CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

RESTMB: API RESTful para Android

JONATHAN CAMPOS

Marília, 2013

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
FUNDAÇÃO DE ENSINO "EURÍPIDES SOARES DA ROCHA"
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

RESTMB: API RESTful para Android

Monografia apresentada ao Centro Universitário Eurípides de Marília como parte dos requisitos necessários para a obtenção do grau de Bacharel em Sistemas de Informação
Orientador: Prof. Ricardo José Sabatine

Marília, 2013

CAMPOS, Jonathan
RESTMB: API RESTful para Android.
Jonathan Campos; orientador: Prof. MSc. Ricardo José Sabatine. Marília, SP: [s.n.], 2013.
68 folhas

Monografia (Bacharelado em Sistemas de Informação) - Centro Universitário Eurípides de Marília.

1. Rest 2. Web Service 3. Recurso

CDD: 005.2

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM SISTEMAS DE INFORMAÇÂO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Jonathan Campos

RESTMB: API RESTful para Android

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Sistemas de Informação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Sistemas de Informação.

Nota: 9,5 (nove e meio)

Orientador: Ricardo José Sabatine

1°. Examinador: Emerson Alberto Marconato

2°. Examinador: Giulianna Marega Marques

Ricardo Sabatine

Marília, 04 de dezembro de 2013.

Aos meus pais pelo apoio e dedicação
na minha formação.

AGRADECIMENTOS

Agradeço primeiramente, a DEUS, por me dar força para seguir sempre em frente e concluir meus objetivos.
Aos meus pais pelo apoio, carinho e incentivo de sempre.
Á Berenice Romeiro de Campos, um anjo que passou por minha vida e, sempre me incentivou a estudar e concluir meus sonhos.
Á Sonia A. Romeiro pelo carinho e amor dedicado ao longo desses anos.
Á minha irmã pelo apoio e cumplicidade sempre.
Á Tatiane Lopes pelo carinho e paciência nos momentos mais difíceis.
Á Maria Fernanda Lopes Tripolone por me esperar terminar esse trabalho, e assim ter seu companheiro de jogos novamente.
Á Sueli Deolinda pelo carinho e por me aguentar falar de TCC ao longo desse ano.
Ao meu orientador por sempre me guiar (como um labrador guia um cego. Rs.) nas partes mais difíceis desse trabalho.
Aos colegas do bacharelado pela companhia nesses anos e pelos grandes debates e crescimento proporcionados.
Ao Renan C. Rinaldi pela amizade, apoio e conforto durante os quatro anos de faculdade.
Aos amigos e à família que de alguma forma compreenderam a minha ausência em muitos momentos.
À instituição analisada e aos participantes da pesquisa pela disponibilidade e atenção.

“As pessoas não sabem o que querem,
até mostrarmos a ela”. (Steve Jobs)

CAMPOS, Jonathan. **RESTMB: API RESTful para Android.2013**. 68 f. Trabalho de Curso de Bacharelado em Sistema de Informação – Centro Universitário Eurípides de Marília, Fundação de Ensino ”Eurípides Soares da Rocha”, Marília, 2013.

RESUMO

Devido ao grande poder de processamento de smartphones e tablets e a facilidade de se manter conectado à internet por wi-fi ou 3G, os dispositivos móveis se tornaram indispensáveis para diversas empresas. É possível utilizar os diversos recursos do dispositivo para enviar e receber dados a todo momento, empresas vêem realizando a integração de sistemas mobile aos seus *ERPs* para assim agilizar seus processos. Para realizar a integração de aplicações heterogêneas são utilizados *Web Services* que disponibilizam recursos para que outras aplicações possam ser facilmente integradas. Este trabalho apresenta o desenvolvimento de uma *API* JAVA com classes e interfaces que mapeiam os conceitos básicos de serviços web baseados em *Rest*. A estrutura da *API* desenvolvida facilita o trabalho com diferentes propostas de arquiteturas *Rest*, além disso a *API* visa facilitar o trabalho de desenvolvimento de uma aplicação Android que se comunique com um servidor *Rest*, não sendo necessário que o desenvolvedor preocupe-se com o desenvolvimento de classes de comunicação com o *Web Service* tampouco classes que realizem o *marshall* e *unmarshall* de dados.

Palavras-Chave: API, Rest, Web Services.

ABSTRACT

Due to the great processing power of smartphones and tablets, and the ease of staying connected to the internet by wi-fi or 3G , mobile devices have become indispensable for many companies . You can use the various features of the device to send and receive data at all times, company have been performing the integration of mobile systems to their ERPs to streamline their processes as well. To realize the integration of heterogeneous applications are used Web Services to provide resources so that other applications can be easily integrated. This paper presents the development of a JAVA API with classes and interfaces that map the basics of Rest-based web services. The structure of the developed API makes working with different proposed architectures Rest in addition the API is intended to facilitate the work of developing an Android application that communicates with a Rest server, not requiring the developer to worry with the development of classes to communicate with the Web Service classes that perform either marshall and unmarshall data.

Keywords: API, Rest, Web Services.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

3G	Terceira Geração de Tecnologia de Internet Móvel
4G	Quarta Geração de Tecnologia de Internet Móvel
API	Application Programming Interface
ERP	Enterprise Resource Planning
GPS	Global Positioning System
HATEOAS	Hypermidia As The Engine Of Application State
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	Javascript Object Notation
MIME	Multipurpose Internet Mail Extension
OAuth	Open Authorization
RAM	Random Access Memory
REST	Representational State Transfer
RestMB	Rest Mobile
RPC	Remote Procedure Call
SOAP	Simple Object Acess Protocol
URI	Uniform resource identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
XML	Extensible Markup Language

SUMÁRIO

INTRODUÇÃO

A computação móvel surgiu com o objetivo de permitir que seus usuários mantenham-se conectados a todo o momento, tal como faria em um *Desktop*, porém tendo a flexibilidade de transportar o dispositivo para qualquer lugar.

Essa tecnologia não é tão nova quanto se pensa e já existia em PDAs e *notebooks* mas com recursos limitados e conexão de dados lenta, a popularização só veio com a chegada da terceira geração de tecnologia móvel (3G) e a sua rápida expansão da área de cobertura, além disso, a mesma permite uma conexão de alta velocidade.

Atualmente os dispositivos como *smartphones* e *tablets* criados por grandes fabricantes de dispositivos móveis tais como: Samsung, LG, Motorola e Sony possuem uma gama de tecnologias como: 3G (Terceira Geração de Tecnologia de Internet Móvel), 4G (Quarta Geração de Tecnologia de Internet Móvel), GPS (Global Positioning System), Câmera, Acelerômetro entre outros.

Uma outra característica que também foi indispensável para o sucesso desses dispositivos foi o aumento da quantidade de memória RAM e as novas arquiteturas de processador que possibilitaram frequências de *clocks* mais altas aumentando assim o desempenho e a possibilidade de se construir aplicações mais sofisticadas. Com esse aumento significante, esses dispositivos puderam ser utilizados como uma ferramenta tanto de trabalho, estudo ou entretenimento.

O Brasil fechou setembro de 2013 com 268,27 milhões de linhas ativas na telefonia móvel e tele densidade de 135,28 acessos por 100 habitantes. A banda larga móvel totalizou

88,31 milhões de acessos, dos quais 552,63 mil são terminais 4G (ANATEL, 2013).

Na tabela 1, é possível analisar o crescimento no número de linhas ativas de aparelhos celulares.

Tabela 1: Número de linhas ativas na telefonia móvel (ANATEL, 2013)

	Acessos em Operação	Densidade (acessos por 100 habitantes)
Brasil	268.266.822	135,28
Distrito Federal	6.007.764	220,04
Goiás	9.177.899	147,12
Mato Grosso	4.501.341	139,54
Mato Grosso do Sul	3.760.416	150,7
Total da Região Centro-Oeste	23.447.420	159,62
Alagoas	3.948.145	118,94
Bahia	17.702.117	116,62
Ceará	10.665.129	119,71
Maranhão	6.478.590	96,51
Paraíba	4.780.966	121,43
Pernambuco	12.267.826	135,05
Piauí	3.838.224	116,44
Rio Grande do Norte	4.496.471	135,8
Sergipe	2.694.435	126,35
Total da Região Nordeste	66.871.903	119,66
Acre	917.352	122,95
Amapá	937.631	135,8
Amazonas	4.120.182	112,73
Pará	8.937.304	113,57
Rondônia	2.380.778	151,3
Roraima	504.739	108,99
Tocantins	1.879.161	137,94
Total da Região Norte	19.677.147	120,28
Espírito Santo	4.534.637	126,2
Minas Gerais	25.909.479	124,92
Rio de Janeiro	23.894.086	147,38
São Paulo	64.949.184	153,63
Total da Região Sudeste	119.287.386	144,03
Paraná	14.427.008	130,73
Rio Grande do Sul	15.942.127	143,4

Santa Catarina	8.613.831	134,58
Total da Região Sul	38.982.966	136,52

Fonte: ANATEL

Devido a popularização dos sistemas operacionais Android e IOS para dispositivos móveis e o crescente aumento de planos de telefonia acessíveis, empresas começaram a despertar interesses em explorar essas novas tecnologias para disponibilizar seus produtos e serviços. Utilizar esse tipo de tecnologia nos negócios possibilita para um empresário ter informação sempre à mão aonde quer que esteja para consultar quando quiser. Através de Web Services é possível criar serviços para que aplicações móveis possam se comunicar, consumir recursos e realizar a manutenção de informações em um ERP *(Enterprise Resource Planning*), por exemplo.

