim, vamos demonstrar diversas maneiras de testar e consumir serviços RESTful com diversos códigos de exemplo.*

Wagner Roberto dos Santos
(wrsconsulting@gmail.com): é lead editor da queue Arquitetura do portal Infoq Brasil (www.infoq.com/br), é entusiasta do NetBeans IDE, tendo diversas participações na criação de plugins, traduções e testes do IDE. Palestrante em diversos eventos nacionais e vencedor de muitos prêmios de desenvolvimento, possui as certificações SCJA, SCJP, SCSNI, SCJWSD, SCBCD e CSM, atualmente presta consultoria como Arquiteto Java e Metodologias Ágeis. Nas horas vagas, mantém o blog http://netfeijao.blogspot.com/.

No ano 2000, um dos principais autores da especificação HTTP, o cientista Roy Fielding, apresentou em sua tese de doutorado (ver referências) uma nova forma de integrar sistemas hipermídias distribuídos chamada REST (Representational State Transfer), baseada nos mesmos princípios de arquitetura por trás da World Wide Web (www), atraindo assim grande atenção.

Uma das grandes novidades no lançamento da especificação Java EE 6 é a inclusão da JSR-311, que define a API JAX-RS (Java API for RESTful Web Service). Neste artigo, vamos entender os conceitos por trás do REST e veremos o quanto é simples desenvolver serviços REST com Java fazendo uso das anotações do JAX-RS.

Veremos também diversas maneiras de se consumir e testar um serviço REST, com o uso de bibliotecas JavaScript e outras opções para o consumo destes serviços em Java.

Introdução ao HTTP – Hypertext Transfer Protocol

Atualmente bilhões de páginas HTML, imagens JPEG, filmes AVI, arquivos de música MP3, arquivos Flash e muitos outros tipos de conteúdo circulam pela internet a cada dia. E o HTTP é o protocolo responsável por transferir esta grande quantidade de informações de uma maneira rápida, conveniente e segura, de diversos servidores web ao redor do mundo para os navegadores web das pessoas em seus desktops.

Isso tudo é possível porque o HTTP é um protocolo genérico, sem estado (stateless), que utiliza um protocolo de transmissão de dados seguro que garante que a informação não será danificada ou perdida durante o transporte. Facilitando a vida tanto do usuário da web, que não precisa se preocupar que suas informações sejam perdidas durante uma transação, e facilita a vida do desenvolvedor que precisa se preocupar apenas com a programação da aplicação.

Clientes Web e Servidores

Todo conteúdo da internet é armazenado em um ou mais servidores web. Para um servidor ser considerado um conteiner web, ele precisa implementar uma das várias versões do protocolo HTTP. A versão mais atual do protocolo HTTP é a versão 1.1.

Se você acessa a internet, então você utiliza um cliente HTTP todos os dias. Neste caso um navegador web qualquer como o Mozilla Firefox ou o Internet Explorer. Os navegodores web fazem requisições HTTP a objetos em servidores web e os apresentam na tela do navegador do usuário. Veja um exemplo desta interação na figura 7.

No exemplo da figura 8, um cliente web qualquer faz uma requisição HTTP (Request) para o servidor que hospeda a página www.mundoj.com.br, o servidor web tenta localizar o documento desejado. Neste caso, a página /default.jsp. Ao localizar a página, o servidor retorna para o cliente uma resposta HTTP (Response) em conjunto com outras informações, como o tipo do objeto, o servidor, status de retorno entre outras informações.

Figura 8. Clientes web e servidores.

Recursos Web

Recursos web são todos os recursos hospedados em um servidor web, ou seja, é qualquer fonte de conteúdo que gere um valor para a web. Como exemplo, podemos citar desde uma simples página estática html até um web service.

Media Types

Por conta dos diversos tipos de conteúdo que trafegam na internet, o protocolo HTTP trata cada recurso trafegado pela web com um rótulo para formato de dados chamado MIME type.

MIME foi projetado inicialmente para resolver os problemas encontrados ao mover tipos de mensagem diferentes entre sistemas eletrônicos de e-mail. Com o sucesso desta solução para e-mail, a especificação HTTP foi atualizada para descrever os tipos de conteúdo multimídia com o uso de rótulos MIME.

Por exemplo, para rotular um tipo de arquivo html, utilizaríamos o tipo text/html, para documentos texto definimos text/plain, para arquivos JPEG utilizamos image/jpeg, entre outros. Por conta desta identificação, os navegadores conseguem renderizar o formato dos dados e apresentar o recurso de maneira adequada, como abrir internamente PDF Reader para arquivos pdf, o media player para arquivos wmv, e assim por diante.

URI

Em computação, uma URI é uma string compacta única utilizada para identificar um recurso físico ou abstrato, geralmente na Internet. Uma URI pode ser classificada como um local (URL), um nome (URN) ou ambos.

URL é o formato mais conhecido por ser o meio de localização de recursos que utilizamos ao navegar pela Internet.

URN significa (Uniform Resource Name) e pode ser atribuído a um nome de uma pessoa ou pode ser utilizado para identificar um objeto, como, por exemplo, um código de barras ou o SBN de um livro. Enquanto uma URN define a identidade única de um item, a URL define a localização deste mesmo item na rede. Ela possui uma sintaxe genérica composta de quatro partes definidas pelo padrão de Internet RFC 3986, apresentada abaixo:

```
<scheme name> : <hierarchical part> [ ? <query> ] [ # <fragment> ]
```

As URIs são definidas em esquemas que definem uma sintaxe específica para cada tipo e protocolo associado, existem diversos URI Schemas oficiais registrados pela IANA (Internet Assigned Numbers Authority), definidos para diversos propósitos como DNS, FTP, LDAP etc. Segue um exemplo dessa sintaxe em um caso real:

```
http://www.exemplo.com.br/produto/informatica?codigo=10#bookmark
```

Para quem conhece ambientes web, fica muito fácil compreender esta URI decompondo o exemplo acima e associando cada parte definida na sintaxe genérica com cada parte do nosso exemplo.

Podemos reconhecer no exemplo o trecho http como sendo o schema, que define o protocolo utilizado para comunicação com o recurso. O host no trecho "www.exemplo.com.br" identifica o servidor onde está localizado o recurso. O caminho "produto/informatica" indica onde o recurso se encontra no servidor, a query "codigo=10", que utilizamos para adicionar informações extras para identificação ou especificar o tipo de retorno do recurso. E finalmente o fragmento "bookmark", que não é transmitido para o servidor, sendo aplicado apenas no ambiente do cliente. Veja outros exemplos de URIs:

```
ISBN-13: 978-0-13-142246-9

    Código ISBN que é uma URN e que identifica o código de um livro;

    http://www.mundoj.com.br/

    URI que identifica um local (URL), neste caso a página do site da revista Mundoj;

    file:///C:/Wagner/artigos/REST.doc

    URI que identifica um recurso no disco.
```

Métodos

Quando fazemos uma requisição HTTP a servidor web, podemos dizer que estamos enviando uma mensagem. O HTTP suporta diversos tipos de comandos de requisição e chamados de métodos http.

Toda mensagem HTTP possui um método. O método diz ao servidor qual a ação que ele deve tomar (retornar uma página, executar um programa, excluir um arquivo etc.). Para tanto, foram disponibilizados alguns métodos HTTP, conhecidos também como verbos, pois o verbo irá determinar a ação a ser tomada no recurso. Na tabela 1 apresentamos os métodos mais comuns.

