WEB SERVICES SOAP E REST

The SOAP and REST web services

Daniel Adorno GOMES
Faculdade Politécnica de Campinas

Peter JANDL JR
Faculdade Politécnica de Campinas
Faculdade de Jaguariúna

Resumo: Uma das tecnologias que mais se discutem atualmente são os w*eb services*. Isso ocorre principalmente porque os w*eb services* são o meio mais apropriado para a implementação de um inovador conceito de arquitetura que envolve tanto tecnologia quanto negócios. Está-se falando da SOA, *Service Oriented Architecture* ou Arquitetura Orientada a Serviços, outro assunto muito abordado nos dias de hoje. Mas, voltando ao tema dos w*eb services*, a questão a ser apresentada diz respeito a todo esse foco que está sendo direcionado aos w*eb services*, o qual fez surgir uma discussão quentíssima em torno destes: qual a melhor maneira ou o melhor padrão para se implementar *web services*? Quando se busca uma resposta para essa questão, basicamente se está colocando em pauta se a melhor maneira para se criar w*eb services* é seguindo o padrão SOAP ou o padrão REST.

Palavras-chave: SOAP, REST, web services.

Abstract: One of the technologies that are most discussed currently is the web service. This is happening mainly because the web service is the most appropriate means for the implementation of an innovative concept of architecture that involves both technology and business. We are talking about SOA, Service Oriented Architecture, another much discussed topic nowaday. But back to the issue of web services, the question that I would submit, relates to all this focus being directed to web services, which gave rise to a discussion around these theme: what is the best or the best method to implement web services? When we seek an answer to this question, basically we are putting on the work if is the best way to create web services is following the standard SOAP or REST.

Key-words: SOAP, RESP, web services.

INTRODUÇÃO

Quando se busca no Google algo relacionado à *web services* SOAP e REST, é muito comum encontrar artigos, notícias, discussões em fóruns, normalmente com o seguinte título: “SOAP versus REST” ou “SOAP x REST”, etc. Para fugir desse clichê,

optou-se por desenvolver o artigo de uma forma que não confrontasse esses dois padrões, evitando produzir a idéia de se definir qual deles é o melhor caminho para o desenvolvimento de *web services*. SOAP pode ser o melhor para o sistema X e REST para o sistema Y, dependendo das características de cada sistema e do que se deseja garantir a eles.

A intenção aqui é apresentá-los de forma clara e imparcial, exibindo suas propostas de arquitetura, como eles realmente funcionam, seus pontos fortes e fracos, e, acima de tudo, mostrar os padrões de forma didática. Tudo isso, para que o leitor possa decidir qual desses padrões é o mais adequado para a criação de *web services*, levando-se em conta o domínio do problema que deve ser informatizado.

O SURGIMENTO DOS PADRÕES

Diante da perspectiva do modelo de computação distribuída, na segunda metade da década de noventa, deparou-se com o surgimento de várias tecnologias como o RMI, o DCOM e o CORBA, que foram bem sucedidas na integração de aplicações em ambientes de redes locais e "homogêneos".

Com a expansão da Internet para o mercado corporativo, surge a necessidade de integração de aplicações, além das redes locais e em ambientes heterogêneos. A solução encontrada nesse momento da história da computação, já em meados da década de noventa, foi a utilização de aplicações Web (JSP, ASP, PHP), juntamente com XML. Porém, como não havia um padrão para o desenvolvimento de tais aplicações, cada empresa criava seus próprios protocolos e cada protocolo possuía suas próprias definições para questões de robustez, segurança, transações, etc, tornando assim complicada a interação entre aplicações que haviam sido criadas para serem interoperáveis.

Diante desse contexto, grandes empresas como IBM, Microsoft, BEA, entre outras participantes do W3C, decidem padronizar a implementação desse tipo de aplicação, objetivando uma total interoperabilidade entre elas, iniciando assim a criação do que conhecemos como conjunto de padrões WS-* (lê-se WS asterisco). Que

constituem os padrões para o desenvolvimento de *web services* baseados no protocolo SOAP.

A partir desse momento, os pesquisadores do W3C começaram a criar especificações para *web services*, envolvendo todo tipo de padronização que julgavam interessante para essa tecnologia. Isso fez com que surgisse uma quantidade enorme de especificações para *web services*, pois eram criados documentos de padronização para situações que nunca haviam sido implementadas ou se mostrado necessárias até aquele momento, a não ser teoricamente. O ponto interessante é que, para praticamente todos os tipos de problemas que possam surgir, já existe uma especificações SOAP criada. A Tabela 1 lista algumas das especificações para *web services*, criadas pelo W3C.

Tabela 1. Especificações do conjunto WS-*.

Especificação	Descrição
WS-Addressing	Especifica mecanismos (ou protocolos) de transporte que permitem aos web services efetuarem trocas de mensagens.
WS-Eventing	Define um protocolo que permite a troca de mensagens entre web services. Dessa forma, um web service pode tanto enviar como receber mensagens de outros web services.
WS-Security	Protocolo que fornece meios para a aplicação de segurança aos web services.
WS-SecureConversation	Proposta que visa permitir conversas seguras entre sites que utilizam web services para se comunicar.
WS-Trust	É uma especificação que provê extensões para o WS-Security, lidando especificamente com emissão, renovação, e validação de tokens de segurança, assim como formas de estabelecer, avaliar a presença de, e manter relações de confiança entre os participantes de uma troca segura de mensagens.
WS-ReliableMessaging	Descreve um protocolo que permite que mensagens SOAP sejam entregues de maneira confiável entre aplicações distribuídas na presença de falhas de software, sistema ou rede.
WS-Coordination	Define um framework extensível para prover protocolos que coordenam as ações de aplicações distribuídas, incluindo transações distribuídas.
WS-AtomicTransaction	Define dois tipos de coordenação de transações: Atomic Transaction (AT) para operações individuais e o Business Activity (BA), para transações de longa duração.
WS-Policy	Especificação que permite aos web services usar XML para publicar suas políticas (políticas de segurança, qualidade de serviços, etc.), e para consumidores de web services especificarem suas políticas.

Agora, fica mais compreensível o porquê da denominação WS-*, citado anteriormente. O asterisco é utilizado nesse caso como um caractere curinga para representar as inúmeras especificações criadas pelo W3C para os *web services* baseados no protocolo SOAP.

No caso do REST, pode-se dizer que o seu surgimento percorreu um caminho inverso ao do padrão SOAP.

Antes de tudo, é conveniente salientar que, quando o padrão REST foi proposto, no ano de 2000, o padrão SOAP já existia. Portanto, o REST constituía uma forma diferente, uma maneira alternativa de se desenvolver *web services*.

O REST ou *Representional State Transfer* foi proposto por Roy Fielding (2000), um dos autores do protocolo HTTP, em sua tese de doutorado. Aqui se tem outro ponto importantíssimo. O padrão REST não foi proposto pelo W3C, nasceu de uma tese de doutorado. Dessa maneira, ao contrário do SOAP, não foi criada uma quantidade enorme de especificações para o protocolo REST, aliás, não foi criada nenhuma especificação.