Objetivos

Esse trabalho proporcionou o desenvolvimento de uma API (*Application Programming Interface*) com o objetivo de facilitar o desenvolvimento de um aplicativo que necessite se comunicar com um serviço baseado em Rest.

Com a conclusão do trabalho a API disponibilizará métodos que encapsulam a complexidade de consultas a um serviço Rest bem como o *marshall* e *unmarshall* dos dados.

1. WEB SERVICE

Devido à necessidade das empresas em tornar seus processos automatizados para atender de maneira eficiente e eficaz a seus clientes é nítido o desenvolvimento de várias ferramentas para agilizar esses processos.

Com a demanda, o mercado passou a oferecer diversas ferramentas que prometem cumprir a solicitação de automatizar os processos e gerar informações de forma ágil, de maneira que ajude as empresas a terem um diferencial no mercado.

Porém, nem sempre um Software como, por exemplo, um ERP fornece todas as ferramentas necessárias para satisfazer a necessidade de todas as áreas da empresa, sendo assim, muitas vezes é necessário realizar a integração entre softwares distintos escritos em linguagens diferentes e rodando em plataformas diferentes que devem ser capazes de se comunicarem e trocarem informações para assim gerar toda a informação necessária para o negócio da empresa.

Web Service é uma aplicação distribuída cujos componentes podem ser implementados e executados em dispositivos distintos (KALIN, 2009). A essência de Web Services está em permitir que diferentes aplicativos em ambientes distintos possam interoperar independente da plataforma ou linguagem em que foram escritos. Um Web Service é uma parte da lógica de negócios que pode ser acessada através de protocolos padronizados como, por exemplo, o HTTP (CHAPPELL & JEWELL, 2002).

Um Web Service deve ter algumas características como:

Padronização para representações de dados

É necessária a utilização de uma padronização para a formatação dos dados para troca de informações entre cliente e servidor. Um exemplo disso é a utilização de XML ou JSON que se tornaram padrões em Web Services, facilitando assim a integração entre sistemas heterogêneos onde ambos conhecem a representação dos dados trocados entre si (CHAPPELL & JEWELL, 2002).

Baixo Acoplamento

Deve-se adotar uma arquitetura de baixo acoplamento para tornar a integração de

sistemas mais simples e robusta assim caso alguma mudança ocorra no servidor, o cliente não precisará ser atualizado (CHAPPELL & JEWELL, 2002).

Capacidade de ser síncrona ou assíncrona

As operações assíncronas permitem que os clientes invoquem um serviço e em seguida execute outras funções, já operações síncronas permitem que os clientes invoquem um serviço e fique aguardando uma resposta para então dar continuidade a algum processamento (CHAPPELL & JEWELL, 2002).

Suportar chamadas de procedimento remoto (RPCs)

Web Services devem permitir a chamada de procedimentos, funções e métodos remotos (CHAPPELL & JEWELL, 2002).

Suportar troca de documentos

Deve permitir a troca de dados sejam eles pequenos ou grandes, simples ou complexos para facilitar a integração entre sistemas distintos (CHAPPELL & JEWELL, 2002).

1.1 A arquitetura WS-*

A arquitetura WS-* é composta por mais de 20 especificações, sendo especificações base deste conjunto a SOAP (Simple Object Acess Protocol), WSDL (Web Services Description Language) e UDDI (Universal Description, Discovery and Integration).

Além dos citados, existem também os padrões WS-Notification, WS-Addressing, WS-Transfer, WS-Police, WSSecurity, WSTransaction entre outros (W3C).

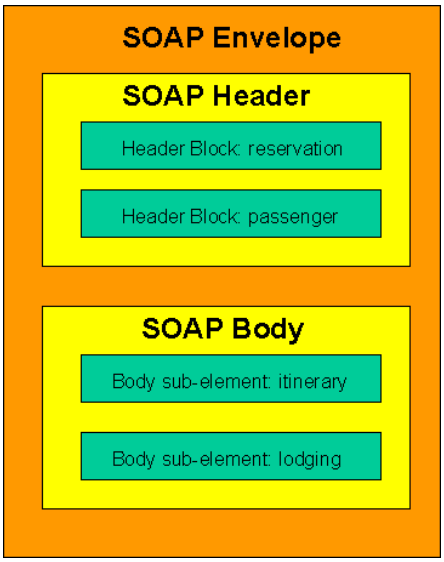
1.1.1 SOAP

Uma forma de se construir um *web service* é utilizando o protocolo SOAP *(Simple Object Acess Protocol)*, através da infraestrutura criada para o HTTP para atravessar “firewalls” e roteadores o SOAP é beneficiado facilitando assim a troca de informações entre os serviços.

Além de leve, SOAP é baseado em XML e foi projetado para a troca de informações em um ambiente de computação distribuída.

A Figura 1 ilustra os elementos do SOAP.

Figura 1 - Estrutura SOAP



Fonte: W3C

- Envelope: Elemento raiz do SOAP e define que essa é uma mensagem SOAP, é utilizado para transportar a mensagem.

- Cabeçalho (Header): Contém bloco de informações sobre como a mensagem será processada, isto inclui configurações de encaminhamento e entrega, autenticação ou declarações de autorização e contextos de transação.

- Corpo (Body): Contém a mensagem que será entregue e processada.

A Figura 2 ilustra o conteúdo de uma requisição SOAP, é possível observar que o método de requisição HTTP é um POST e o conteúdo é estruturado utilizando representação de dados XML.

Figura 2 - Exemplo de Envelope SOAP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Fonte: W3C

1.1.2 WSDL

WSDL *(Web Services Description Language)* é uma especificação que permite descrever Web Services utilizando XML.(Cerami, 2002).

WSDL descreve 4 pontos críticos de dados:

- Informações que descrevem todas as funções disponíveis publicamente.
- Descreve os tipos de dados para as solicitações e respostas de mensagens.
- Informações sobre o protocolo de transporte utilizado.
- Endereço para localizar o serviço.

O WSDL representa um contrato entre o cliente que solicita o serviço e o servidor que dispõe o serviço para ser acessado, facilitando assim a integração entre eles, pois no contrato estarão todos os métodos disponíveis no Web Service e os tipos de dados de entrada e saída de cada método.

A especificação WSDL está divididas em 6 elementos:

- **Definitions**

É o elemento raiz de todo documento WSDL, contém o nome do Web Service e declara várias namespaces que são utilizadas ao longo do documento.

- **Types**

É o elemento que descreve todos os tipos de dados aceitos para troca entre o Web Service e a aplicação cliente.

- **Messages**

É o elemento que descreve os dados que serão trocados entre o Web Service e o Cliente, o Web Service tem duas mensagens, entrada (input) e saída (output).