Método	Descrição
GET	Solicita que o servidor envie um recurso.
PUT	O inverso do GET. Realiza operações de escrita no servidor.
DELETE	Solicita a exclusão de um recurso no servidor.
POST	Foi projetado para envio de dados de input, como, por exemplo, os dados de um formulário para o servidor.
HEAD	Mesmo comportamento que o método GET, porém o servidor retorna apenas o cabeçalho da resposta.

Tabela 1. Métodos comuns do HTTP.

Ainda existem outros métodos, como TRACE, OPTIONS e métodos de extensão do HTTP como WebDAV.

Código de Status

Toda resposta do servidor Web a uma requisição vem com um código de status utilizado para identificar o status da operação. Segue, na tabela 2, a classificação dos tipos de códigos de status que podemos receber de um servidor web. Quem nunca recebeu o código de retorno 400 de Bad Request, ou ainda o 404 Not Found?

Range definido	Descrição
100-101	Informacional
200-206	Sucesso
300-305	Redirecionamento
400-415	Erro do cliente
500-505	Erro do servidor

Tabela 2. Códigos de status HTTP segmentados em categorias.

Afinal, o que é REST?

Conforme a definição do criador Roy Fielding em sua tese de doutorado, REST, que significa Representational State Transfer (ou Transferência de Estado Representativo), é um estilo de arquitetura de software para sistemas hipermídia distribuídos, como, por exemplo, a Web do jeito que conhecemos atualmente, ou seja, onde utilizamos um navegador web para acessar recursos que estamos interessados, geralmente uma página HTML, ou um documento XML, mediante a digitação de uma URL.

Ainda em sua tese, Fielding afirma que o estilo REST é uma abstração dos elementos arquiteturais de um sistema multimídia distribuído definidos como:

Elementos de dados: são classificados como os recursos, o metadado destes recursos e seus identificadores (URIs) e as suas representações que pode ser um documento HTML, XML, imagens etc.;

Conectores: são os vários tipos de conectores que REST utiliza para encapsular as atividades de acesso aos recursos e a transferência de uma representação de um recurso. Podemos classificar os conectores como clientes, servidores, cache e tunnel;

Componentes: são classificados pelos seus papéis na ação de uma aplicação, os tipos de componentes são classificados como servidores de origem, gateways, proxy e user agents. Neste último caso, o exemplo mais comum é um navegador Web.

Em uma arquitetura REST, dados e funcionalidades são considerados recursos e estes recursos são acessados via URIs (Uniform Resource Identifier), geralmente via links.

REST foi concebido baseando-se em HTTP, até pela formação de Fielding que foi um dos principais autores da especificação do HTTP. Por ser baseado em HTTP, a arquitetura de um sistema REST geralmente é cliente-servidor e os serviços não possuem estado (stateless).

Ao aplicar os princípios REST no desenvolvimento de uma aplicação web, exploramos o uso do protocolo HTTP e de URIs de forma natural, o que torna as aplicações mais simples, leves e de alta performance.

Recursos e representações

Para um melhor aproveitamento da leitura, é recomendado que o leitor tenha um conhecimento básico do Protocolo de Transferência de Hipertexto (HTTP), que pode ser adquirido no quadro "Introdução ao HTTP".

Antes de nos aprofundarmos nos conceitos REST, precisamos distinguir o que são Recursos e o que são Representações.

Podemos entender uma URI como um ID único, por exemplo, podemos ter uma URI http://www.shopping.com.br/calcados/tenis/1512 que identifica um recurso. Neste caso, esta URI é uma URL que aponta para o recurso calcados/tenis/1512. Por ser uma URL lógica, ela poderia ser uma página estática HTML, ou poderia ser um pedaço de uma página, como uma div. Admitindo que esta é uma URL válida, geralmente ao digitarmos esta informação em um browser o retorno provável é em HTML, pois é uma Representação que foi projetada para a compreensão dos seres humanos.

Ainda neste exemplo, o nosso recurso é calcados/tenis/1512, que podemos entender como um tênis cujo código é 1512 e a sua representação ao digitar esta URL em browser seria o HTML.

Agora porque não criar representações para a leitura de máquinas também? Poderíamos ter este mesmo recurso com representação em XML, JSON, JPG etc. O formato de uma representação é determinado pelo tipo de conteúdo e no mundo REST a interação com a representação do recurso é determinada pelo método HTTP que vamos utilizar.

Uma vez que o recurso é identificado, precisamos definir a ação que iremos tomar sobre este recurso, em REST estas operações são descritas com o uso de quatro verbos que são GET, PUT, POST e DELETE. No mundo web, automaticamente associamos estes verbos aos métodos do protocolo HTTP, que possuem o mesmo nome.

Um sistema clássico que utiliza REST e que todos utilizam é a Web. Isso mesmo, nós estamos utilizando REST há um bom tempo sem darmos conta disso.

Saindo um pouco da teoria proposta pelo dr. Fielding em sua tese e trazendo tudo isto para o mundo que conhecemos, quando criamos web services utilizando o estilo de arquitetura REST, estamos criando "web services RESTful". Os web services RESTful expõem os recursos através de URIs e utilizam os métodos do HTTP para criar, retornar, alterar e excluir os seus recursos.

Em RESTful mapeamos os principais métodos HTTP (POST, DELETE, GET E PUT) com operações CRUD (CRUD = Create, Retrieve, Update e Delete) de um banco de dados. Veja a tabela 1.

Métodos HTTP	Operações CRUD
GET	SELECT
POST	INSERT, UPDATE, DELETE
PUT	CREATE, UPDATE
DELETE	DELETE

Tabela 1. Associação dos métodos HTTP com operações CRUD.

O método PUT nos dá a possibilidade de criar um novo recurso ou substituir por outro mais atualizado. O método DELETE é utilizado para excluir um recurso existente e o método GET que é o método mais utilizado, pois com ele podemos efetuar operações de leitura em algum recurso ou utilizá-lo como mecanismo de busca. Veja alguns exemplos na Listagem 1.

Listagem 1. Exemplos de operações REST.

```
GET http://www.exemplo.com.br/wagner/fotos/palestra.jpg
    Lendo uma foto em um repositório utilizando GET;

    PUT http://www.exemplo.com.br/wagner/fotos/palestra.jpg
    Podemos atualizar a mesma foto, utilizando POST;

    DELETE http://www.exemplo.com.br/wagner/fotos/palestra.jpg
    Excluindo uma foto utilizando DELETE.
```

JAX – RS – Java API for RESTful Web Services

JAX-RS é a solução do JCP para o estilo de programação REST. A proposta final da especificação foi liberada para o público no início de agosto de 2008. A especificação define um conjunto de APIs Java para auxiliar no desenvolvimento de web services baseados em REST.

O objetivo da API é fornecer um conjunto de anotações, classes e interfaces para expor uma classe POJO como um web service RESTful, de modo que possamos fazer uma programação fácil e de alto nível.

Trabalhando com os recursos

Para uma classe ser determinada como um recurso, ela tem que ser uma classe POJO com pelo menos um método anotado com a anotação @ Path.

A anotação @Path pode ser colocada na declaração de classe ou de um método e possui o elemento value obrigatório. Por este elemento definimos o prefixo da URI que a classe ou o método irá atender. Na Listagem 2, a classe Repositorio é identificada pela URI relativa "/repositorio/{id}", em que {id} é o valor do parâmetro id, fornecido junto a URI.

Mais adiante, demonstraremos como extrair valores como esse de uma URI utilizando anotações específicas.