Em sua tese, Fielding (2000) propunha que os w*eb services* fossem implementados com base na utilização dos recursos oferecidos pelo protocolo HTTP. O que havia de fato era um conjunto de regras que mostravam como os w*eb services* baseados na proposta de Fielding deveriam ser implementados. Assim, conforme a abordagem REST ia chegando ao conhecimento dos profissionais e empresas de TI, novas implementações iam surgindo com base nessa linha de desenvolvimento de *web services*. Os casos de sucesso foram surgindo, e com eles veio também a necessidade de padronizar, de tornar as idéias reutilizáveis. Enfim, pode-se dizer que a evolução do padrão REST se deu de uma forma “natural”, ou seja, bastante gradativa em relação ao padrão SOAP. Pois, as normas e padronizações foram sendo criadas à medida que as experiências práticas expunham essa necessidade. Diferentemente do SOAP, que possui muitas especificações que foram criadas com base no que foi “imaginado” que seria uma necessidade para o mercado.

A seguir, serão expostos os dois caminhos para a criação de *web services*. Mas antes de continuar, é importante fechar esse tópico com o esclarecimento de que o padrão REST é uma arquitetura *web*, uma forma alternativa de se criar *web sevices*. Já o conjunto de especificações WS-*, ou *web services* baseados no padrão SOAP, constituem um padrão W3C.

COMO FUNCIONAM OS WEB SERVICES SOAP

Na Figura 1 é apresentado um esquema que ilustra quais são os componentes envolvidos numa chamada a um *web service* SOAP e a sequência em que essa chamada acontece:

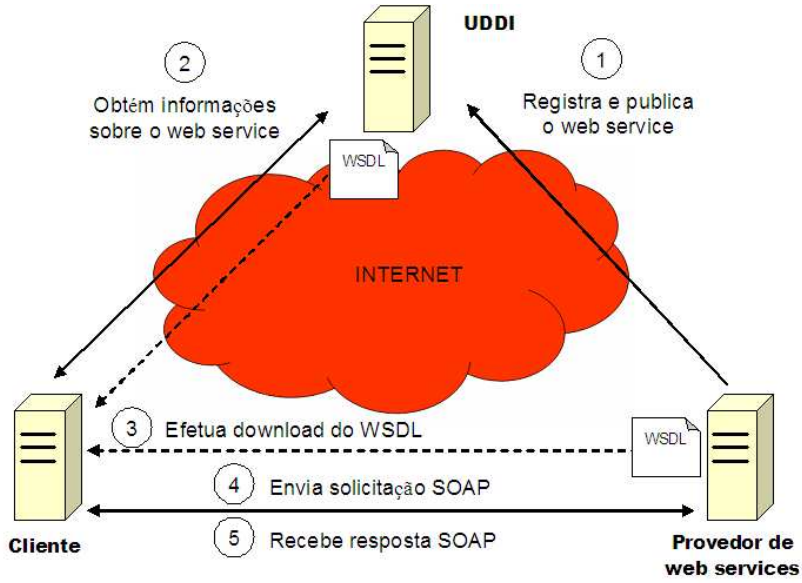

Figura 1. Arquitetura dos *web services* SOAP especificada pelo W3C.

Primeiramente, serão descritos os componentes envolvidos nesse esquema, individualmente:

- **SOAP** (*Simple Object Access Protocol*): protocolo padrão para transmissão de dados quando se fala em *web services* pertencentes ao conjunto de especificações WS-*. Baseado no XML, o SOAP segue o modelo REQUEST-RESPONSE do HTTP. Ele é um dos pilares de sustentação dessa arquitetura criada pelo W3C. É tão importante que muitas vezes os web services do padrão WS-* são referidos como “web services SOAP”.

- **WSDL**: (*Web Services Description Language*): arquivo, padrão XML, que tem por finalidade fornecer uma descrição detalhada do *web service*, ao solicitante ou cliente. Essa descrição envolve a especificação das operações que compõem o serviço, definindo claramente como deve ser o formato de entrada e saída de cada operação. Como mostrado na figura 1, o WSDL pode tanto estar armazenado no Provedor de *web services*, quanto no UDDI.

Listagem 1. Trecho de um arquivo WSDL

```
<types>
<xsd:schema>
<xsd:import namespace="http://webservices.soap.ws.policamp/"
schemaLocation="http://localhost:8080/ContasAPagarWSService/ContasAPaga
rWS?xsd=1" />
</xsd:schema>
</types>
<message name="inserirConta">
<part name="parameters" element="tns:inserirConta" />
</message>
<message name="inserirContaResponse">
<part name="parameters" element="tns:inserirContaResponse" />
</message>
<message name="excluirConta">
<part name="parameters" element="tns:excluirConta" />
</message>
<message name="excluirContaResponse">
<part name="parameters" element="tns:excluirContaResponse" />
```

- **UDDI** (*Universal Description, Discovery and Integration*): mecanismo que atende tanto ao cliente de *web services*, quanto ao provedor. Ao provedor de *web services* o UDDI tem que fornecer recursos para que os *web services* sejam registrados e publicados, para que dessa forma, possam ser pesquisados e localizados pelos clientes de *web services*. O UDDI também pode ser utilizado para o armazenamento de arquivos WSDL.
- Cliente: é um consumidor de *web service*, ou seja, um software que irá utilizar as operações de um determinado *web service*. Porém, na figura anterior é importante ressaltar que o cliente está representando várias etapas do ciclo de vida desse software. Desde sua pré-existência, quando o arquivo WSDL é obtido por um desenvolvedor, até o momento em que o

software já está operando, onde ele faz solicitações e recebe os resultados dos *web services*.

- Provedor de *web services*: é possível ser bastante claro quanto a esse componente. Ele é um *application server* ou um *web container*, dependendo do caso onde o *web service* ficará armazenado. Como se pode verificar no esquema mostrado anteriormente, ele pode armazenar também os arquivos WSDL.

Nos itens seguintes, discute-se como se dá a integração dos componentes descritos na Figura 1. Porém, antes de prosseguir com a explicação, é essencial observar que toda essa arquitetura tem como base a linguagem XML. Portanto, todas as informações trocadas entre qualquer uma das partes são enviadas e recebidas através de mensagens no padrão XML. A seguir, é explicitada a sequência de operações realizadas por cada um dos componentes da arquitetura, utilizando os números referenciados na Figura 1.

1. **Registra e publica o *web service*:** o provedor é o local onde os *web services* ficam armazenados, juntamente com os seus respectivos descritores, os arquivos WSDL. Quando um determinado *web service* é criado, ele é disponibilizado para utilização no provedor. Porém, para que ele possa ser utilizado por algum cliente, ele e o seu WSDL precisam ser localizados, ou seja, o cliente precisa saber qual o endereço do serviço ou o seu URI (*Uniform Resource Identifier*). Dessa forma, após a criação e armazenamento de um *web service* no provedor, ele deve ser registrado e publicado num diretório de registro de *web services* ou UDDI.
2. **Obtém informações sobre o *web service*:** um cliente de *web services*, quando necessita utilizar um determinado serviço, primeiramente irá pesquisar em diretórios de registro de *web services* (UDDI), pelo tipo de serviço desejado. Por exemplo, um *web service* que retorne a cotação do dólar. Os recursos de pesquisa e de localização de *web services* foram incluídos na arquitetura do W3C, pois diversos *web services* de fornecedores de software diferentes podem disponibilizar a mesma operação. Sendo assim, o cliente precisa obter a informação que dirá onde

está o *web service* que ele deseja utilizar e o seu respectivo arquivo WSDL. Traduzindo, o UDDI irá fornecer o endereço (URI) do *web service* e do seu WSDL.