Figura 3 - Exemplo de uma mensagem SOAP

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>
```

Fonte: W3C

- **portType**

Combina vários elementos da mensagem para formar um pedido e uma mensagem de resposta para uma única operação.

- **Binding**

É o elemento que descreve em detalhes como a operação portType será transmitida pela rede.

- **Service**

  É o elemento que define o endereço para invocar um determinado serviço.

1.1.3 UDDI

Universal Description, Discovery and Integration UDDI é um protocolo aprovado como padrão pelo OASIS e especifica um método para publicar e descobrir diretórios de serviços em uma arquitetura orientada a serviços.

1.2 REST

A *World Wide Web* faz parte do cotidiano de muitas pessoas, ou seja, no trabalho, na escola ou no momento de lazer. A *web* é utilizada para diversos fins como: ler uma notícia, acessar as redes sociais, fazer um trabalho escolar, aprender um novo idioma, enfim tudo isso é proporcionado devido a uma arquitetura robusta, eficiente e tolerante a falhas.

Fielding em sua tese de doutorado apresentou princípios da arquitetura *Web*, descrevendo um conjunto de restrições, ou um estilo de arquitetura, algo como um conjunto limitado de operações, a tal estilo de arquitetura foi dado o nome *Representational State Transfer* (REST).

REST é um estilo de arquitetura para sistemas de hipermídia distribuídos Fielding (2000).

Seguindo os princípios da arquitetura REST é possível construir sistemas mais flexíveis, eficientes, com baixo acoplamento, leves e com isso ter um sistema que poderá ser facilmente reutilizado.

1.2.1 REST e o HTTP

O REST não é um protocolo de comunicação como SOAP, o REST trabalha em conjunto com o protocolo HTTP aproveitando os recursos do mesmo, diferentemente do SOAP que utiliza o protocolo apenas como um meio de transporte para trafegar seus envelopes através de requisições feitas por aplicações cliente e respostas do servidor.

O HTTP é o principal protocolo utilizado na Web por ser robusto e simples, seus principais métodos são GET, PUT, POST e DELETE.

Segundo (Burke, 2010) o Funcionamento do HTTP é muito simples, o cliente envia uma requisição ao servidor informando o método a ser utilizado, a localização do recurso a ser invocado, um conjunto de variáveis no cabeçalho e um corpo de mensagem opcional que pode ser qualquer formato tais como: HTML, XML ou JSON entre outros. Abaixo exemplo de uma requisição a página da coca-cola no facebook utilizando o método GET:

Tabela 2 - Exemplo de Requisição ao Facebook Graph API

Request URL:
`https://graph.facebook.com/coca-cola`
Request Method: `GET`
Status Code: `200 OK`
host: `graph.facebook.com`
method: `GET`
path: `/coca-cola`
scheme:`https`
version: `HTTP/1.1`
accept: `json`
accept-encoding: `gzip,deflate,sdch`
accept-language: `pt-BR,pt;q=0.8,en-US;q=0.6,en;q=0.4`
authorization:`CAACEdEose0cBANDkUZA6kJ2ZBxz09OuEZBUNxsP90u3RDFNMIUZBYMLT1OnwInZAXJuKXa1iSyBIUH VoZCXsjnZAjXDCZA3dhHg1UFcmzccc6YcOfnksOOmWYM2Tb9gSkx5D7VxNmid21j2JTHDU8MJ6rZCZBND8rEDlkZD`

Fonte: Jonathan Campos

O REST é uma junção de vários estilos de arquitetura de rede, ou seja, é um estilo hibrido tais como:

· Cliente/Servidor
· Stateless

· Interface Uniforme
· Cache
· Code on Demand.

1. **Cliente/Servidor**

O estilo Cliente/Servidor visa separar as responsabilidades e é muito utilizado em aplicações baseadas em rede, visa principalmente separar a interface do usuário do armazenamento de dados, facilitando assim a responsabilidade do servidor. No estilo cliente/servidor têm-se um Servidor responsável por oferecer um conjunto de serviços que serão invocados por aplicações clientes, o servidor disponibiliza seus recursos, para ser consumido por algum cliente independente da plataforma ou da linguagem escrita (Fielding, 2010).

2. **Stateless**

O estilo Stateless implica em não manter o estado entre a aplicação cliente e a aplicação servidora gerando assim um desacoplamento entre as aplicações, a cada nova requisição o cliente deve enviar todas as informações necessárias junto à requisição para que a mesma possa ser processada pelo servidor, isso diminui a sobrecarga dos servidores já que os mesmos não precisam armazenar o estado de cada cliente.

O Estilo Stateless cita três propriedades importantes, são elas:

- **Visibilidade:** Cada pedido (requisição) contém todas as informações necessárias para que o servidor possa processa-lo.
- **Confiabilidade:** Permite a recuperação de falhas parciais.
- **Escalabilidade:** Por o servidor não precisar armazenar o estado de cada requisição o mesmo pode servir a mais pedidos em um tempo menor

(Fielding, 2010).

3. **Interface Uniforme**

Estilo que define alguns princípios fundamentais como a identificação de recursos, diferentes representações, mensagens auto descritivas e recursos hipermídia (links) (Fielding, 2010).

4. **Cache**

Estilo que permite fazer cache das informações para que não seja preciso o acesso ao banco de dados a todo o momento, é utilizado para otimização de desempenho do sistema (Fielding, 2010).

5. **Code-on-Demand**

Código sob demanda é um paradigma de sistemas distribuídos que define a possibilidade e as técnicas de mover um código existente no servidor para a execução no cliente, um exemplo disso são os códigos JavaScript. Porém, o mesmo quebra alguns princípios de interoperabilidade, com isso, seu uso é opcional (Fielding, 2010).

2. Restful

De acordo com Fielding (2000) *REST* descreve os princípios que faz da Web uma imensa aplicação distribuída, oferecendo diversos serviços e aplicativos através de um conjunto de arquiteturas bem definidas. O termo *Restful* é utilizado para denominar aplicações que seguem os princípios REST.

Abaixo serão descritos os princípios que fazem do *REST* um estilo de sucesso adotado por diversas empresas entre elas Yahoo!, Facebook, Twitter, Amazon entre outros.

2.1 Endereçamento (Addressability)

O princípio de endereçamento indica que todo recurso em um sistema deve poder ser acessado por meio de um identificador exclusivo (Burke, 2010).

Para utilizar um recurso é preciso ter meios para identificá-lo na rede e poder manipulá-lo, a identificação se dá pelo uso de URIs que tornam os recursos endereçáveis e capazes de serem manipulados através da utilização de um protocolo, como por exemplo o Http.

Em Rest todo endereçamento é gerenciado pelo uso de URIs, toda requisição deve conter a URI do objeto que se está solicitando a informação. O formato de uma URI é formado através do seguinte padrão:

Figura 4 – Formato de URI

scheme://host:port/path?queryString#fragment

Fonte: Jonathan Campos

O primeiro elemento é o “scheme” que indica o protocolo que está sendo utilizado para realizar a comunicação, para aplicações Restful o protocolo utilizado é HTTP ou HTTPS. O elemento “host” indica um DNS ou IP do servidor que se quer acessar seguido pela porta do mesmo. O “path” expressa um conjunto de seguimento e é delimitado por uma “/”, uma analogia seria pensar em uma lista diretórios em um computador. O elemento “?” é utilizado para separar o elemento “path” da “queryString” que nada mais é do que uma String de consulta, onde se tem

uma lista de parâmetros separados por pares de chave e valor, cada par é delimitado utilizando-se o caractere &.

Abaixo exemplo de uma URI de requisição:

Figura 5 – Exemplo de URI de Requisição

http://univem.edu.br/alunos?nome=Jonathan&ra=449369

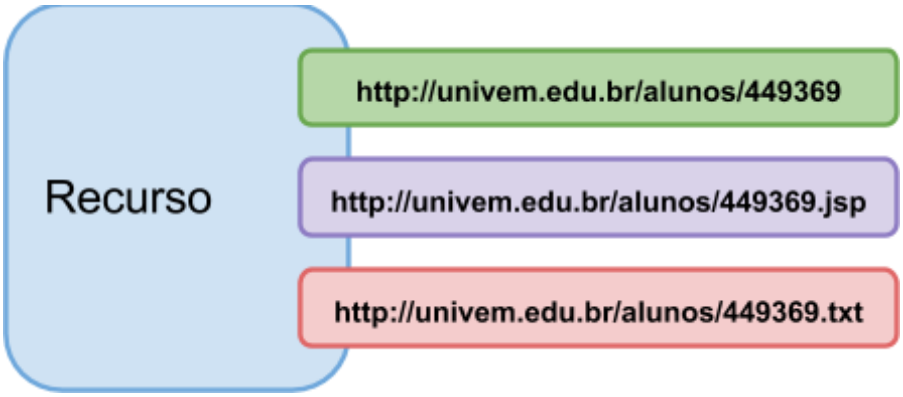
Fonte: Jonathan Campos

Nota-se no exemplo acima que conforme explicado anteriormente tem-se:

1. “http” como protocolo de comunicação.
2. “univem.edu.br” como o DNS do servidor Restful.
3. “/alunos” como o path a ser acessado no servidor.
4. O Caractere “?” utilizado para separar o path da lista de parâmetros de consulta.
5. “nome=Jonathan&ra=449369” a lista de parâmetros chave valor utilizados para consulta separados pelo caractere “&”

O relacionamento entre recursos e *URI* é de muitos-para-um, ou seja, uma URI pode identificar apenas um recurso, mas um recurso pode ter mais de uma URI associada a ele, conforme imagem abaixo:

Figura 6 - Recurso x URI



Fonte: Jonathan Campos

2.2 Interface Uniforme

O REST prega restrições quanto ao uso de operações, isso indica que não há um parâmetro de ação na URI como por exemplo “buscarAluno”, com isso o REST utiliza um conjunto limitado de métodos do protocolo HTTP.

Antes de listar os métodos utilizados é preciso entender dois conceitos importantes utilizados em cada método, tais como: métodos idempotente e seguro.

2.2.1 Método Idempotente

Um método é considerado idempotente quando não importa o número de vezes que o mesmo é chamado ele trará sempre o mesmo resultado, ou seja, se o método for chamado uma vez e trouxer o resultado “105”, caso seja chamado mais dez vezes deverá sempre trazer o mesmo resultado “105” (Webber ; Parastatidis ; Robinson, 2010).

2.2.2 Método Seguro

Um método é considerado seguro quando o mesmo não altera um recurso, como por exemplo, os métodos GET e HEAD nunca devem ser utilizados para se alterar um recurso (Webber ; Parastatidis ; Robinson, 2010).

Tabela 3 – Métodos HTTP

Método HTTP	Idempotente	Seguro
GET	SIM	SIM
PUT	SIM	NAO
POST	NÃO	NAO
DELETE	SIM	NAO
HEAD	SIM	SIM
OPTIONS	SIM	SIM

Fonte: Jonathan Campos

Abaixo os métodos HTTP utilizados em sistemas Restful:

- **GET**

O método é uma operação apenas de leitura e é usado para buscar informações no servidor, é um método idempotente e seguro.

- **PUT**

O método solicita que o servidor armazene os dados enviados através dele, pode ser utilizado para criar ou atualizar dados, é também idempotente, o método conhece a identidade do recurso que será criado ou atualizado.

- **DELETE**

Método utilizado para remover um recurso, é também um método idempotente.

- **POST**

O único método do HTTP que não é idempotente e nem seguro, é utilizado para criar um recurso.

- **HEAD**

Igual ao método GET, porém ao invés de retornar o response body retorna apenas o response code e os cabeçalhos associados à requisição.

- **OPTIONS**

O método é utilizado para saber quais métodos podem ser utilizados em determinado recurso, conforme exemplo:

Uma chamada a URI “https://graph.facebook.com/coca-cola” utilizando o método OPTIONS retornaria:

Figura 7 – Exemplo de resposta utilizando o http Options