Listagem 2. Mapeando uma URI para uma classe com @Path.

```
@Path("/repositorio/{id}")
public class Repositorio { ... }
```

A especificação define que no ciclo de vida de um recurso, por padrão, sempre que for feita uma requisição a um recurso, será criada uma nova instância de uma classe REST. Primeiro o construtor é chamado pelo contêiner, por conta disto, o construtor da classe deve ser público. Após este primeiro passo, o contêiner efetua as injeções de dependência nos devidos métodos e o método designado para aquele recurso é invocado. E finalmente após a chamada ao método o objeto fica disponível para o garbage collector.

Geralmente a anotação @Path é incluída na declaração de um método quando queremos atribuir um caminho mais específico para um recurso, de forma a especializar nosso método, como na Listagem 3. Note no exemplo da Listagem 3 como a URI é mapeada com a classe e o método.

Listagem 3. Mapeando uma URI para uma classe com @Path.

```
@Path("/vendas/")
public class Repositorio {
   @GET
   @Produces("application/xml")
   @Path("/pedidos/{numPedido}/")
   public PedidoAsXML getPedido(@PathParam("numPedido") Integer id){
      // retorna Pedido em formato XML.
   }
}
```

Acessando os recursos

Para acesso aos recursos, a especificação JAX-RS define um conjunto de anotações correspondentes aos métodos HTTP, como @GET, @POST, @PUT, @DELETE, @HEAD. Elas devem ser atribuídas a métodos públicos.

O método anotado com @GET, por exemplo, irá processar requisições HTTP GET na URI atribuída. O comportamento do recurso é determinado pelo método HTTP ao qual o recurso está respondendo.

É bom entender o papel e o uso de cada um destes métodos HTTP no momento de projetar nossos serviços.

Além dos métodos definidos pelo JAX-RS, podemos criar uma anotação customizada como @MKCOL, com uso da anotação HttpMethod, conforme ilustra a Listagem 4.

Com esta anotação, podemos criar métodos customizados e estender a gama dos métodos preexistentes. Seria possível, por exemplo, utilizar os métodos definidos pelo WebDAV, como fazemos na Listagem 4. Também podemos alternar um método padrão para a anotação customizada. Para isso poderíamos simplesmente informar como valor para anotação HttpMethod o valor do método que queremos sobrepor. Por exemplo, para sobrepor o método PUT, a declaração da anotação ficaria @HttpMethod("PUT").

Listagem 4. Utilizando a anotação HttpMethod para criar uma anotação customizada.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod("MKCOL")
public @interface MKCOL {
}
```

Representações: Quais são os sabores?

As classes de uma aplicação RESTful podem ser declaradas para suportar diversos tipos de formatos durante um ciclo requisição-resposta. Para isso, a especificação define as anotações @Consumes e @Produces para o tratamento da request e response, respectivamente.

Com estas anotações, o servidor declara o tipo de conteúdo que será trafegado, através do cabeçalho (header) do protocolo HTTP, estas anotações podem ser aplicadas na declaração de uma classe, ou podemos utilizar estas anotações na declaração do método para sobrescrever a declaração da classe, na ausência destas anotações, será assumido como default qualquer tipo de retorno ("*/*").

No caso da anotação @Produces, ela é utilizada para mapear uma requisição de um cliente com o cabeçalho do cliente (parâmetro Accept). Desta maneira, podemos definir mais de um tipo de retorno da URI solicitada, como JSON e XML, conforme exemplo da Listagem 5.

Listagem 5. Declarando o tipo de retorno com a anotação @Produces.

```
@GET
@Produces({"application/xml", "application/json"})
public PessoaConverter getPessoa(@QueryParam("CPF") String numCPF) {
   // Retorna representação em XML ou JSON
}
```

Pegando o exemplo da Listagem 5, podemos testar o retorno da chamada utilizando uma das ferramentas citadas no tópico "Como consumir serviços REST", como a ferramenta RESTClient, por exemplo, e incluir na aba "Headers" o parâmetro "Accept" e no campo Value especificar o tipo de retorno primeiro com XML e depois com JSON, conforme figura 1.

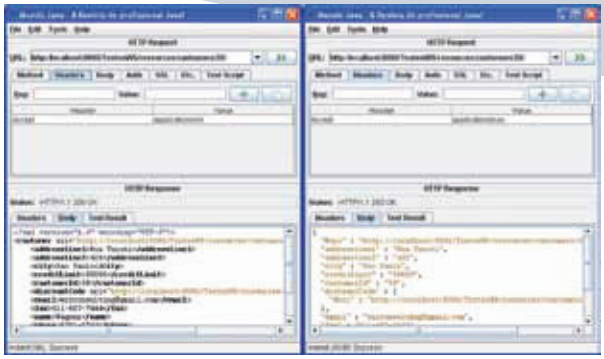

Figura 1.Testando tipos de retorno para a mesma URI com RESTClient.

Com a anotação @Consumes, por outro lado, podemos definir os tipos de mídia que um recurso em particular consome. Como o exemplo da Listagem 6, na qual declaramos que o método consome apenas formatos do tipo XML e JSON:

Listagem 6. Uso da anotação @Consumes.

```
@PUT
@Consumes("application/xml","application/json")
@Path("/autores/")
public Response putPessoa(PessoaBinding pessoa) {}
```

Extraindo parâmetros e valores da URI na requisição.

JAX-RS fornece algumas anotações para extrair informações de uma requisição. Existem seis tipos de parâmetros (ver tabela 2) que podemos extrair para utilizar em nossa classe recurso. Para os parâmetros de query utilizamos a anotação @QueryParam. Para os parâmetros de URI (path), existe a anotação @PathParam. Para os parâmetros de formulário, existe @FormParam. Já para parâmetros de cookie existe @CookieParam. Para os parâmetros de header existe @HeaderParam e, finalmente, para os parâmetros de matriz existe a anotação @MatrixParam.

Estas anotações podem ser aplicadas diretamente a parâmetros de um método. Desta forma, vinculamos o valor do parâmetro de uma URI a algum parâmetro de entrada de um método. Para contextualizar, veja um exemplo na Listagem 7.

Listagem 7. Mapeando os parâmetros de uma URI para os parâmetros de um método.

```
@Path("/Mundoj/")
public class EditoraResource {
   @GET
   @Produces("application/xml")
   @Path("/autores/{nomeAutor}/")
   public PessoaBinding getPessoa(@PathParam("nomeAutor") String name,
         @QueryParam("idade") int idade,
   @HeaderParam("CPF") String numCPF,
   @MatrixParam("statusCivil") String statusCivil) {
      return new PessoaBinding(name, idade, numCPF, statusCivil);
   }
}
```

Na Listagem 7, vemos a aplicação de várias anotações de parâmetro de URI em um único método. Para entender melhor como será feito o DE-PARA da URI para os parâmetros de entrada, vamos fazer uma chamada a este recurso com a biblioteca curl, apresentado na figura 2.

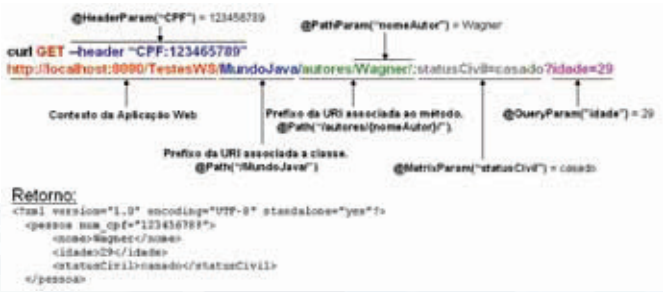

Figura 2. Fragmentando a URI para demonstrar anotações de mapeamento.

No exemplo da Listagem 7 não demonstramos o uso das anotações CookieParam e FormParam, mas o seu uso é bem similar. No caso de FormParam, esta anotação pode capturar todos os valores submetidos de um formulário HTML e injetá-los nos parâmetros desejados. Veja um exemplo simples na Listagem 8. Trata-se de um formulário HTML e de um método que irá receber estas informações.

No caso da anotação CookieParam, conseguimos injetar o valor de cookie ou a classe Cookie do javax.ws.rs.core, que representa o valor de um cookie HTTP na invocação de um método.

Anotação	Aplicado em	Descrição
PathParam	Parâmetro, campo ou método.	Especifica que o valor do parâmetro, campo ou propriedade de um bean será extraído do valor de um parâmetro indicado na URI. O valor da anotação identifica o nome do parâmetro da URI.
QueryParam	Parâmetro, campo ou método.	Extrai o valor de um parâmetro indicado na query de uma URI e atribui a um parâmetro, campo ou propriedade de um bean. O valor da anotação identifica o nome do parâmetro na query.
FormParam	Parâmetro.	Especifica que o valor do parâmetro será extraído de um parâmetro de formulário no body da requisição.
MatrixParam	Parâmetro, campo ou método.	Extrai os valores (chave/valor) da matriz de parâmetros da URI.
CookieParam	Parâmetro, campo ou método.	Extrai os valores de cookies vinculados à sessão.
HeaderParam	Parâmetro, campo ou método.	Extrai dados do cabeçalho de uma requisição HTTP.

Tabela 2. Anotações JAX-RS para extração de informações da URI.

Listagem 8. Uso da anotação FormParam.