3. **Efetua download do WSDL:** após a obtenção dos URI's do *web service* e do seu descritor (WSDL), o cliente poderá efetuar o download do arquivo WSDL e prosseguir com a utilização do *web service* desejado. Não se deve esquecer que o arquivo WSDL pode estar disponível para download tanto no provedor de *web services*, quanto no UDDI. A partir da obtenção do arquivo WSDL e do URI do *web service*, um desenvolvedor terá condições de criar um software cliente que irá fazer uma chamada ao *web service* em questão e, em seguida, obter uma resposta.
4. **Envia solicitação SOAP:** com o desenvolvimento do software cliente, este irá enviar solicitações no padrão SOAP ao *web service*, referenciando o serviço através do seu URI.
5. **Recebe resposta SOAP:** após o envio da solicitação SOAP ao *web service*, normalmente o software cliente irá receber uma resposta, também no padrão SOAP, como resultado da solicitação anterior.

A arquitetura descrita anteriormente mostra o funcionamento do que foi "especificado" pelo W3C. Porém, como já citado no início do texto, todas essas especificações foram criadas na teoria, sem ter por base nenhuma experiência prática. O ponto onde se pretende chegar é o seguinte, no "mundo real" a utilização dos *web services* não se dá exatamente como foi descrito até aqui, apesar de ser perfeitamente possível. O que ocorre é que normalmente não se tem a figura do UDDI, de forma que a comunicação se dá entre o cliente e o provedor de *Web sevices*, sem intermediações.

Figura 2. Arquitetura dos *web services* SOAP utilizada no “mundo real”.

Outro fator importante, que deve ser ressaltado com relação ao que foi projetado e o que de fato acontece no “mundo real”, refere-se ao protocolo sobre o qual as mensagens do tipo SOAP podem ou devem trafegar. As mensagens enviadas aos *web services* SOAP ou emitidas por eles são independentes do protocolo usado para transportá-las. Ou seja, podem trafegar sobre HTTP, SMTP, FTP, TCP puro ou qualquer outro protocolo. Essa característica deve-se ao fato de que as mensagens SOAP são auto-contidas, todas as informações necessárias para que um *web service* desse tipo possa executar um processamento estão dentro de um único arquivo XML, garantindo essa independência e permitindo que as mensagens SOAP sejam enviadas, por exemplo, por e-mail ou softwares de mensagens instantâneas. No entanto, o que se vê na prática é a combinação SOAP + HTTP. Pode-se dizer que praticamente 100% das implementações SOAP profissionais (excluindo aqui as experiências) são feitas sobre o protocolo HTTP. A preferência pelo HTTP tem motivos óbvios. Esse protocolo de transporte domina praticamente toda a Web, sendo suportado por praticamente todo tipo de plataforma de software/hardware, há muito tempo. Portanto, muito maduro, confiável e sem problemas com restrições por firewalls.

O envio de mensagens de um cliente de *web services* SOAP (*Sender*) para um *web service* SOAP (*Receiver*) propriamente dito pode ocorrer de duas maneiras:

- **One-Way Messaging:** também conhecida como *fire-and-forget*, é uma forma de envio de mensagens unilateral, onde o cliente envia a mensagem sem se preocupar com o retorno, conforme Figura 3. O *web service* irá executar um determinado processamento e não enviará um retorno ao cliente.

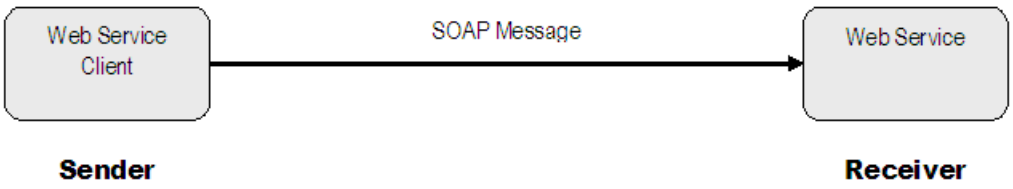

Figura 3. One-Way Messaging (Fire and Forget).

- **Resquest-Response Messaging:** é uma forma de envio de mensagens que se assemelha ao formato HTTP, ou seja, é bilateral, ocorrendo o envio de mensagens ao *web service* por parte do cliente, um processamento qualquer, e o retorno, resultante deste processamento, que será enviado ao cliente pelo *web service*. A Figura 4 ilustra o processo. Esse tipo de envio de mensagens pode ser síncrono ou assíncrono.

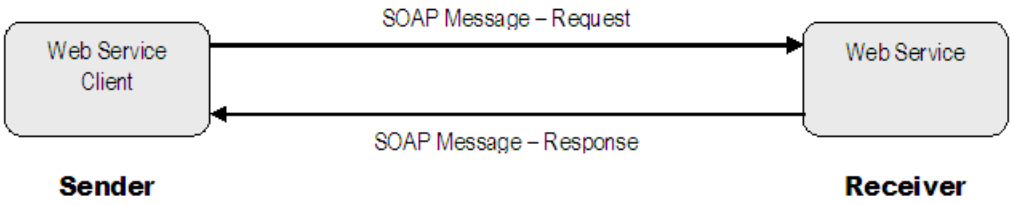

Figura 4. Resquest-Response Messaging.

Para o desenvolvimento de *web services* SOAP, conta-se com uma grande quantidade de *frameworks* Java como o Apache Axis, Apache Axis2, Java WS Dev Pack e XFire. Os principais IDE's utilizados, como Eclipse e Netbeans, também contam com ferramentas que, baseadas nos frameworks citados, ajudando a criar *web services* SOAP de maneira bastante ágil. É importante destacar esse ponto, pois, atualmente, seria muito improdutiva a criação de tais *web services* sem a ajuda de qualquer um desses recursos.

No paradigma SOAP, cada serviço corresponde a uma atividade ou tarefa, normalmente um processamento enxuto e bem definido, como por exemplo, "Verificar CPF", "Calcular Imposto de Renda" ou "Obter Cotação do Dólar", etc.

Tecnicamente, na linguagem Java, cada serviço equivale a um método de uma determinada classe, também chamado de “*web method*”. Uma classe pode conter um ou mais “*web methods*”.

COMO FUNCIONAM OS WEB SERVICES REST

Descrevendo de forma gráfica a comunicação entre os componentes envolvidos numa chamada à *web services* do padrão REST, como na Figura 5, nota-se facilmente que se trata de uma arquitetura de comunicação mais simples do que a arquitetura proposta pelo W3C (Gregório, 2009A).