```
Allow: GET, HEAD
```

Fonte: Jonathan Campos

Isso indica que para o recurso coca-cola é permitido utilizar os dois métodos citados acima.

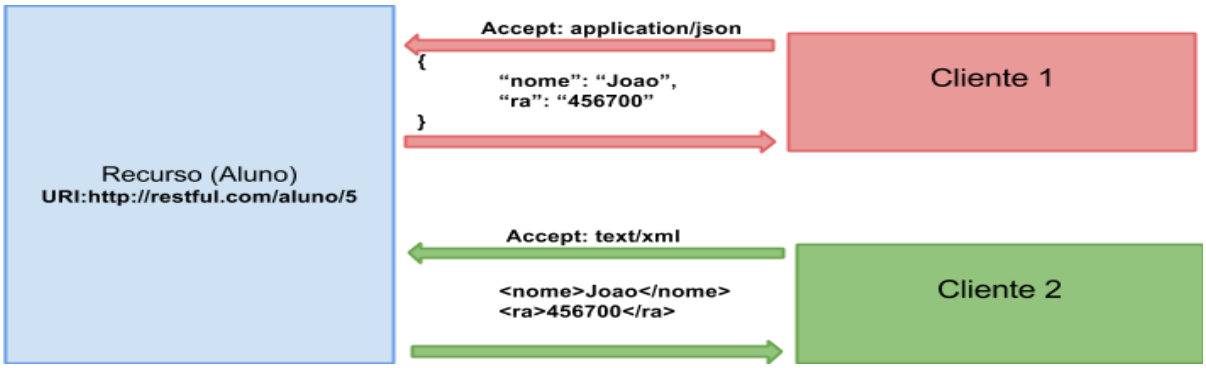
2.2.3 Orientado à Representações

Todo serviço se torna endereçável através de uma URI específica e representações são utilizadas para troca de informações entre cliente e servidor. As representações mais conhecidas são: json, xml, html, text, atom, mas nada impede de se utilizar uma representação diferente desde que, tanto o cliente como o servidor saibam como interpretá-la.

Utilizando-se um método GET, por exemplo, a aplicação cliente irá receber uma representação do recurso. O cabeçalho Content-Type utiliza o formato MIME (Multipurpose Internet Mail Extension) e é utilizado para dizer qual formato de dado o cliente ou o servidor está recebendo.

O método HTTP provê uma característica muito interessante para o tipo MIME que é a possibilidade do cliente e servidor negociarem o formato dos dados que os mesmos irão trocar. Através do cabeçalho “Accept”, um cliente pode dizer ao servidor qual o formato preferencial.

Figura 8 - Exemplo de Requisição



Fonte: Jonathan Campos

2.2.4 Comunicação sem Estado

“Rest stateless (sem estado) significa que não há dados de sessão de cliente armazenados no servidor. O servidor só registra e gerencia o estado dos recursos que ele expõe” (Burke, 2010).

O princípio Stateless diz que o servidor não deve armazenar nenhuma informação do cliente, isso facilita a escalabilidade e também o desempenho do serviço pois a única tarefa do servidor é responder às solicitações feitas pelos clientes, a comunicação sem estado também diz que sempre que uma nova requisição HTTP for feita, a mesma deve conter todas as informações necessárias para que o servidor consiga processá-la, sendo assim, o servidor não deve depender de informações realizadas por requisições anteriores.

2.2.5 HATEOAS

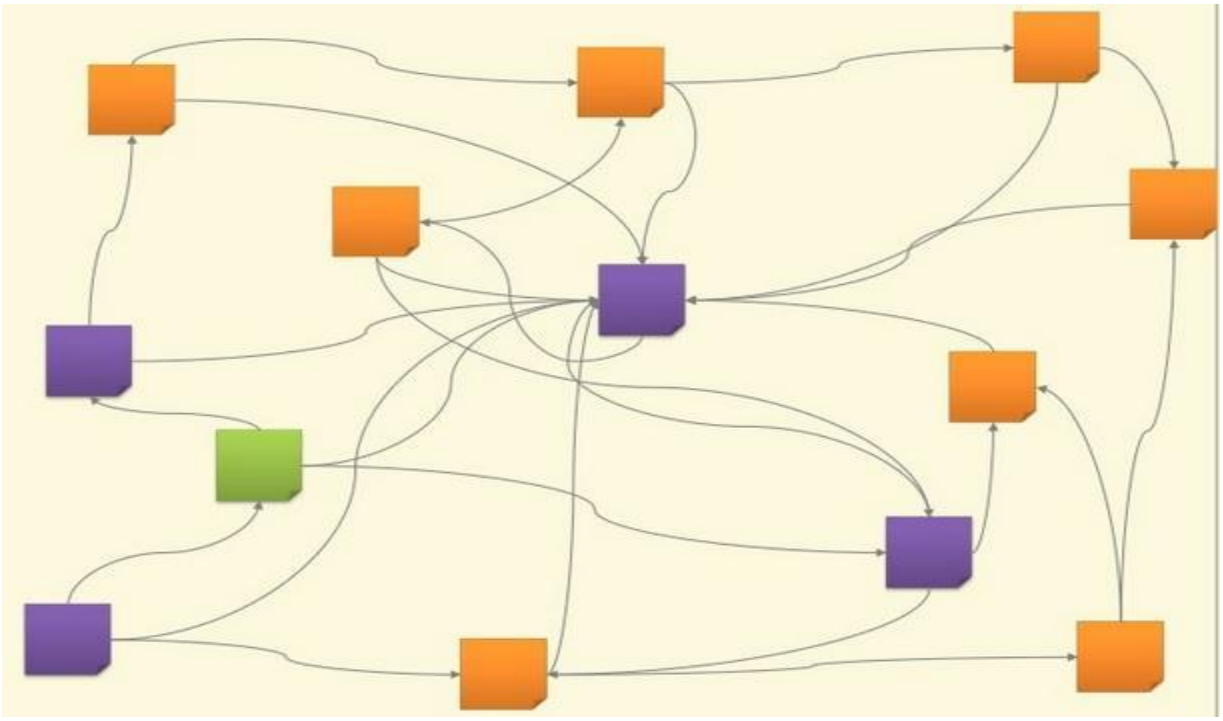
A internet ou Web como também é conhecida é tão utilizada por ser uma enorme rede onde todas as informações estão interligadas, através de hiperlinks.

Hateoas (Hypermidia As The Engine Of Application State), embora pareça ser um conceito novo, Hateoas já é utilizado na Web desde o princípio e nada mais é do que links que fazem uma ligação entre páginas ou estados de uma aplicação. Segundo Fielding quando o usuário avança em uma aplicação selecionando links (transições de estado) o resultado é uma próxima página, e essa nova página representa o próximo estado da aplicação.

A Web possui uma imensidão de links e tudo está conectado, um exemplo seria o google, um mecanismo de busca que oferece links para milhares de sites na Web.

Abaixo, imagem onde as páginas coloridas representam o estado da aplicação enquanto as setas que interligam as páginas representam a transição de estado da aplicação.

Figura 9 - Exemplo HATEOAS



Fonte: Jonathan Campos

A imagem acima é uma representação do conceito utilizado pelo Hateoas (Hypermidia As The Engine Of Application State), pois, é possível notar que a aplicação não é isolada, ou seja, a mesma possui links que podem levar o usuário a uma imensidão de estados diferentes, por estado entende-se que poderia ser uma página web, por exemplo.

Aplicações que seguem o conceito Restful em sua maioria não contém apenas dados, mas sim fazem uso da hipermídia contendo links para outros recursos.

Figura 10 - Exemplo XML HATEOAS