```
<form method="POST" action="/Mundoj/autores">
   Nome: <input type="text" name="nomeAutor"> <br>
   Idade: <input type="number" name="idade">
</form>

@Path("/Mundoj/")
public class EditoraResource {
   @GET
   @Path("/autores/")
   public PessoaBinding getPessoa(@FormParam("nomeAutor") String name,
         @FormParam("idade") int idade) {
      return new PessoaBinding(name, idade);
   }
}
```

Dados de contexto

A especificação JAX-RS dispõe de um recurso para a obtenção de informações do contexto da aplicação e de requisições individuais. Estas informações são disponíveis tanto para as classes recursos quanto para os providers. Para a recuperação destas informações, existe a anotação @Context, que ao ser aplicada sobre um campo, método ou parâmetro, identifica um alvo a ser injetado pelo contêiner.

O contêiner fornece instâncias dos recursos listados na tabela 3, mediante a aplicação da anotação @Context.

Classe	Descrição
UriInfo	Esta classe fornece informações relativas à URI requisitada, como o caminho absoluto da URI, os parâmetros de query, entre outras informações. Na Listagem 9, nós a utilizamos para buscar o caminho absoluto da request.
HttpHeaders	Fornece informações sobre o Header HTTP. Pode ser injetada em um parâmetro de método ou campo de uma classe, entre as informações, podemos saber os tipos de mídia e linguagens aceitas, os valores do header da requisição HTTP, retornados como chave-valor pela classe MultivaluedMap.
Request	O uso de Request simplifica o suporte, a negociação do conteúdo e as pré-condições da chamada ao método e determina a melhor representação pelo método evaluatePreconditions(), baseado em parâmetros, como uma entity tag, por exemplo.
SecurityContext	Fornece informações relacionadas ao contexto de segurança, e o esquema de autenticação utilizado.
Providers	Esta classe fornece um meio para efetuar lookups em instâncias de providers baseado em critérios de busca.

Tabela 3. Lista de recursos injetados pelo contêiner pela anotação @Context.

Tratando o retorno dos métodos ao cliente

Os tipos de retorno de uma chamada a um método recurso podem ser do tipo void, Resource, GenericType ou outro tipo Java. Esses tipos de retorno são mapeados ao entity body da Response cada um de uma maneira, de acordo com o provider padrão, conforme veremos a seguir.

Para os retornos do tipo void, o retorno será um corpo vazio com status 204 do HTTP. Para tratar o retorno ao cliente foi disponibilizada a classe abstrata Response. Com essa classe, definimos um contrato entre a instância de retorno e o ambiente de execução, quando uma classe precisa fornecer metadados para ambiente de execução.

Podemos estender esta classe diretamente ou, ainda melhor, podemos criar uma instância utilizando sua classe interna ResponserBuilder. Por essa classe, podemos construir objetos do tipo Response, adicionar metadados, adicionar cookies, adicionar dados no Header, informar a linguagem, entre outras informações.

Na Listagem 9, no método putPessoa, fazemos uso do método Response. Note que não a instanciamos diretamente, pois ela é uma classe abstrata que implementa o padrão de projeto Builder.

Dentro do método, primeiramente efetuamos uma chamada ao método estático created, passando como parâmetro a URI que obtemos através da classe injetada UriInfo. Esta classe retorna o objeto ResponseBuilder, que é uma subclasse de Response, esta subclasse é utilizada exclusivamente para criar instâncias de Response.

Por ResponseBuilder ser uma classe de construção (Builder), podemos efetuar chamadas recursivas aos métodos de parametrização. Após a chamada ao método created, chamamos o método status, no qual atribuímos o código de status HTTP 202 (Accepted) e logo após atribuímos uma entidade à requisição. No nosso exemplo, um código HTML simples, pelo método entity. Na chamada ao método type seguinte, especificamos o tipo de mídia trafegado, neste caso TEXT_HTML. No final, fazemos uma chamada ao método build, que constrói o objeto Response.

E por fim o objeto GenericEntity representa uma entidade de um tipo genérico, muito útil quando precisamos retornar uma Response personalizada e reter informações genéricas. Pois informações de tipos genéricos são apagadas ao utilizar uma instância.

Listagem 9. Tratando a Response do cliente.

```
@Context protected UriInfo uriInfo;
@PUT
@Consumes("application/xml")
@Path("/Mundoj/update/")
public Response putPessoa(PessoaBinding pessoa) {
   String retorno = "<html><body><h1>Bem vindo "+pessoa.getNome()
   +"</h1></body></html>";
   Response response = Response.created(uriInfo.getAbsolutePath()).
         status(Response.Status.ACCEPTED).
         entity(retorno).
         type(MediaType.TEXT_HTML).
         build();
   return response;
}
```

Entity providers

Entity providers fornecem serviços de mapeamento entre representações e seus tipos associados Java. Existem dois tipos de entity providers, MessageBodyReader e MessageBodyWriter.

A especificação JAX-RS define que para alguns tipos o contêiner pode automaticamente serializar (marshal) e deserealizar (unmarshal) o corpo de diferentes tipos de mensagens, listados na tabela 4.

Tipo de mídia	Tipo Java
Todos os tipos de mídias (*/*)	byte[]
Todos os tipos de mídias-texto (text/*)	java.lang.String
Todos os tipos de mídias (*/*)	java.io.InputStream
Todos os tipos de mídias (*/*)	java.io.Reader
Todos os tipos de mídias (*/*)	java.io.File
Todos os tipos de mídias (*/*)	javax.activation.DataSource
Todos os tipos de mídias (*/*), somente MessageBodyWriter	StreamingOutput
Tipos XML (text/xml, application/xml e application/*+xml)	javax.xml.transform.Source
Tipos XML (text/xml, application/xml e application/*+xml)	javax.xml.bind.JAXBElement e classes JAXB de aplicação.
Conteúdo de Formulário (application/x-www-form-urlencoded)	MultivaluedMap<String, String>

Tabela 4. Tipos de mídia e seus tipos Java correspondentes.

Para requisições HTTP, podemos mapear o corpo da entidade para um parâmetro de método com uso da interface MessageBodyReader<T>, para o tratamento das responses, o valor de retorno é mapeado para o corpo da entidade de um método HTTP com uso da interface MessageBodyWriter<T>.

Pode ser que no desenvolvimento de nossas aplicações, estes tipos padrões não atendam à necessidade de negócio e tenhamos que lidar com tipos que não sejam suportados pelos tipos default, para contornar esta limitação, a API JAX-RS permite que a criação de Providers para Message Body customizáveis, com métodos para conversão de InputStream/OutputStream para objetos Java do tipo T.

Para criar nosso próprio provider customizado, a especificação disponibiliza a anotação @Provider, que ao ser aplicado sobre uma classe, estamos automaticamente registrando esta classe junto ao contêiner.

Porém, é importante ressaltar que esta funcionalidade apesar de muito útil pode ser tornar um problema em grandes projetos, que podem utilizar providers com o mesmo nome em diferentes bibliotecas, podendo ocasionar conflitos.

Caso a aplicação necessite de informações adicionais, como HTTP Headers ou um código de status diferente, o método pode retornar o objeto Response que encapsule a entidade.

Veja um exemplo extraído de um sample da implementação de referência da Sun, o Jersey, na Listagem 10, esta classe implementa um MessageBodyWriter para uma classe Properties.

Listagem 10. Uso da tag @Provider.