Figura 5. Arquitetura dos *web services* REST.

Normalmente, os componentes envolvidos são o “Cliente de *web services*”, o “Provedor de *web services*” e o protocolo “HTTP”. Pode-se considerar um componente a mais para os *web services* REST, denominado “*Web Application Description Language*“ (WADL). Esse recurso é equivalente ao WSDL para os *web services* do padrão SOAP. Porém, pelo fato de ser um recurso relativamente novo e ainda com pequena adesão, neste artigo não será considerado como parte dessa arquitetura.

Com esse cenário, basicamente tem-se o cliente enviando mensagens de solicitação a um determinado *web service* disponível no provedor. O *web service* em questão irá realizar um determinado processamento e retornar uma mensagem de resposta ao cliente solicitante. O protocolo que determina o formato das mensagens enviadas e recebidas é o HTTP, o qual, nesse caso, também será o protocolo de transporte das mensagens.

A diferença entre os protocolos que ditam o formato das mensagens SOAP e REST será exposto mais adiante. Agora, discute-se como são arquitetados os serviços REST.

No paradigma REST os serviços correspondem à recursos. Cada recurso corresponde a uma URI que deve ser única e deve estar armazenada em um único local.

É um pouco complexo entender essa visão, pois, em geral, tem-se em mente que os serviços são atividades e não recursos, o que condiz com a maneira de se enxergar os serviços do ponto-de-vista SOAP, que é mais familiar até pelo seu tempo de existência.

Para facilitar, a explicação é colocada em termos práticos. Por exemplo, convém relembrar do velho amigo “Sistema Comercial”. Como é de praxe, esse tipo de sistema tem que contar com informações sobre clientes, produtos, funcionários, vendas, etc. Mas, da ótica dos *web services* REST, o que seria um recurso dentro desse contexto? Em primeiro lugar, limita-se aqui a utilização dos “clientes” para a explicação, deixando de lado os demais componentes de um sistema com esse perfil.

Para um sistema comercial baseado na arquitetura REST, cada “cliente” corresponde a um recurso, ou seja, para cada cliente tem-se uma URI. É possível imaginar que nesse sistema os clientes sejam identificados unicamente por um código. O código que identifica o cliente “João da Silva Moreira” é o código “0000125”. Nesse caso a URI que o identificaria poderia ser: **http://www.sistemacomercial.com.br/cliente/0000125**.

Traduzindo para o idioma REST, a URI acima identifica o recurso “cliente” correspondente ao código “0000125”. Querendo executar qualquer operação para a manipulação dos dados desse cliente, tem-se que, obrigatoriamente, utilizar essa URI.

Outra situação que se poderia imaginar seria a listagem de todos os clientes cadastrados. Nesse caso, também tem-se que enxergar esse conjunto composto por todos os clientes, como um novo recurso e, portanto, ele também teria que possuir uma

URI que o identificasse unicamente. Por exemplo: **http://www.sistemacomercial.com.br/clientes**.

Essa estruturação provém do fato dos *web services* REST serem totalmente baseados no HTTP, em seus métodos e em seus códigos de retorno.

A princípio, o HTTP define como os dados que trafegam entre os *web services* e seus solicitantes serão “envelopados”, o que será visto mais adiante. Os métodos do HTTP correspondem às operações para a manipulação de dados relacionada a cada “recurso” de um determinado sistema. E os códigos de retorno do HTTP funcionam como um resultado da execução das operações de cada recurso.

Na Tabela 2 são apresentados os métodos do HTTP e a operação relacionada a cada um deles. Na Tabela 3 os principais códigos de retorno do HTTP mais utilizados em aplicações REST.

Tabela 2. Métodos do HTTP.

Método	Descrição
GET	usado para obter um recurso ou uma lista deles
POST	usado para incluir um recurso
PUT	usado para editar um recurso
DELETE	usado para excluir um recurso

No momento em que se projeta um sistema baseado na arquitetura REST, primeiramente define-se os recursos envolvidos, posteriormente os métodos relacionados a cada um dos recursos, suas representações que farão parte da URI e, por fim, quais são os códigos HTTP retornados, que serão considerados para cada um dos métodos (Gregório, 2009A). A Tabela 4 mostra como ficariam essas definições para os recurso “cliente”, que corresponde à um único cliente de um sistema comercial, e para o recurso clientes, que corresponde ao conjunto de todos os clientes desse sistema.

Tabela 3. Códigos de retorno do HTTP.

Código	Significado
200	Sucesso
201	Criado
204	Sem conteúdo
400	Requisição inadequada
401	Não autorizado
403	Acesso negado
404	Não encontrado
412	Falha na pré-condição
500	Erro Interno no servidor
501	Não implementado

Tabela 4. Mapeamento de recursos, métodos e códigos de retorno do HTTP.

Recurso	Método	Representação	Retorno	Significado
Cliente	GET	cliente	200	Recurso encontrado e retornado com sucesso
			404	Recurso não encontrado
	PUT	cliente	200	Recurso encontrado e atualizado com sucesso
			400	Requisição com formato inválido
			404	Recurso não encontrado
	DELETE		200	Recurso encontrado e excluído com sucesso
			404	Recurso não encontrado
	POST	cliente	201	Recurso criado com sucesso
			400	Requisição com formato inválido
Clientes	GET	clientes	200	Recurso encontrado e retornado com sucesso
			404	Recurso não encontrado

Um ponto importante a ser destacado sobre os métodos do HTTP é que todos eles, com exceção do POST, são idempotentes. Ou seja, se uma operação GET, PUT ou DELETE não for adiante ou se não se souber se a operação em questão seguiu ou não adiante, o procedimento correto é enviar a requisição novamente. Nesse caso, mesmo que várias requisições idênticas sejam feitas em sequência, isto não deve causar danos ao servidor.

Com relação à construção de *web services* REST, o ideal é contar com o auxílio de *frameworks*, como o *Restlet*, que permite mapear conceitos REST para classes Java. Outro exemplo mais recente é a API JAX-RS, Java API for RESTful *Web Services* (JCP, 2009), e sua implementação de referência, o projeto Jersey. Basicamente, através dessa nova API, pode-se criar *web services* REST através de classes, onde uma classe corresponde a um recurso REST e cada método dessa classe equivale a um dos métodos do HTTP.

PONTOS FORTES E FRACOS ENTRE OS WEB SERVICES SOAP E REST

Os *web services* SOAP são utilizados, em sua maioria, com o protocolo “HTTP POST”, para o envio e recebimento de suas mensagens, as quais constituem arquivos no padrão XML. Nesse caso, o HTTP serve única e exclusivamente para transportar as mensagens XML. Já os *web services* REST são totalmente baseados nos recursos oferecidos pelo HTTP, ou seja, o HTTP constitui ao mesmo tempo o protocolo de transporte da mensagem, assim como o próprio recipiente dessa mensagem, podendo ser utilizado com qualquer um dos seus métodos GET, PUT, DELETE e POST, permitindo também a utilização de *cachê* (Gregório, 2009B).