```
<aluno>
   <id>5454</id>
   <nome>José Fulano</nome>
   <link rel="notas" href="/aluno/5454/notas" />
   <link rel="horaria_aula" href="/aluno/5454/horaria_aula" />
</aluno>
```

Fonte: Jonathan Campos

O exemplo acima demonstra o uso de links onde através do resultado em XML do

aluno tem-se a opção de verificar suas notas e os seus horários de aula.

2.3 Segurança

Na web a segurança é uma parte fundamental pois várias tarefas que exigem segurança são utilizadas, como por exemplo, acesso a Internet Banking, utilização de cartão de crédito em compras online, enfim, várias informações importantes e confidenciais são trocadas a todo o momento na web, sendo assim, é necessário a utilização de ferramentas para encriptar e trafegar essas informações de forma segura. A segurança das informações trafegadas é muito importante pois caso não exista nenhum tipo segurança, qualquer usuário na rede será capaz de interceptar uma mensagem e utilizá-la como bem entender.

2.3.1 Autenticação

Segundo Boyd (2012) a autenticação é um processo que verifica a identidade de um usuário, confirmando se o mesmo é quem diz ser. Um exemplo no mundo real sobre autenticação é o processo de Check-in nos aeroportos onde é necessário se identificar mostrando um documento original com foto que comprove a identidade.

Independente de sistemas Web ou Desktop o processo de autenticação se dá através de um login, onde o usuário deve informar seu nome de usuário e senha para ter acesso ao sistema, dessa forma o sistema consegue identificar que o usuário é realmente quem diz ser.

2.3.2 Autorização

Segundo Boyd (2012) a autorização é um processo que verifica se o usuário tem a permissão para executar determinada ação, como por exemplo, visualizar um relatório de faturamento analítico. Em qualquer sistema independente da plataforma, o processo de autorização geralmente ocorre após o processo de autenticação, pois primeiro é necessário verificar a identidade do usuário e após isso verificar quais recursos do sistema o mesmo está

autorizado a utilizar.

Na seção a seguir serão descritos alguns métodos de autenticação que podem ser utilizados para aumentar a segurança no tráfego das informações.

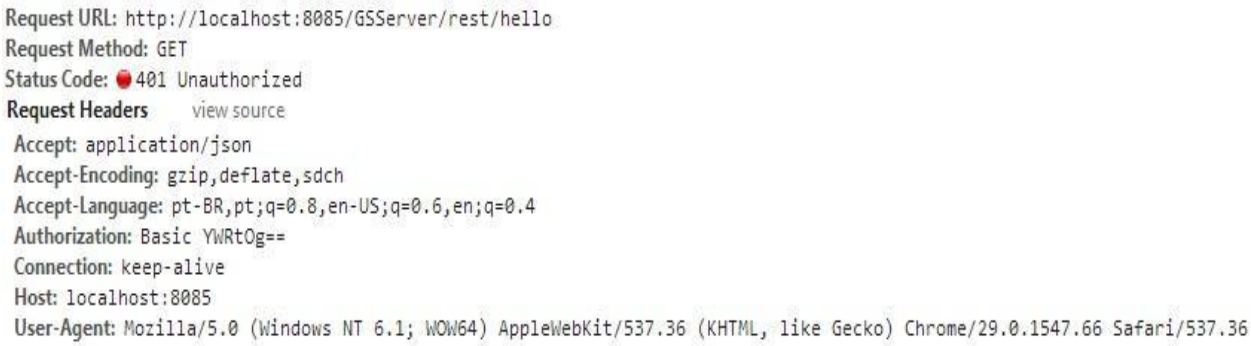
2.3.3 Autenticação Básica

A autenticação básica provê uma forma simples de autenticar um usuário sobre o protocolo HTTP (Boyd, 2012).

1. O Cliente envia o nome de usuário e um caractere dois pontos seguidos pela senha codificada em uma String Base64 no cabeçalho de autenticação do protocolo HTTP.
2. O servidor recebe o pedido, realiza a decodificação da String Base64 e verifica o usuário e senha informado no cabeçalho "Authorization" do protocolo HTTP e, caso o mesmo não exista retorna o status "!401 Unauthorized".

Abaixo, exemplo de uma requisição HTTP onde o servidor não conseguiu realizar a autenticação do Cliente.

Figura 11 - Exemplo Autenticação Básica



Fonte: Jonathan Campos

No exemplo acima é possível verificar através do cabeçalho que o tipo de autenticação é

a básica, ou seja, “Authorization: Basic” seguido do usuário e senha codificados em Base64 “YWRtOg==”.

Esse tipo de autenticação não é seguro pois se algum usuário mal intencionado na rede interceptar a requisição poderá facilmente decodificá-la para obter o usuário e senha e enviar suas próprias requisições utilizando o usuário e a senha obtida.

2.3.4 OAuth 2.0

OAuth (Open Authorization) é um protocolo aberto baseado em padrões IETF e licenciado pela Open Web Foundation que permite aos usuários de uma aplicação Web concederem acesso a seus recursos privados para uma aplicação externa, sem ter que compartilhar seu login e senha (Boyd, 2012).

O coração do OAuth é um token de autorização com direitos limitados, que o usuário pode revogar a qualquer momento (Boyd, 2012).

Segundo Dick Hardt o OAuth define quatro papéis, são eles:

I. Proprietário do Recurso

O proprietário do recurso pode ser qualquer entidade que conceda acesso a um recurso protegido. No caso de ser uma pessoa, o proprietário é o usuário final.

II. Servidor de Recursos

É o servidor que mantém os recursos protegidos hospedados e concede acesso aos mesmos através do uso de Tokens de acesso.

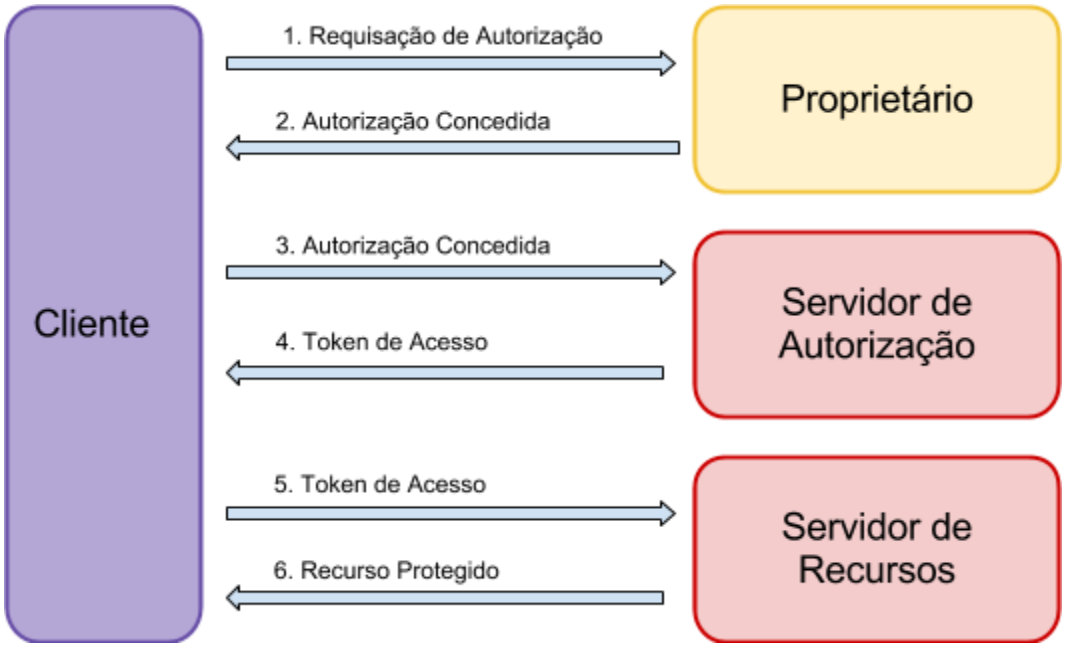
III. Cliente

O cliente é qualquer aplicação que faz solicitações de um recurso protegido em nome do proprietário do recurso e com a sua autorização.

IV. Servidor de Autorização

O servidor é o responsável por emitir o token de acesso para o cliente após a autenticação do proprietário do recurso.

Figura 12 - Funcionamento acesso OAuth



Fonte: Jonathan Campos

3. RestMB

Visando o desenvolvimento de uma API que será utiliza por dispositivos com sistema operacional Android a linguagem de programação adotada foi o Java, que é a linguagem nativa para desenvolvimento de aplicações Android.

O Java além de ser uma linguagem utilizada em vários tipos de dispositivos tais como celulares, micro-ondas dvds e etc., é também uma linguagem robusta e madura que possui uma grande quantidade de bibliotecas nativas que ajudam no desenvolvimento de aplicações (ORACLE, 2013).

A escolha do Android se deve pelo fato de ser um sistema muito utilizado em smartphones e tablets de custo acessível e que detém uma grande parte do mercado de dispositivos móveis.

3.1 Métodos e Materiais.

Para o desenvolvimento desse trabalho foi desenvolvida uma API REST visando facilitar assim a criação de aplicações clientes para consumir recursos de servidores baseados em REST, o objetivo da criação do RestMB foi a criação de classes que realizassem o encapsulamento das principais funcionalidades que uma aplicação deve ter, tais como: utilizar os verbos do protocolo HTTP (*Get, Put, Post, Delete, Head e Options*), *marshalling* e *unmarshalling* de dados.

A principal vantagem de se utilizar o RestMB está em não precisar escrever classes para realizar as requisições HTTP, apenas é necessário conhecer a url a ser acessada e qual o método HTTP a ser utilizado na requisição, além disso, não é necessário realizar o *unmarshalling* das informações para um tipo de objeto específico, a própria API já realiza essa conversão através de genéricos e reflexões.

Uma outra vantagem está em não depender de biblioteca (.jar) de terceiros como por exemplo as bibliotecas Gson do Google e Jackson, o RestMB possui todas as implementações necessárias para realizar requisições HTTP e fazer o *marshalling* e *unmarshalling* dos dados.

Abaixo serão descritos os pacotes utilizados para dividir as classes da API:

com.restmb

Contém as classes responsáveis por realizar a montagem de uma requisição e enviá-la a um servidor Rest, as principais classes serão descritas abaixo:

DefaultClient: Essa classe é responsável por conter os métodos que serão utilizados para realizar a comunicação com um servidor REST. Contém os métodos http (Get, Put, Post, Delete, Head e Options) além de métodos para informar o cabeçalho da requisição como setRequestProperty e também utilizar autenticação básica com o setBasicAuthentication. Essa classe irá chamar os métodos da classe a seguir.

DefaultMakeRequest: Classe responsável por construir a requisição montando a URI e os parâmetros que deverão ser enviados no cabeçalho da requisição. Após realizar todo o processo de montagem da requisição essa classe finalmente chamará o método correspondente na classe DefaultWebRequestor que será descrito a seguir.

DefaultWebRequestor: Essa classe contém os métodos que são utilizados para executar uma requisição a um servidor Rest, seus principais métodos são: executeGet, executePost, executePut, executeDelete, getInputStream.

DefaultJsonMapper: Essa classe é utilizada para mapear um objeto *JSON* para objeto Java e vice versa.

com.restmb.annotation

Contém as anotações que são utilizadas pelo RestMB para montar a Url de requisição ao servidor Rest e para mapear os campos de um Json para um objeto Java, as anotações são @Endpoint e @RestMB.

com.restmb.authentication

Contém a classe responsável por armazenar o usuário e senha que serão convertidos para Base64 e enviados junto ao cabeçalho http da requisição ao servidor Rest.

com.restmb.exception

Contém as classes responsáveis por tratar as exceções e apresentar uma mensagem amigável à aplicação que está utilizando a API.

RestMBException: Classe abstrata que herda de RuntimeException utilizada para ser a superclasse de todas as classes que tratam exceção na API RestMB.

RestMBNetworkException: Utilizada para lançar (*Throw*) uma exceção caso ocorra algum problema ao realizar uma requisição a um servidor Rest.

RestMBJsonMappingException: Utilizada para lançar (*Throw*) uma exceção caso ocorra algum problema ao tentar realizar a conversão de um JSON para um objeto Java ou um objeto Java para um objeto JSON.

com.restmb.json

Contém as classes necessárias para realizar o marshalling e unmarshalling de dados, trabalha tanto com objetos quanto com array de objetos.

com.restmb.util

Esse pacote contém classes que possuem métodos que tem por objetivo facilitar a execução de alguns procedimentos dentro da API, um exemplo é a classe CreateInstance, a mesma é utilizada para criar uma instância de uma classe genérica passada como parâmetro para o método “get” de DefaultClient para assim obter as anotações pertencentes a classe.

Uma outra classe muito utilizada dentro da API é a StringUtils, os métodos de maior destaque dentro desse pacote são:

fromInputStream: Responsável por transformar o resultado de uma requisição a um servidor REST (Response Body) em uma String.

inputStreamToByte: Responsável por transformar o resultado de uma requisição a um servidor REST (Response Body) em um array de bytes, muito útil para buscar imagens e arquivos.

fromMapHeader: Responsável por transformar o cabeçalho (Header) de uma requisição a um servidor REST em uma String.

base64Encoder: Utilizado para codificar uma String de dados em Base64, esse método é utilizado para autenticação básica.

3.2 Recurso da API

A fim de facilitar a integração com serviços Rest a API disponibiliza alguns recursos muito úteis para o desenvolvedor utilizar que serão descritos no tópicos abaixo:

3.2.1 Anotações

O uso de anotações teve inicio a partir da versão 5 do Java e tem como objetivo servir como a declaração de metadados para os objetos de uma aplicação, além do mais servem também para fornecer informações ao compilador, informações em tempo de compilação e em tempo de execução.

Um exemplo muito comum de anotação utilizada pelo Java é o @Override que indica que o método assinado está sobrescrevendo um método de mesmo nome de sua superclasse.

Visando facilitar o trabalho da API RestMB foram criados dois tipos de anotações que serão explicadas a seguir:

@EndPoint

A anotação @Endpoint é utilizada em uma classe para possibilitar a definição do endereço da requisição a ser feita no servidor. Com isso a API RestMB será capaz de montar uma requisição e enviá-la ao servidor. Abaixo segue imagem de uma classe demonstrando o exemplo de uso da anotação @Endpoint.

Figura 13 – Exemplo de uso da anotação @Endpoint