```
@Produces("text/plain")
@Provider
public class PropertiesProvider implements MessageBodyWriter<Properties>
{
   public void writeTo(Properties p,
         Class<?> type, Type genericType, Annotation annotations[],
         MediaType mediaType, MultivaluedMap<String, Object> headers,
         OutputStream out) throws IOException {
      p.store(out, null);
   }
   public boolean isWriteable(Class<?> type, Type genericType,
   Annotation annotations[], MediaType mediaType) {
      return Properties.class.isAssignableFrom(type);
   }
   public long getSize(Properties p, Class<?> type, Type genericType,
   Annotation annotations[], MediaType mediaType) {
      return -1;
   }
}
```

Tratando exceções

Para tratamento de exceções, a especificação JAX-RS define a exceção WebApplicationException que estende RuntimeExcetion, que pode ser lançada por um método de recurso por um provider ou por uma implementação de StreamingOutput. Esta exceção permite que abortemos a execução de um serviço JAX-RS.

Como parâmetro de construtor, podemos utilizar um código de status HTTP ou até um objeto Response. Veja um exemplo na Listagem 11.

Listagem 11. Uso de exceção com WebApplicationException.

```
@GET
@Produces("application/xml")
@Path("/autores/{personName}/{idade: [0-9]+}/")
public PessoaBinding getPessoa(@PathParam("personName") String name,
@PathParam("idade") int idade, @HeaderParam("CPF") String numCPF) {
   if (idade <= 0 || idade > 120){
      throw new WebApplicationException(Response.status(412).
entity("Idade inválida!").build());
   }
   return new PessoaBinding(name, idade, numCPF);
}
```

Por padrão, quando uma classe JAX-RS ou um método provider lança uma exceção em tempo de execução, essa exceção é mapeada a um código de status HTTP adequado. Nós podemos customizar a nossa exceção conforme a necessidade. Para isto, a especificação define a interface ExceptionMapper. Com ela, podemos criar nossos próprios providers e customizar este mapeamento. Para tanto, a implementação desta interface deve estar anotada com @Provider. Veja um exemplo na Listagem 12.

Listagem 12. Mapeando uma exceção Java para uma Response.

```
@Provider
public class IdadeInvalidaExceptionMapper implements
ExceptionMapper<IdadeInvalidaException>{
   public Response toResponse(IdadeInvalidaException ex) {
      return Response.status(412).entity(ex.getMessage()).build();
   }
}
```

Na Listagem 12 perceba que registramos a classe ExceptionMapper da mesma maneira que registramos MessageBodyReaders e MessageBodyWriters. Ao utilizarmos a exceção IdadeInvalidaException em um método RESTful como no método da Listagem 13, o contêiner irá em tempo de execução identificar o provider e irá mapear esta exceção com IdadeInvalidaExceptionMapper.

Listagem 13. Uso de exceção de negócio, que é mapeada para Response.

```
@GET
@Produces("application/xml")
@Path("/autores/{personName}/{idade: [0-9]+}/")
public PessoaBinding getPessoa(@PathParam("personName") String name,
@PathParam("idade") int idade, @HeaderParam("CPF") String numCPF)
throws IdadeInvalidaException {
   if (idade <= 0 || idade > 120)
      throw new IdadeInvalidaException("Idade Inválida!");
   return new PessoaBinding(name, idade, numCPF);
}
```

Testando e consumindo serviços REST

A especificação JAX-RS, como vimos anteriormente, é uma ótima opção para criar web services REST e fornece meios de desenvolver componentes server-side, mas não descreve como os desenvolvedores devem desenvolver seus componentes client-side em Java, e essa já é uma das promessas para a

próxima release do JAX-RS.

Pelo fato de nossos serviços RESTful serem URIs e a forma de acesso a estes serviços serem os próprios métodos HTTP, podemos trabalhar diretamente com requisições HTTP ou utilizar bibliotecas para facilitar este trabalho. Felizmente é relativamente fácil trabalhar diretamente com requests e responses HTTP, e as linguagens mais populares de programação possuem métodos/bibliotecas HTTP, como, por exemplo, urllib2 e httplib em Python, libcurl em PHP, HTTPWebRequest em C#, open-uri em Ruby, e o pacote java.net.* e o projeto HttpClient da Apache para Java, entre outros. Mas para qualquer linguagem que seja feita a requisição ao serviço RESTful, temos que passar por alguns passos, conforme segue:

1º montar os dados que irão trafegar pela requisição HTTP, como a URI, HTTP header (se houver) e o método HTTP desejado;

2º formatar estes dados como uma requisição HTTP, e enviá-los para um servidor HTTP apropriado;

3º efetuar o parsing dos dados retornados (XML, JSON etc.) para as estruturas de dados que o seu programa precisa.

Para facilitar a pesquisa, montamos um pequeno guia para os desenvolvedores e estudiosos que querem aprender um pouco mais sobre REST, no qual iremos apresentar algumas bibliotecas para teste e consumo de serviços RESTful.

cURL

Se o intuito for apenas testar os serviços REST desenvolvidos e validar o retorno, o mais simples é utilizar ferramentas existentes na web como é o caso da biblioteca cURL, que é uma ferramenta de transferência de arquivos entre cliente-servidor desenvolvida em C. Ela suporta protocolos como HTTP, HTTPS, FTP, FTPS etc.

A Listagem 14 apresenta exemplos de como fazer uma requisição GET e POST com uso da biblioteca cURL. Como podemos ver, ela não possui uma interface gráfica, sendo uma ferramenta de linha de comando.

Listagem 14. Uso da biblioteca cURL.