A principal discussão em torno desses dois padrões para construção de *web services* é exatamente a grande quantidade de recursos consumida pelos *web services* SOAP, por causa do peso do XML em relação aos *web services* REST que, para a obtenção dos mesmos resultados, consomem uma quantidade de recursos muito menor, pois, o *parsing* XML nesse caso é bem menor ou mesmo nenhum, dependendo do *Content-Type* definido nas mensagens. Pode-se definir qualquer uma das seguintes opções de *Content-Type* para *web services* REST:

- *application/xml*
- *applicatio/json*
- *text/plain*
- *text/xml*
- *text/html*

Para se compreender melhor essa questão, ou seja, como ocorre o envio de uma mensagem a um *web service*, e o retorno de uma mensagem de resposta, gerada por esse *web service*, ao cliente solicitante. Obviamente, serão explicados nos dois padrões, SOAP e REST, para que se possa analisar o "esforço" que tem de ser realizado para a execução do serviço em cada um dos casos. O HTTP é um protocolo orientado à mensagens, ou seja, as solicitações e as respostas HTTP são "mensagens". Define-se uma mensagem como sendo um envelope com um conteúdo dentro. No caso do REST, o envelope onde será colocado o conteúdo que se quer enviar a um *web service* é o corpo de uma mensagem HTTP. Dessa forma, o conteúdo da mensagem viaja no próprio corpo do protocolo de transporte da mensagem. A Figura 6 ilustra o processo.

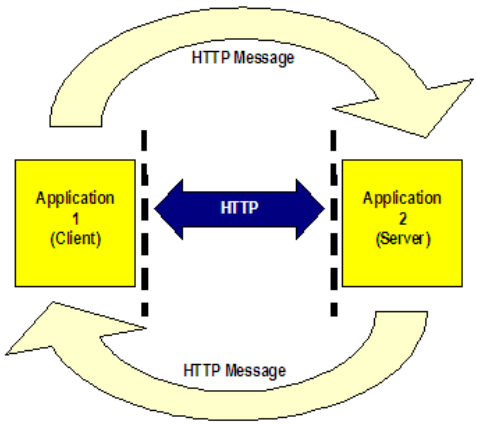

Figura 6. Troca de mensagens HTTP.

O *web service* irá ler o conteúdo que está dentro do envelope HTTP, processar de alguma forma, e retornar uma resposta ao cliente no mesmo padrão, colocando o resultado dentro de outro envelope HTTP, ou seja, no corpo da solicitação HTTP enviando uma resposta ao solicitante. Aqui, pode-se pensar: "Isso se parece com uma solicitação *Resquest/Response* padrão". Na verdade, é uma solicitação

Request/Response padrão, pois o REST se baseia exclusivamente em recursos HTTP, e o funcionamento padrão do HTTP, desde o início de sua utilização na Web é esse.

Listagem 2. Exemplo de envelope HTTP de solicitação.

```
GET http://localhost:8080/ContasAPagarWSRest/resources/pagaments/2/
Protocol HTTP/1.1
Host localhost:8080
User-Agent Mozilla/4.0
Accept      application/xml
Accept-language pt-br
Accept-encoding gzip, deflate
Connection Keep-Alive
```

Listagem 3. Exemplo de envelope HTTP de resposta.

```
<?xml version="1.0" encoding="UTF-8"?> version="1.0" encoding="UTF-8"
standalone="yes"
<pagament
uri="http://localhost:8080/ContasAPagarWSRest/resources/pagaments/2/">
<cedente>SABESP</cedente>
<descricao>CONTA DE AGUA</descricao>
<dpagto>
<dvenc>10/11/2008</dvenc>
<ndoc>2</ndoc>
<valor>34.5</valor>
</pagament>
```

Mas, o que muda na troca de mensagens SOAP? Para o entendimento, deve-se observar a Figura 7.

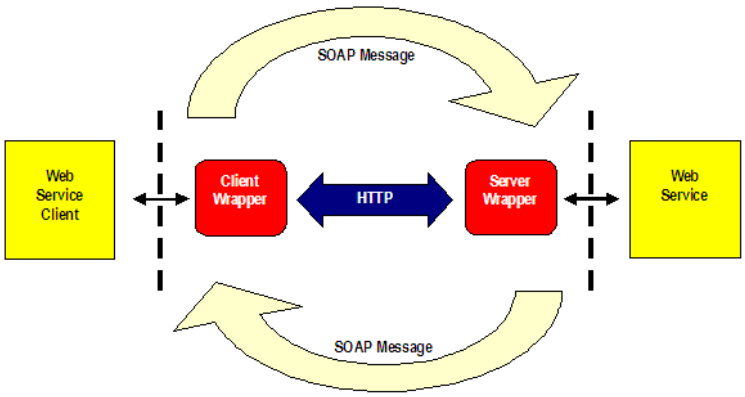

Figura 7. Troca de mensagens SOAP.

No caso SOAP, as informações que serão enviadas ao *web service* devem ser colocadas dentro de um envelope no padrão SOAP, portanto no formato XML. Posteriormente, esse envelope SOAP será colocado dentro de um envelope HTTP, e

somente então a mensagem será enviada. Esse “envelopamento” no padrão SOAP é feito pelo “*wrapper*”, no caso o que faz parte da aplicação cliente. Quando a mensagem chega ao *web service* também há um *wrapper* para “desenvelopar” as informações, que serão processadas pelo *web service* e “envelopadas” novamente no padrão SOAP, e depois em HTTP, para retornarem ao solicitante do serviço. Esse último, quando recebe a resposta do *web service*, executa o mesmo processo descrito anteriormente, para obter as informações contidas no envelope SOAP e processá-las em seguida.

Listagem 4. Exemplo de envelope SOAP de solicitação.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Header/>
<S:Body>
<ns2:listarConta xmlns:ns2="http://webservices.soap.ws.policamp/">
<nDoc>2</nDoc>
</ns2:excluirConta>
</S:Body>
</S:Envelope>
```

Listagem 5. Exemplo de envelope SOAP de resposta.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
<S:Body>
<ns2:listarContaResponse
xmlns:ns2="http://webservices.soap.ws.policamp/">
<return>
<cedente>SABESP</cedente>
<descricao>CONTA DE AGUA</descricao>
<dpagto/>
<dvenc>10/11/2008</dvenc>
<ndoc>2</ndoc>
<valor>34.5</valor>
</return>
</ns2:listarContasResponse>
</S:Body>
</S:Envelope>
```

É fácil perceber com essa explicação que a troca de mensagens no padrão SOAP é relativamente mais “pesada” que no REST. Esse *parsing* XML, imposto pelo padrão SOAP, causa um grande consumo de recursos computacionais.

Apesar da longa explicação anterior para expor o peso do XML nos *web services* SOAP, e como os *web services* REST são leves, fazendo essa comparação, não se pode esquecer questões como segurança, transações e até mesmo o envio de

mensagens de forma confiável, que são áreas em que as especificações e implementações SOAP estão muito mais avançadas em relação ao HTTP nativo. Tem-se, portanto, esses pontos que pesam a favor do SOAP e contra o REST, pois, em qualquer sistema que deva considerar em primeiro lugar a segurança das informações transportadas, certamente o pesado *parsing* XML do não será um problema.