```
6  @Endpoint("/cliente")
7  public class Cliente {
8      @RestMB
9      private long id;
10
11     @RestMB
12     private String name;
13
14     @RestMB
15     private String endereco;
16
17     //Getters e Setters
```

Fonte: Jonathan Campos

@RestMB

A anotação @RestMB é utilizada nos atributos de uma classe e através dela a API RestMB poderá identificar quais os atributos deverão ser utilizados para realizar o mapeamento de um resultado *Json* para a classe passada como parâmetro para o método “get” de um objeto “DefaultClient”.

Figura 14 – Exemplo de uso GET

```
DefaultClient restmbClient = new DefaultClient(“http://www.exemplo.com”);

Produto p = restmbClient.get(“/produto”, Produto.class);
```

Fonte: Jonathan Campos

Abaixo, exemplo de uma classe que contém todos os seus atributos marcados com a anotação @ RestMB:

Figura 15 - Exemplo de uso da anotação @RestMB

```
6  @Endpoint("/produto")
7  public class Produto {
8      @RestMB
9      private long id;
10
11     @RestMB
12     private String descricao;
13
14     @RestMB
15     private double valor;
16
17     //Getters e Setters
```

Fonte: Jonathan Campos

A classe acima possui os atributos id, descricao e valor, o seguinte Json poderia ser mapeado para essa classe:

Figura 16 - String JSON de Produto

```
{
    "id": "1567899",
    "descricao": "GALAXY TAB 7 P3110",
    "valor": "535.00"
}
```

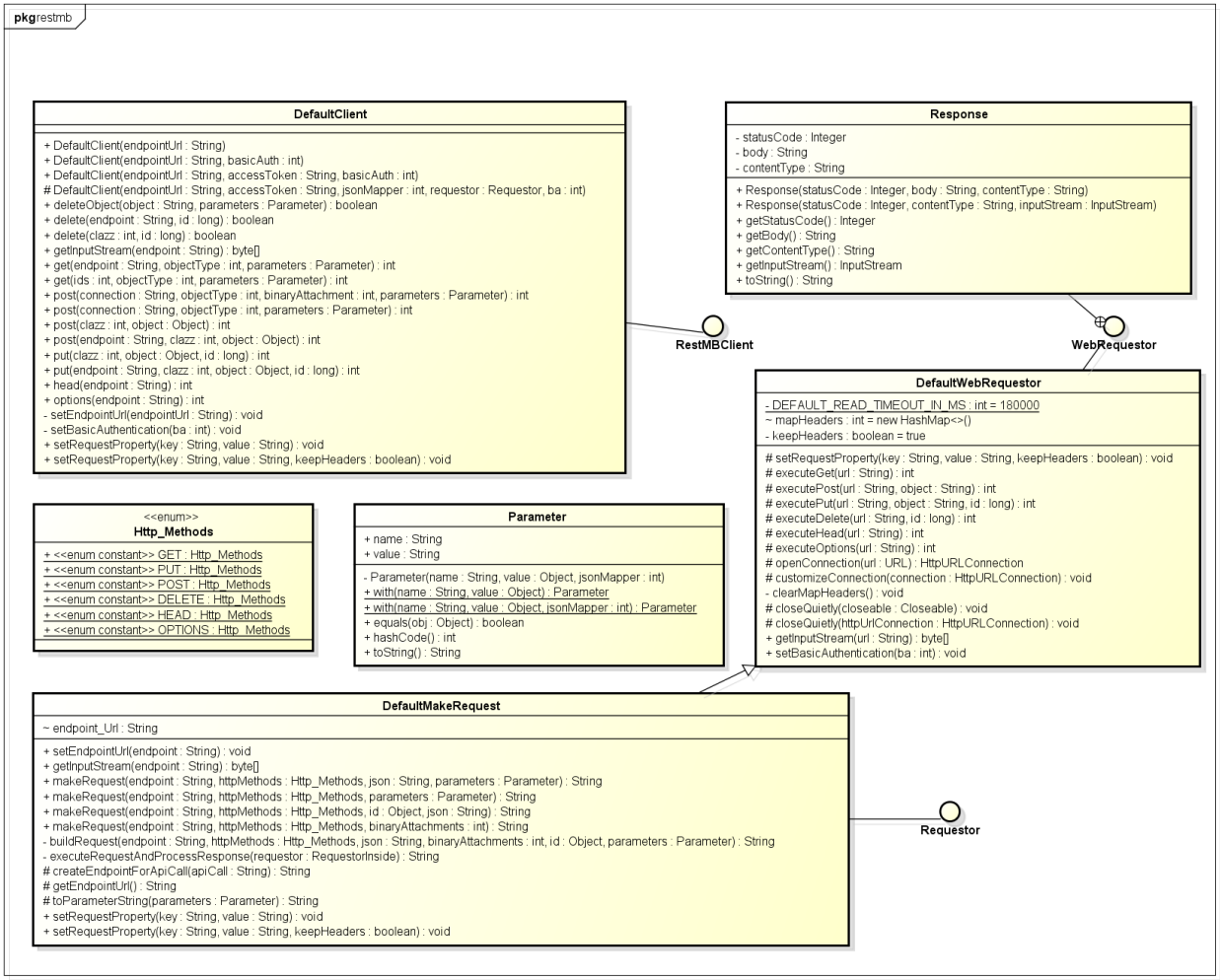
Fonte: Jonathan Campos

É através dessa anotação que a API é capaz de realizar *marshalling* e *unmarshalling* dos dados, ou seja, um objeto Java será transformado em um objeto *Json* para ser enviado ao servidor e um objeto *Json* será transformado no tipo de objeto passado como parâmetro ao método get da classe DefaultClient como na Figura 14.

3.2.2 Requisição HTTP (GET, PUT, POST, DELETE, OPTIONS e HEAD)

Abaixo será mostrada uma imagem que representa o diagrama de classe das principais classes responsáveis por montar uma requisição ao servidor Rest e devolver uma informação para o cliente.

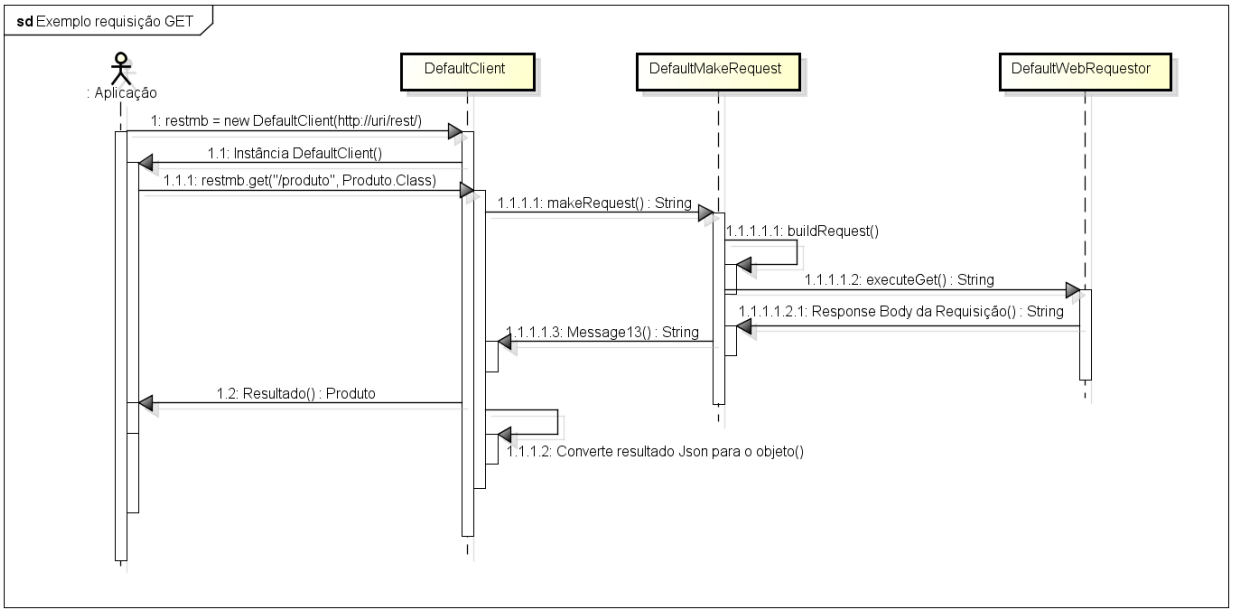
Figura 17 – Diagrama de classe RestMB.



Fonte: Jonathan Campos

Abaixo será mostrada uma imagem que representa o diagrama de sequência do método GET da API Rest MB.

Figura 18 - Diagrama de Sequência GET



Fonte: Jonathan Campos

A API RestMB seguindo os princípios Rest provê alguns métodos Java que fazem uso dos principais métodos do protocolo Http que serão demonstrados a seguir:

GET

O método “get” provê uma forma simples e rápida para realizar uma requisição, pois, o mesmo encapsula toda a parte de montagem da requisição e comunicação com o servidor *Rest* e também o *unmarshall* de dados, ou seja, ao receber uma resposta do servidor que virá em um formato *Json* o mesmo deve ser convertido para um tipo de objeto especifico, o tipo de objeto será o mesmo passado como um dos argumento do método “get”, como por exemplo Cliente.class.

Abaixo, exemplo de uma requisição GET que retornará um objeto do tipo Cliente:

Figura 19 - Exemplo RestMB GET