```
curl http://localhost:8080/ContatosMJ/resources/customers/2/cliente/

result:
{"@uri":"http://localhost:8080/ ContatosMJ/resources/customers/2/
cliente/","addre
ssline1":"9754 Main Street","addressline2":"P.O. Box
567","city":"Miami","credit
Limit":"50000","customerId":"2","discountCodeRef":{"@uri":"http://
localhost:8080
/CustomerDB/resources/customers/2/cliente/discountCode/","discountCode
":"77"},"e
mail":"www.tsoftt.com","fax":"305-456-8889","name":"Livermore
Enterprises","phon
e":"305-456-8888","state":"FL","zip":"33055"}

// Fazendo uma requisição POST, passando como  query parameter name =
JumboComLtda
curl -d name=JumboComLtda http://localhost:8080/ ContatosMJ /resources/
customers/2/cliente/

// Excluindo um registro com DELETE, pelo parâmetro –X
curl -v –X DELETE http://localhost:8080/ContatosMJ/resources/customers/99/
Registro excluído com sucesso !
```

RESTClient

Como o nome sugere, RESTClient é uma aplicação Swing própria para auxiliar nos testes de serviços RESTful. Para utilizar, basta efetuar o download (veja o site nas referências) do arquivo jar (com dependências), e começar a utilizar digitando o comando java –jar restclient-X.X-jar-with-dependencies.jar, em que X.X é o número da versão atual. Outra opção é executar RESTClient diretamente pela sua aplicação, como fazemos na Listagem 15. Ao optar por qualquer uma das opções será apresentada uma tela bem intuitiva, como na figura 3.

Listagem 15. Chamando o Aplicativo RESTClient via código.

```
public static void main(String[] args) {
   RESTMain app = new RESTMain("Mundoj - A Revista do profissional Java!");
   app.getView();
}
```

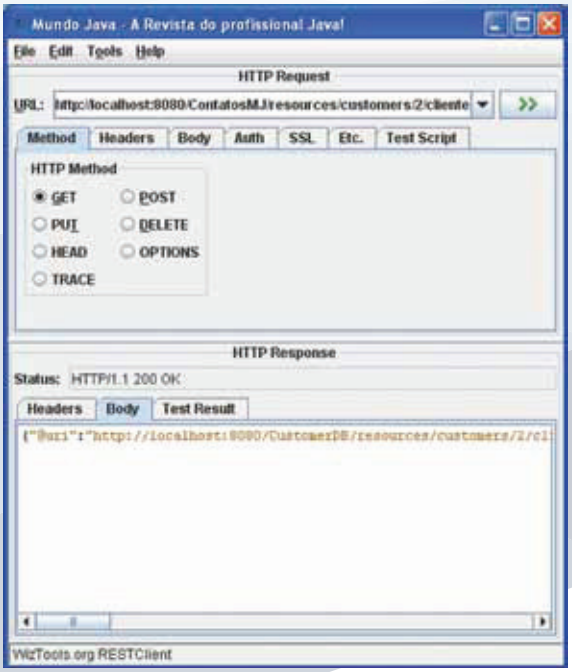

Figura 3. Tela inicial do RESTClient.

Pela aparência da interface gráfica, podemos deduzir facilmente o modo de utilizá-la, basta digitar no campo URL o caminho desejado, selecionar algum método HTTP na aba Method, e executar a consulta clicando no botão [>>]. O resultado será apresentando no bloco HTTP Response. Ainda é possível fazer testes unitários baseados em JUnit 3.x, pela opção Test Script.

Testando Web Services RESTful no NetBeans

Para quem é usuário do NetBeans, outra opção para testar Web Services RESTful é utilizar o suporte do próprio IDE, com um projeto Web criado e os serviços RESTful devidamente configurados. É possível testá-los clicando com o botão direito do mouse em cima do projeto e selecionar a opção "Test RESTful Web Services" (figura 4), lembrando que esta opção só estará disponível se o projeto WEB possuir serviços REST.

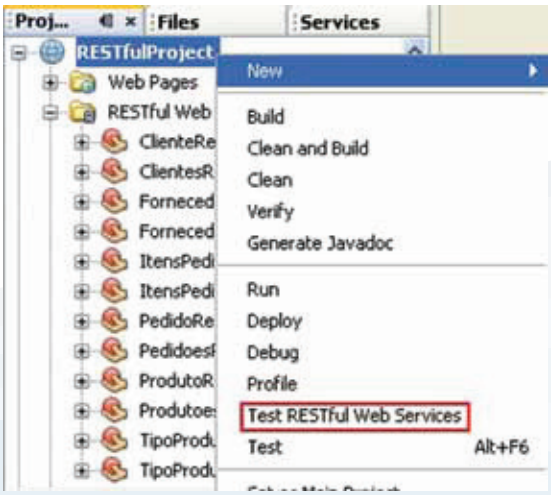

Figura 4. Opção de testar serviços RESTful com o NetBeans.

Ao selecionar esta opção, será feito o build e o deploy da aplicação web, e ao final do processo será disponibilizada uma página de testes web, como mostra a figura 5.

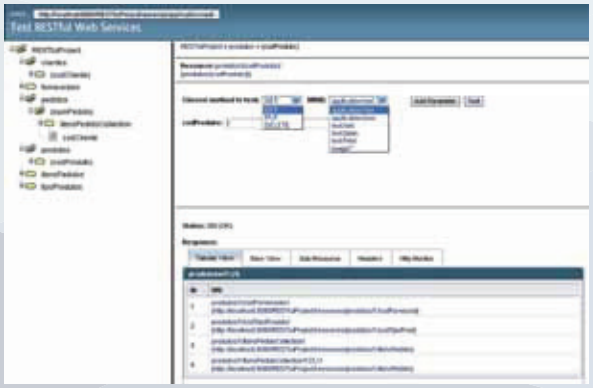

Figura 5. Tela de testes de Web Services RESTful.

Na página apresentada é possível testar todos os serviços disponíveis, criar novos parâmetros para a requisição (botão "Add Parameter"), e também é possível selecionar o tipo de método HTTP para teste e o tipo MIME de retorno.

Para iniciar o teste, basta clicar no botão "Test", após a execução, dentro da seção Response, podemos analisar os dados de retorno, os dados do cabeçalho e o status da chamada.

Além disso, de acordo com os serviços criados, o NetBeans ainda gera o arquivo WADL, visível no canto superior esquerdo da figura 5.

JAXB

JAXB (Java Architecture for XML Binding) fornece a API, as ferramentas e um framework que automatiza o mapeamento entre documentos XML e objetos Java. Ou seja, fornece compiladores que compilam Schemas XML para objetos Java. Em tempo de execução, podemos deserializar (unmarshal) o conteúdo de um arquivo XML para representações Java.

Além disso, podemos acessar, alterar e validar a representação Java contra regras de um Schema e, por fim, podemos serializar (marshal) o conteúdo de um objeto Java em conteúdo XML. Veja sua arquitetura na figura 6.

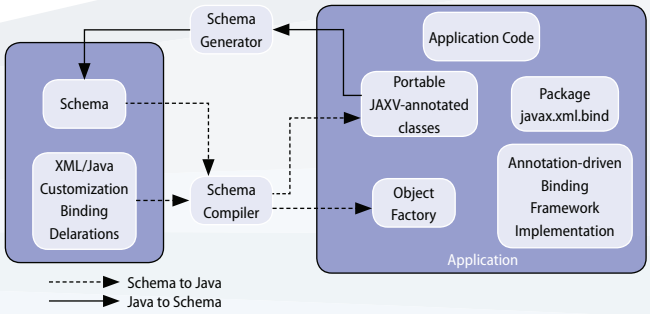

Figura 6. Fluxo JAXB.

Está fora deste artigo um estudo mais aprofundado sobre o JAXB, mas apenas para conhecimento, a API JAXB acabou se tornando a forma padrão de mapeamento entre Java e XML. Com JAXB, temos anotações que nos permitem criar uma representação em Java de um Schema XML. Estas anotações estão presentes no pacote javax.xml.bind.annotations, e possuem anotações associadas a pacotes Java (@XmlSchema, @XmlSchemaType etc.), as classes Java (@XmlType e @XmlRootElement), a propriedade e campos (@XmlElemen e @XmlAttribute).

Para exemplificar, considere o exemplo da Listagem 16, esta é uma classe POJO representando uma pessoa, com anotações JAXB. Ao fazer um marshalling de uma instância da classe PessoaBinding para XML, teremos o resultado apresentado na Listagem 17.

Listagem 16. Classe PessoaBinding com anotações JAXB.