ATIVIDADES E RECURSOS

Além das diferenças técnicas já apresentadas entre os *web services* SOAP e REST, a partir de agora será abordada a forma como se deve olhar para um determinado domínio de problema, de acordo com o tipo de *web services* que se deseja implementar.

Se a intenção é implementar um sistema dentro dos padrões SOAP, então estuda-se o domínio do problema tentando identificar “atividades”, ou seja, “o quê” ou “quais atividades” devem ser disponibilizadas pelo sistema. Por isso, os *web services* SOAP são denominados como “Orientados a Atividades”. Em termos práticos, após a implementação, teremos o cliente de *web services* invocando as “atividades” que na verdade serão métodos de um determinado *web service*. A Figura 8 ilustra o processo.

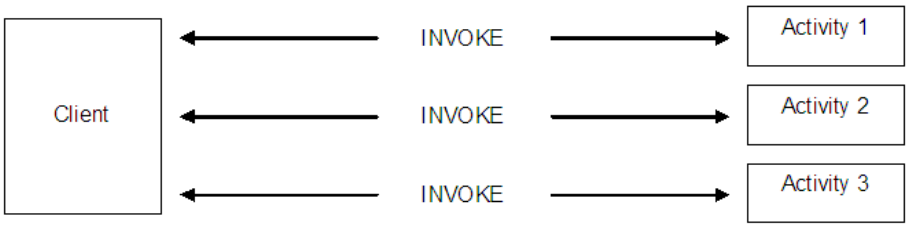

Figura 8. SOAP: web services orientados à atividades.

Por outro lado, querendo implementar o sistema seguindo os padrões REST, o domínio do problema deve ser estudado de forma que sejam identificados “recursos”. Dessa forma, a pergunta que se deve tentar responder é: “Quais recursos compõem esse sistema?”. É por esse motivo que os *web services* REST são conhecidos também como “Orientados à Recursos” (IBM, 2009). Após a implementação, o que se tem é o cliente de *web services* utilizando o “recurso” através da chamada aos métodos do HTTP, como GET, PUT, DELETE e POST, conforme mostrado na Figura 9.

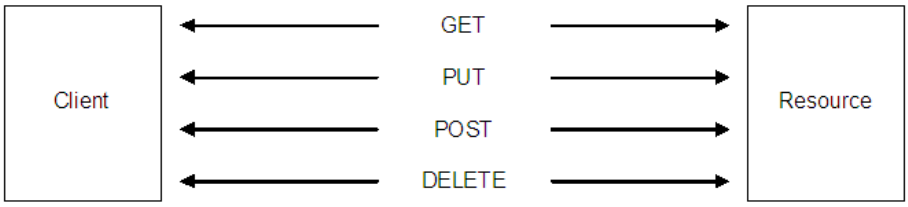

Figura 9. REST: *web services* orientados à recursos.

Para tornar essa explicação mais palpável, será apresentado um domínio de problema e, partir dele, a identificação das “atividades” para um sistema SOAP, e dos “recursos” para um sistema REST, apresentando em seguida uma alternativa de implementação para cada um dos padrões.

DEFININDO WEB SERVICE

Parte-se de uma proposta simples e pequena, para que seja didática. Dessa forma, o leitor poderá compreender exatamente o que foi e como foi produzido, podendo a partir daí, criar propostas de implementação mais elaboradas.

O sistema consiste em um controle de contas a pagar, e deve permitir ao usuário listar todas as contas lançadas, inserir novas contas, excluir contas e alterar os dados de uma conta. As informações das contas que devem ser consideradas pelo sistema são: Número do Documento, Descrição, Cedente, Data de Vencimento, Valor, Data de Pagamento e Valor Pago. Com base nessas informações, foi criado o seguinte diagrama use-case, Figura 10, para auxiliar na definição dos *web services*.

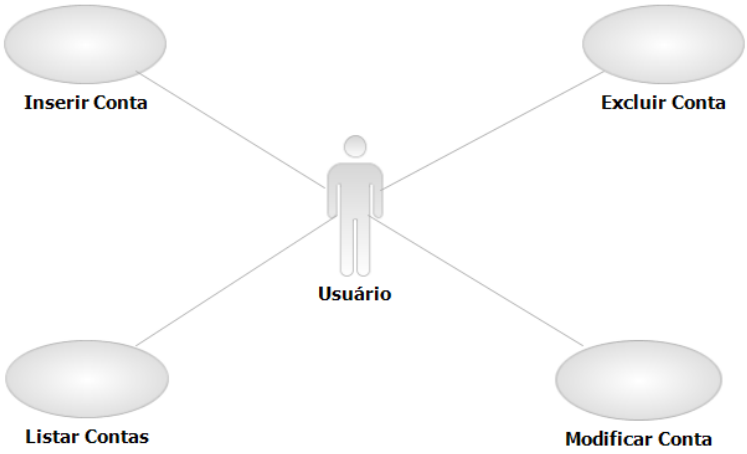

Figura 10. Diagrama Use-Case do sistema de conta a pagar.

Do ponto de vista SOAP, é necessário identificar quais são as atividades que deverão ser disponiblizadas pelo sistema. Basicamente, cada um dos *use-cases* correspondendo a uma atividade. As atividades definidas são:

- Inserir Conta
- Listar Contas
- Modificar Conta
- Excluir Conta

Como o sistema será implementado em Java (Sun, 2009), há que se criar uma classe, que corresponderá ao *web service*. Essa classe será composta por métodos, também conhecidos como "*web method*", os quais representarão as quatro atividades identificadas.

Listagem 6. Exemplo *web service* SOAP implementado em Java.

```
@WebService()
@Stateless()
public class ContasAPagarWS {
    @EJB
    private PagamentFacadeRemote pagamentFacadeBean;

    @WebMethod(operationName = "inserirConta")
    public void inserirConta(
                @WebParam(name = "descricao")String descricao,
                @WebParam(name = "cedente")String cedente,
                @WebParam(name = "dVenc")String dVenc,
                @WebParam(name = "valor")double valor){
       pagamentFacadeBean.adicionar(descricao,cedente,dVenc,valor);
```

```
    }
}
```

Quando se fala em REST, tem-se que identificar quais recursos compõem o sistema e, a partir dos recursos definir quais métodos HTTP (GET, PUT, POST e DELETE) agirão sobre cada um. Nesse caso, tem-se dois recursos, “Conta”, que representa uma única conta cadastrada, e “Contas”, que corresponde à todas as contas cadastradas. Segue a Tabela 5, na qual há o mapeamento dos recursos, seus métodos, status de resposta e suas representações:

Tabela 5. Mapeamento de recursos, métodos, representações e códigos de retorno do HTTP.

Recurso	Método	Representação	Status
Conta	GET	Pagament	200
			404
	PUT	Pagament	200
			404
	POST	Pagament	201
	DELETE		200
			404
Contas	GET	Pagaments	200
			404

Cada recurso definido na tabela será implementado por uma classe Java, com os seus respectivos métodos.