```
6  public class GetExample {
7      DefaultClient client = new DefaultClient("http://localhost:8085/Server_TCC-2013/rest/");
8
9      public void get() {
10         Cliente c = client.get("/client", Cliente.class);
11         System.out.println(c.getId());
12         System.out.println(c.getRazao());
13         System.out.println(c.getEndereco());
14     }
15 }
```

Fonte: Jonathan Campos

Na imagem acima é possível notar na linha 10 que "client" recebe como um de seus parâmetros uma classe e retorna um objeto do mesmo tipo da classe passada como parâmetro, isso é possível devido o uso da anotação @RestMB que foi vista no tópico anterior desse trabalho.

PUT

O método "put" visa disponibilizar também uma forma simples e rápida para se atualizar um recurso em um servidor *Rest*. É possível enviar um objeto e a própria API se encarrega de realizar o *marshall* de dados para o formato Json e enviar para o servidor Rest.

Abaixo, exemplo de uma requisição PUT que retornará um objeto do tipo Cliente:

Figura 20 - Exemplo RestMB PUT

```
9      public void put() {
10         client.setRequestProperty("Content-Type", "application/json");
11
12         Cliente cli = new Cliente();
13         cli.setRazao("TESTE");
14
15         /***************PUT informando o Endpoint *******/
16         cli = client.put("/client", Cliente.class, cli, 35);
17
18         System.out.println(cli.getId());
19         System.out.println(cli.getRazao());
20         System.out.println("============================");
21         /*********************************************/
22
23         /************PUT sem informar o Endpoint *******/
24         cli.setRazao("TESTE2");
25         cli = client.put(Cliente.class, cli, 35);
26
27         System.out.println(cli.getId());
28         System.out.println(cli.getRazao());
29         /*********************************************/
30     }
```

Fonte: Jonathan Campos

No exemplo acima, é possível ver o método “put” sendo utilizado de duas formas, são elas:

- Na linha 16 o primeiro argumento do método é o “/cliente” que identifica qual o nome do recurso (endpoint) no servidor Rest.
- Na linha 25 não é passado o nome do recurso (endpoint) como parâmetro, isso é possível devido ao uso da anotação @Endpoint, pois, a mesma é utilizada para esse fim, com isso não é necessário escrever o endpoint em todas as classes que forem realizar o uso do método “put” para atualizar um cliente.

POST

O método “post” visa facilitar a criação de um recurso no servidor *Rest* o mesmo é muito semelhante ao método “put” possuindo a mesma lista de parâmetros e a mesma lógica para realizar o retorno. Uma funcionalidade interessante é que ao criar um recurso é possível receber um objeto do mesmo tipo do recurso criado, isso é necessário porque os sistemas baseados em *Rest* em sua maioria possuem um id único gerado pelo servidor, com isso ao criar um recurso é possível recebe-lo de volta já com seu id gerado.

Abaixo, exemplo de uma requisição POST que retornará um objeto do tipo Cliente:

Figura 21 - Exemplo RestMB POST

```
9    public void post() {
10       client.setRequestProperty("Content-Type", "application/json");
11
12       Cliente cli = new Cliente();
13       cli.setRazao("JOSE ALDO");
14
15       /***************POST informando o Endpoint *******/
16       cli = client.post("/client", Cliente.class, cli);
17
18       System.out.println(cli.getId());
19       System.out.println(cli.getRazao());
20       System.out.println("============================");
21       /*************************************************/
22
23       /************POST sem informar o Endpoint *******/
24       cli.setRazao("NEYMAR JR.");
25       cli = client.post(Cliente.class, cli);
26
27       System.out.println(cli.getId());
28       System.out.println(cli.getRazao());
29       /*************************************************/
30   }
```

Fonte: Jonathan Campos

DELETE

O método “delete” como o nome sugere foi criado para realizar a exclusão de um recurso no servidor *Rest*. Assim como os outros métodos citados acima, o mesmo possui duas formas, a primeira na qual é possível informar qual o *endpoint* do recurso e a segundo sem informar o *endpoint*. O método delete retorna um valor booleano indicando se o recurso foi excluído com sucesso ou não.

Abaixo exemplo de uma requisição DELETE que retornará um valor booleano:

Figura 22 - Exemplo RestMB DELETE

```
9   public void delete() {
10      boolean result;
11
12      /***************DELETE informando o Endpoint *******/
13      result = client.delete("/client", 35);
14      System.out.println(result);
15      System.out.println("============================");
16      /*********************************************/
17
18      /**********DELETE sem informar o Endpoint *******/
19      result = client.delete(Cliente.class, 35);
20      System.out.println(result);
21      /*********************************************/
22
23  }
```

Fonte: Jonathan Campos

HEAD

O método “head” é muito semelhante ao método “get” porém o mesmo não retorna o corpo (*body*) da requisição e sim apenas o cabeçalho, muito útil quando se deseja saber algumas informações sobre o recurso a ser solicitado como: *Content-Length, Last-Modified e Content-Type* por exemplo.

Abaixo, exemplo de uma requisição HEAD que retornará uma String contendo os valores do cabeçalho (*head*):

Figura 23 - Exemplo RestMB HEAD

```
5  public class HeadExample {
6      DefaultClient client = new DefaultClient("http://masterstudioweb.com.br/gustavo/wp-includes/images");
7
8      public void head() {
9          System.out.println("HEAD");
10         String result = client.head("/android-google.jpg");
11         System.out.println(result);
12     }
13 }
```

Fonte: Jonathan Campos

Abaixo, o retorno da requisição feita utilizando o método “head”:

Figura 24 - Cabeçalho HTTP Head

Accept-Ranges → bytes

Cache-Control → max-age=1800

Connection → Keep-Alive

Content-Length → 1488617

Content-Type → image/jpeg

Date → Mon, 04 Nov 2013 22:55:44 GMT

Keep-Alive → timeout=2, max=100

Last-Modified → Wed, 15 Feb 2012 09:46:09 GMT

Server → Apache

Vary → Accept-Encoding

X-Powered-By → W3 Total Cache/0.9.2.4

Fonte: Jonathan Campos

OPTIONS

O método OPTIONS é responsável por retornar uma String contendo quais os métodos Http aceitos para a manipulação de um recurso.

Abaixo, exemplo de uma requisição OPTIONS que retornará uma String contendo os valores aceitos pelo servidor:

Figura 25 - Exemplo RestMB OPTIONS

```
5  public class OptionsExample {
6      DefaultClient client = new DefaultClient("http://masterstudioweb.com.br/gustavo/wp-includes/images");
7
8      public void options() {
9          String result;
10
11         System.out.println("OPTIONS");
12         result = client.options("/android-google.jpg");
13         System.out.println(result);
14
15     }
16 }
```

Fonte: Jonathan Campos

Abaixo, o retorno da requisição feita utilizando o método “options”:

Figura 26 - Cabeçalho HTTP Options

```
Allow → GET,HEAD,POST,OPTIONS
```

Fonte: Jonathan Campos

3.2.3 Parâmetros na Url

A API RestMB possibilita a adição de parâmetros na URL através de um argumento vargs passado como parâmetro a um método de requisição. Abaixo, exemplo da utilização de parâmetros em uma URL utilizando a API:

Figura 27 – Utilização de Parâmetros na URL