```
@XmlRootElement(name="pessoa")
@XmlType(name="", propOrder={"nome","idade","statusCivil"})
public class PessoaBinding {
   /* Construtores e Setters omitidos */
   private String nome;
   private int idade;
   private String statusCivil;
   private String cpf;

   @XmlElement
   public String getNome() {
      return nome;
   }
   @XmlElement
   public int getIdade() {
      return idade;
   }
   @XmlAttribute(name="num_cpf")
   public String getCpf() {
      return cpf;
   }
   @XmlElement
   public String getStatusCivil() {
      return statusCivil;
   }
}
```

Listagem 17. XML gerado após marshalling de classe JAXB PessoaBinding.

```
<?xml version="1.0" encoding="UTF-8"?>
<pessoa num_cpf="123456789">
   <nome>Wagner</nome>
   <idade>29</idade>
   <statusCivil>Casado</statusCivil>
</pessoa>
```

Como vimos anteriormente no tópico "JAX-RS – Java API for RESTful Web Services", a especificação já fornece alguns Entity Providers padrões, entre eles, provedores para JAXB, para quando o tipo de conteúdo trafegado for do tipo xml (application/xml, text/xml e application/*+xml). Ainda na classe PessoaBinding da Listagem 16, poderíamos então, no nosso exemplo, criar um serviço RESTful cujo retorno seja a classe JAXB PessoaBinding. Neste caso a declaração do serviço seria similar ao método da Listagem 18.

Listagem 18. Serviço RESTful cujo retorno é uma classe JAXB.

```
@GET
@Produces("application/xml")
@Path("/Mundoj/{idPessoa}/")
public PessoaBinding getPessoa(@PathParam("idPessoa") Integer id) {
   return dao.getPessoaAsBinding(id); // Retorna uma entidade Pessoa
                                      // como PessoaBinding
}
```

Ao fazermos o consumo deste serviço RESTful, vamos perceber que a conversão é feita automaticamente pelo entity provider padrão para XML. De maneira inversa poderíamos criar um serviço RESTful para receber requisições PUT e receber como parâmetro de entrada do método a classe PessoaBinding via HTTP Body. Conforme apresenta a Listagem 19.

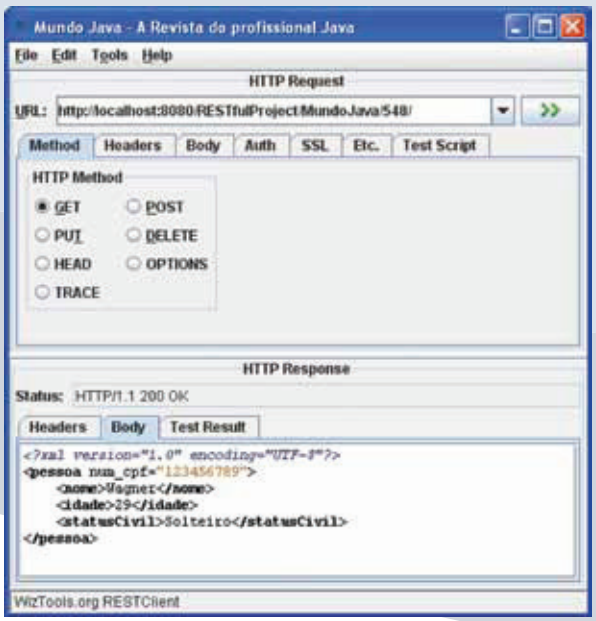

Figura 7. Retorno do serviço RESTful cujo retorno é uma classe JAXB.

Listagem 19. Convertendo código XML para objeto JAXB em chamada PUT com REST.

```
@PUT
@Consumes("application/xml")
@Path("/Mundoj/ ")
public void putPessoa(PessoaBinding pessoa) {
   // Operação de update
}
```

JAKARTA COMMONS – HTTPCLIENT

HttpClient é um subprojeto open source da Jakarta Commons que se tornou independente em 2007, e que foi concebido para facilitar o desenvolvimento de aplicações que utilizam o protocolo HTTP.

Ele é um projeto escrito totalmente em Java, e implementa todos os métodos HTTP (GET, POST, PUT, DELETE, HEAD, OPTIONS e TRACE).

Possui suporte ao protocolo HTTPS, suporte ao gerenciamento de conexões para uso em aplicações multithread, suporte a cookie, possui mecanismos de autenticação Basic, Digest e criptografia NTLM.

Na Listagem 20, demonstramos o uso da biblioteca HttpClient, no qual consumimos dois serviços RESTful, um com uma chamada GET e outra com uma chamada PUT.

Listagem 20. Consumindo serviços REST via GET e PUT com HTTPClient.

```
1 public void testHTTPClient() {
2   try {
3   HttpClient client = new HttpClient(new /multiThreadedHttpConnectionManager());
4   client.getHttpConnectionManager().getParams().setConnectionTimeout(30000);
5   final String CONTENT_TYPE = "application/xml";
6   final String CHARSET = "UTF8";
7
8   /* Executando chamada com método HTTP GET */
9   String getURI = "http://localhost:8080/TestesWS/Mundoj/autores/
      Wagner/?idade=29";
10  GetMethod get = new GetMethod(getURI);
11  Header meuHeader = new Header("CPF","123456789");
12  get.setRequestHeader(meuHeader);
13  int statusCodeGET = client.executeMethod(get);
14  String responseBody = get.getResponseBodyAsString();
15  System.out.println("Chamada GET");
16  System.out.println(" Status Code: "+statusCodeGET+
      " \nResponse Body:\n"+responseBody);
17
18  /* Executando chamada com método HTTP PUT */
19  String putURI = "http://localhost:8080/TestesWS/Mundoj/autores/
      update/";
20  PutMethod put = new PutMethod(putURI);
21  StringRequestEntity requestEntity = new StringRequestEntity(
      responseBody, CONTENT_TYPE, CHARSET);
22  put.setRequestEntity(requestEntity);
23   int statusCodePUT = client.executeMethod(put);
24  responseBody = put.getResponseBodyAsString();
25  System.out.println("Chamada PUT");
26  System.out.println(" Status Code: "+statusCodePUT+
      " \nResponse Body:\n"+responseBody);
27  } catch (Exception ex) {/* OMITIDO */}
28 }
```

Retorno da chamada ao método

Chamada GET

Status Code: 200

Response Body:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <pessoa num_cpf="123456789">
    <nome>Wagner</nome>
    <idade>29</idade>
    <statusCivil>Casado</statusCivil>
  </pessoa>
```

Chamada PUT

Status Code: 202

Response Body:

```
<html>
  <body><h1>Bem vindo Wagner</h1></body>
</html>
```

Primeiro, na linha 3, instanciamos a classe HttpClient que é o nosso agente HTTP que irá conter os atributos de persistência com cookies, e credenciais de autenticação através da classe HttpState. E também onde será armazenado uma ou mais conexões HTTP, em que faremos chamadas aos métodos HTTP.

Na linha 4 atribuímos um timeout para a conexão de 30 segundos. Depois nas linhas 5, 6 e 9 declaramos as variáveis quer irão determinar o tipo de conteúdo, o character set e a URI de acesso ao serviço REST.