Listagem 7. Exemplo *web service* REST implementado em JAVA

```
public class PagamentResource {
    private Integer id;
    private UriInfo context;

    public PagamentResource(Integer id, UriInfo context) {
        this.id = id;
        this.context = context;
    }

    @GET
    @ProduceMime({"application/xml", "application/json", "text/xml",
                  "text/plain, "text/html"})
    public PagamentConverter get() {
```

```
            try {
                return new PagamentConverter(getEntity(),
                                context.getAbsolutePath());
            } finally {
                PersistenceService.getInstance().close();
            }
        }
    }
```

É óbvio que outras classes terão de ser criadas para a composição dos sistemas como um todo, tanto para o caso SOAP quanto para o REST.

IMPLEMENTANDO WEB SERVICES ATRAVÉS DO NETBEANS 6.1

Para a implementação dos *web services* SOAP e REST propostos foram utilizados o *Netbeans* 6.1, o *Glassfish* V2 e o Java DB.

No caso do *web service* SOAP, foi criado um projeto do tipo “EJB Module”, selecionando o “*Glassfish V2*” como servidor. A criação do *web service* foi realizada clicando-se com o botão direito do mouse sobre o projeto, selecionando a opção “*New > web service* ”. Os *web methods* podem ser adicionados ao *web service* através de uma interface gráfica ou diretamente ao código da classe. A Figura 11 ilustra a interface gráfica para criação.

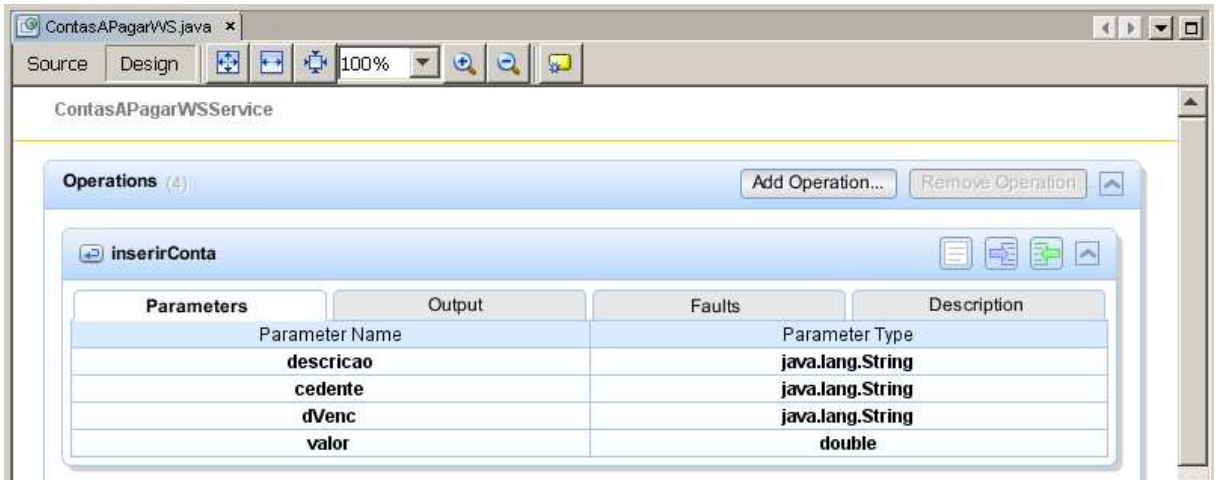

Figura 11. Interface gráfica para a criação de *web services.*

O cliente do *web service* SOAP foi implementado através de um projeto do tipo “*Web Application*” e o “Glassfish V2” como servidor. Clicando com o botão direito do mouse sobre o projeto e selecionando a opção “*New > Web Service Client*”, basta

informar qual projeto contém o *web service* que se deseja consumir, e todas as classes (*stubs*) necessárias para o acesso a esse *web service* serão criadas, com base no WSDL do serviço informado (Figura 12).

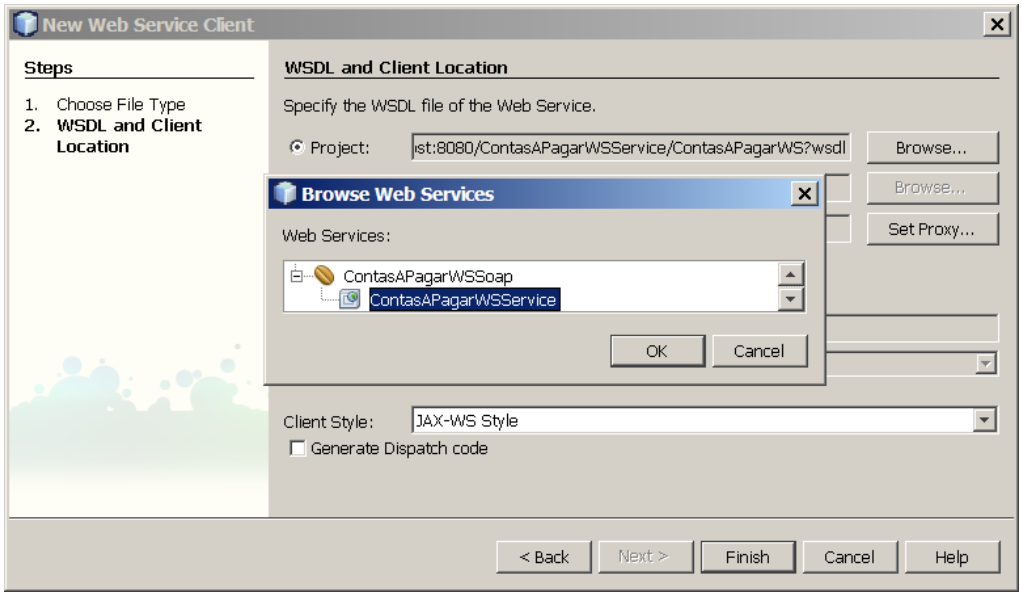

Figura 12. Criando um SOAP *web service client*.

O *web service* REST foi criado a partir de um projeto "*Web Application*", tendo o "*Glassfish* V2" como servidor. Em seguida, foi criada uma classe de entidade baseada numa tabela denominada "*Pagament*", exibida mais adiante. Clicando-se com o botão direito do mouse sobre o projeto e selecionando a opção "*New > Entity Classes from Database*". A geração do *web service* foi realizada clicando-se com o botão direito do mouse sobre o projeto, e escolhendo a opção "*New > RESTful Web Services from Entity Classes*", conforme Figura 13.

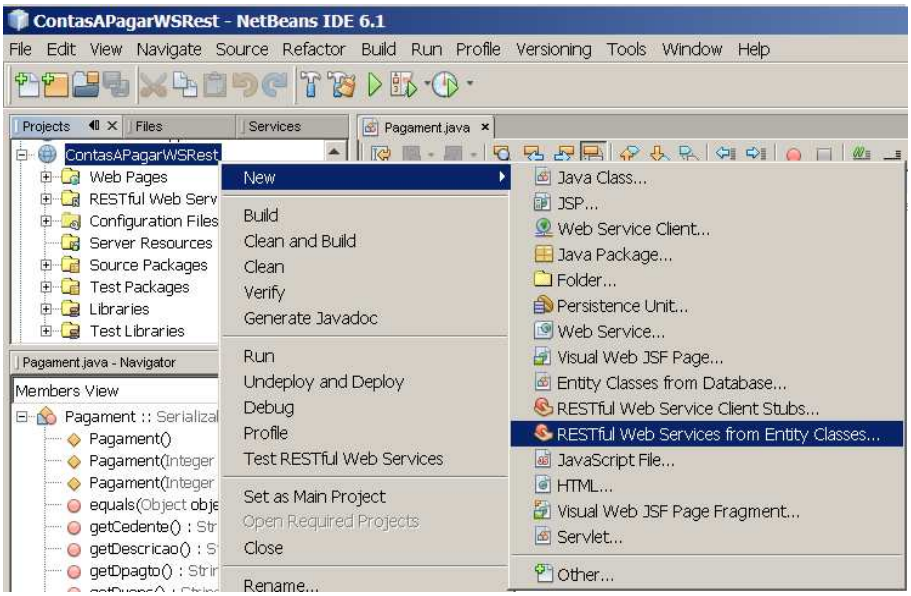

Figura 13. Criando um *web service* REST baseado em *Entity Classes.*

O cliente para o *web service* REST foi desenvolvido através de um projeto do tipo "*Web Application*", tendo o "*Glassfish V2*" como servidor. Clicando-se com o botão direito do mouse sobre o projeto e selecionando a opção "*New > RESTful Web Service Client Stubs*", basta informar o nome do projeto que contém o *web service* a ser consumido pelo cliente. Nesse caso, serão criados arquivos "*Java Script*" que permitirão o acesso e o consumo do *web service* em questão, conforme ilustrado na Figura 14.

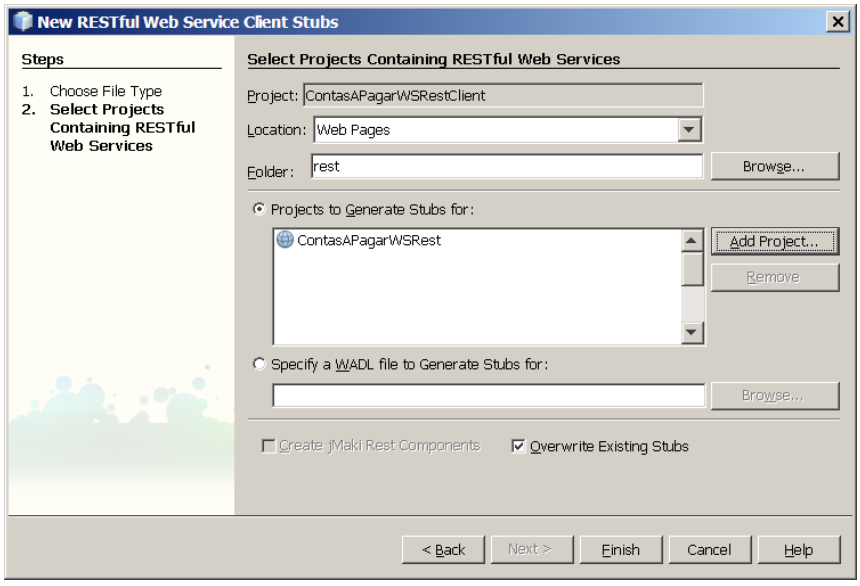

Figura 14. Criando um *web service* cliente para REST.

É importante ressaltar que ambos os projetos foram criados com a utilização de um banco de dados Java DB, chamado “FINANCE”, contendo uma única tabela denominada “PAGAMENT”, cuja estrutura é mostrada na Figura 15.

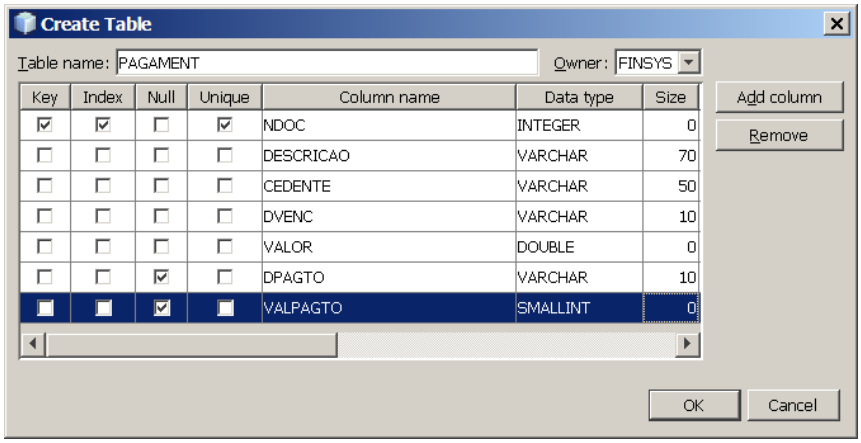

Figura 15. Estrutura da tabela “PAGAMENT”.

CONSIDERAÇÕES FINAIS

Como antecipado, a intenção deste artigo era expor ao máximo as características de cada um dos padrões e não confrontá-los. De acordo com o que foi apresentado, pode-se observar o que realmente acontece quando *web services* SOAP e REST são executados. Também pode-se apontar quais suas melhores e piores características, em vários aspectos. Dessa forma, o leitor, com base nesse texto, tem totais condições de definir qual o melhor padrão a ser utilizado diante de um determinado domínio de problema. Por exemplo, optando pelo REST, quando se deseja alcançar um melhor desempenho, ou, colocando a segurança das informações antes de qualquer outra coisa, e definindo o SOAP como opção de implementação. Essa maneira de conduzir o desenvolvimento de software é extremamente coerente, deixando de lado as “bandeiras” e procurando sempre definir a melhor solução para o problema em questão.

REFERÊNCIAS BIBLIOGRÁFICAS

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Tese de Doutorado, Univ. da Califórnia, 2000. Disponível em: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, recuperado em 10/06/2009.

GREGORIO, Joe. **BITWORKING How to create a REST protocol**. Disponível em: http://www.bitworking.org/news/How_to_create_a_REST_Protocol, recuperado em 10/06/2009A.

GREGORIO, Joe. **BITWORKING REST and WS**. Disponível em: http://bitworking.org/news/125/REST-and-WS, recuperado em 10/06/2009B.

IBM**. Resource-oriented vs. activity-oriented web services.** Disponível em: http://www.ibm.com/developerworks/xml/library/ws-restvsoap/, recuperado em 10/06/2009.

JCP. JSR 311: The Java API for RESTFul Web Services. Disponível em: https://jsr311.dev.java.net/, recuperado em 10/0-6/2009.

SUN MICROSYSTEMS. **Overview: The JavaEE 5 Tutorial**. Disponível em: http://java.sun.com/javaee/5/docs/tutorial/doc/bnaaw.html, recuperado em 10/06/2009.