Na linha 10, instanciamos a classe GetMethod que, como o próprio nome indica, representa o método GET, passando como parâmetro a URL do nosso serviço RESTful (getURI). Na linha 11 criamos um objeto Header, passando como parâmetro no construtor a chave e o valor que representam o parâmetro e o valor do cabeçalho, no nosso exemplo, passamos um número fictício de CPF. Na linha 12 atribuímos o objeto header para o objeto GetMethod.

Na linha 13, fazemos a chamada ao serviço RESTful via HTTP GET, e armazenamos o código de status do retorno na variável statusCodeGET. Na linha 14, extraímos os dados da Response como String para a variável responseBody. Pelo fato do retorno ser em XML, poderíamos facilmente utilizar JAXB para trabalhar o retorno como um objeto Java. Finalmente nas linhas 15 e 16 imprimimos no console o retorno da chamada a estes métodos.

A partir da linha 18, iniciamos o mesmo processo, mas agora para efetuar uma chamada via método PUT, as únicas diferenças são o uso do método PutMethod, que implementa o método HTTP PUT e o uso da classe StringRequestEntity na linha 21, que passamos como parâmetro de construtor o xml de retorno da chamada que fizemos ao serviço REST GET, o tipo de conteúdo (CONTENT_TYPE) e o charset. Com esta classe, atribuímos uma entidade (como String) ao método PUT que será enviado junto a requisição.

Nas linhas 25 e 26 imprimimos o retorno da requisição PUT, imprimindo o valor da tag nome do código xml enviado.

JavaScript

Graças ao objeto XMLHttpRequest, conseguimos nos comunicar com servidores de forma assíncrona, desde então temos todas as vantagens do AJAX ao nosso dispor. Para quem desenvolve interfaces WEB, este recurso resolveu grandes problemas no lado do cliente, mas vale lembrar que JavaScript não é Java, não possui threads, nem tipos, e possui uma grande gama de frameworks Ajax, como, por exemplo, Prototype, JQuery, Dojo, Script.aculo.us, Ext-JS, entre outros.

Na Listagem 21, temos um exemplo de uma função em JavaScript que consome um serviço RESTful cujo retorno é um XML.

Listagem 21. Consumindo um serviço RESTful (retorno XML) com Ajax.

```
function showCustomer(str){
   xmlHttp=GetXmlHttpObject(); // omitido código do método
   if (xmlHttp==null) {
      alert("Your browser does not support AJAX!");
      return;
   }
   var url=&apos;http://localhost:8080/ ContatosMJ/resources/
customers/58/&apos;;
   xmlHttp.onreadystatechange=stateChanged;
   xmlHttp.open(&apos;GET&apos;,url,true);
   xmlHttp.send(null);
}

function stateChanged() {
   if (xmlHttp.readyState==4){
      var xmlDoc=xmlHttp.responseXML.documentElement;
      document.getElementById("nome").innerHTML=
       xmlDoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
   }
}
```

Na Listagem 21, vimos um exemplo de um serviço que retorna XML, mas uma das grandes vantagens dos serviços REST, é que podemos trabalhar com diversos formatos para troca de informação de um mesmo recurso. Entre eles, JSON.

jQuery

jQuery é uma biblioteca JavaScript que vem chamando atenção por conta de sua facilidade de desenvolvimento. Ela simplifica muito a manipulação dos elementos de um documento HTML, o tratamento de eventos e as interações Ajax para prover um desenvolvimento rápido de aplicações web. Com seu uso, o desenvolvedor se livra de preocupações relacionadas à compatibilidade de navegadores e aderência à CSS.

A biblioteca jQuery fornece algumas funções para tratamento de requisições Ajax, ideais para o consumo de serviços REST, que reduzem muito a complexidade e a quantidade de linhas de código. Com a função $.ajax() do jQuery, conseguimos um alto nível de controle nas requisições ajax.

A sintaxe do comando é $.ajax(options), em que o parâmetro options são as propriedades que passamos para controlar como a requisição é feita e o retorno da chamada.

Na Listagem 22, demonstramos o uso das funções $.ajax().

Listagem 22. Consumindo um serviço REST com a função $.ajax().

```
$.ajax({
   type: "DELETE",
         url: "http://localhost:8080/Mundoj/autores/"+idAutor+"/",
         success: function(msg){
      $("#alert").html(msg);
   }
});
```

Na Listagem 22, usamos dois parâmetros na função $.ajax(), o parâmetro type para indicar o método HTTP que queremos executar e a url de chamada. Para tratar tipos de retorno JSON, o jQuery oferece a função $.getJSON(), utilizada para carregar dados JSON mediante uma requisição HTTP GET.

Na Listagem 23 mostramos um exemplo de uso da função $.getJSON() em um serviço REST do Flickr. Nessa listagem, fazemos uma chamada ao serviço REST e passamos o retorno da chamada ao método de callback. Dentro da função de callback, criamos a tag <img> passando como valor o endereço da foto retornada pelo serviço REST e a incluímos na div #foto. Note que a variável data é um map chave-valor dos dados retornados pela função REST.

Listagem 23. Uso da função $.getJSON para consumo de dados no formato JSON.

```
$.getJSON("http://api.flickr.com/services/rest/?method=flickr.photosets.
getPhotos& photoset_id=72157614488723406&format=json&jsoncallback=?",
  function(data){
    $.each(data.photoset.photo, function(i,item){
      if (item.title == foto){
        $("<img/>").attr("src", "http://farm"+item.farm+".static.
        flickr.com/"+item.server+"/"+item.id+"_"+item.secret+"_m.jpg").
        appendTo("#foto");
      }
    });
});
```

Considerações finais

Neste artigo, procuramos abordar REST de uma maneira simples e objetiva, de modo que tanto os que já possuem conhecimento em REST quanto os que estão iniciando tenham proveito.

Assim como toda nova tecnologia, JAX-RS não é uma "bala de prata", mas sabendo o momento de usá-la se torna uma ferramenta poderosa de integração, em termos de facilidade no desenvolvimento e requerer uma infraestrutura mais leve dispensando o uso de um middleware WS-*.

Falamos sobre a especificação JAX-RS com diversos exemplos práticos, porém é válido lembrar que existem diversas implementações da JAX-RS que possuem funcionalidades adicionais não contempladas neste artigo.

Descrevemos diversas maneiras sobre como consumir e testar serviços RESTful, obviamente as soluções apresentadas neste artigo não são as únicas disponíveis no mercado, ficando este artigo como um guia de opções• MJ

Referências

- *Livro: RESTful Web Services por Leonard Richardson e Sam Ruby (Author).*
- *Livro: HTTP – The Definitve Guide por Dayid Gourley e Brian Totty.*
- *http://jcp.org/en/jsr/detail?id=311 – JCP 311 para a API JAX-RS*
- *https://jersey.dev.java.net/ – Site do projeto Jersey.*
- *http://tools.ietf.org/html/rfc3986 – Sintaxe Genérica definida para uma URI.*
- *http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm – Capítulo 5 da dissertação de Roy Fielding sobre REST*
- *https://wadl.dev.java.net/ – Site do projeto WADL no Glassfish.*
- *https://jaxb.dev.java.net/ – Site do projeto JAXB*
- *http://curl.haxx.se – Ferramenta cURL*
- *http://code.google.com/p/rest-client/ – Site do projeto RESTClient*
- *http://tools.ietf.org/html/rfc2616 – Especificação HTTP 1.1*
- *http://www.webdav.org/specs/rfc2518.html#http.methods.for – Extensões HTTP – WebDAV*
- *http://jquery.com/ – Site oficial do jQuery.*