```
DefaultClient restmbClient = new DefaultClient(“http://www.exemplo.com”);
Cliente c = client.get("/client", Cliente.class,
                        Parameter.with("parametro1", "valor1"),
                        Parameter.with("parametro2", "valor2"),
                        Parameter.with("parametro3", "valor3") );
```

Fonte: Jonathan Campos

Nota-se no exemplo acima que são passados 3 argumentos, mas poderiam ser passados quantos argumentos fossem necessários, isso é possível porque o java possui o varargs uma

funcionalidade que permite a um método receber uma lista variável de argumentos (SIERRA, 2006).

HTTP Header Params

A API RestMB possibilita a adição de parâmetros no cabeçalho HTTP através do método “setRequestProperty()” o mesmo recebe dois parâmetros do tipo String que serão utilizados para montar os parâmetros que serão enviados no cabeçalho da requisição HTTP ao servidor Rest. Abaixo exemplo de como adicionar um parâmetro utilizando a API RestMB:

Figura 28 – Adicionando Http Headers

```
DefaultClient restmbClient = new DefaultClient(“http://www.exemplo.com”);
client.setRequestProperty("Accept", "application/json");
```

Fonte: Jonathan Campos

3.2.4 Autenticação Básica

A API RestMB também tem a opção de utilizar o tipo de autenticação básica, para realizar requisições ao servidor Rest.

A própria API se encarrega de realizar a codificação para Base64 bastando que se passe um nome de usuário e senha para que a mesma realize a codificação. Abaixo, exemplo de utilização da API para que se realize uma requisição utilizando Autenticação Básica:

Figura 29 – Autenticação Básica RestMB

```
DefaultClient client =
        new DefaultClient("http://localhost:8085/Server_TCC-2013/rest/",
                          new BasicAuthentication("univem", "123"));
```

Fonte: Jonathan Campos

4. Resultados

Visando realizar testes para verificação de desempenho e o ciclo completo de uma requisição utilizando a API RestMB foram utilizados dois ambientes de testes, sendo o primeiro em um servidor local e o segundo consumindo recursos do Facebook.

4.1 Servidor Local

Para realizar os testes de desempenho em um servidor local foi criado um Web Service Rest utilizando o Jersey. O Jersey é uma API open source para Java que possibilita a criação de Web Services Restful de maneira rápida e simplificada (JERSEY, 2013).

Foi criada a classe cliente que representa um recurso a ser buscado no servidor local conforme figura 30 abaixo:

Figura 30 – Classe Cliente Servidor Local

```
10 @XmlRootElement (name="cliente")
11 public class Cliente {
12
13     @XmlElement
14     private String id;
15     @XmlElement
16     private String cgcCpf;
17     @XmlElement
18     private String inscrEstadual;
19     @XmlElement
20     private String endereco;
21     @XmlElement
22     private String codigonomecidade;
23     @XmlElement
24     private String estado;
25     @XmlElement
26     private String bairro;
27     @XmlElement
28     private String telefone;
29     @XmlElement
30     private String cep;
31     @XmlElement
32     private String nomefantasia;
33
34     //getters e setters
```

Fonte: Jonathan Campos

A classe acima possui algumas anotações importantes que fazem parte do JAXB (Java for XML Binding) que é um padrão Java que descreve como objetos Java serão convertidos para XML e vice e versa (ORACLE, 2013).

Abaixo, descrição das anotações da classe exibida na figura 30:

@XmlRootElement – Usado para definir o elemento raiz para uma árvore XML.

@XmlElement – Esse elemento precisa ser utilizado apenas quando o nome do elemento que vier no resultado JSON ou XML for diferente do nome JavaBeans.

A figura 31 abaixo mostra a classe criada no Web Service que representa o recurso cliente.

Figura 31 – Classe representando recurso no Servidor Local

```
21 @Path("/cliente")
22 public class ClienteResources {
23     @GET
24     @Produces("application/json")
25     public Response getCliente() {
26         ResponseBuilder builder = Response.status(Status.NOT_FOUND);
27 
28         Clientes clientes = new Clientes();
29 
30         ArrayList<Cliente> all = clientes.getAll();
31         clientes.list = new ArrayList<Cliente>();
32 
33 
34         for (Cliente c : all)
35             clientes.list.add(c);
36 
37         if(clientes.list.size() > 0)
38             builder.status(Status.OK).entity(clientes);
39 
40         return builder.build();
41     }
42     @GET
43     @Path("{id}")
44     @Produces("application/json")
45     public Response getClienteId(@PathParam("id") String id) {
46         ResponseBuilder builder = Response.status(Status.NOT_FOUND);
47 
48         ClienteDAO cDao = new ClienteDAO();
49         Cliente c = cDao.getById(id);
50 
51         builder.status(Status.OK).entity(c);
52 
53         return builder.build();
54     }
55 }
```

Fonte: Jonathan Campos

O Jersey faz uso de anotações para tratar algumas informações como apresentado na figura 31. As anotações serão descritas abaixo:

@Path - O valor dessa anotação é um caminho URI relativo. Na figura 31, a classe Java será hospedada na URI /cliente. É possível notar também que o método getClienteId também possui uma assinatura @Path, porém a mesma possui um parâmetro que é utilizado para indicar o ID do recurso como em “/cliente/098790” (JERSEY, 2013).

@Get – Essa anotação corresponde ao método de mesmo nome do protocolo HTTP, ou seja, o método anotado com @Get irá processar requisições HTTP GET.

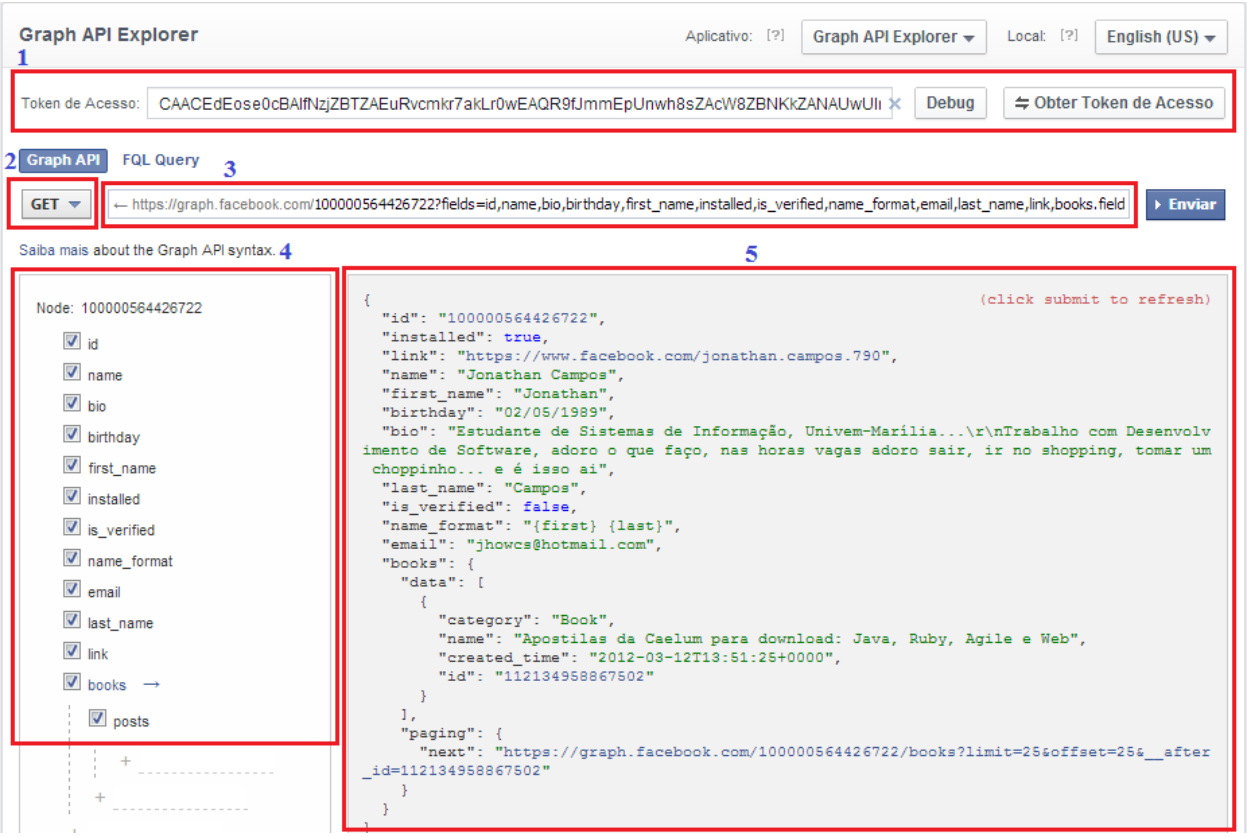
@Produces – Essa anotação é usada para especificar o tipo de MIME que um recurso irá produzir e enviar de volta para o cliente tais como: JSON, XML, HTML, TEXT entre outros.

4.2 Servidor Remoto (Facebook - Graph Facebook API)

Para realizar os testes em um servidor remoto foi utilizado o Facebook, o mesmo disponibiliza uma interface que possibilita a geração de um Token de Acesso e também visualizar as possíveis URIs disponíveis para consumo, tais como: informações sobre amigos, fotos, últimos posts entre outros.

A Figura 31 mostra a interface do Facebook Graph API, a mesma foi dividida em partes que serão explicadas a seguir.

Figura 32 – Interface Facebook Graph API



Fonte: Jonathan Campos

1- Exibe o Token de acessso que deverá ser utilizado por uma aplicação em desenvolvimento que deseja consumir recursos do Facebook, esse Token tem um tempo de vida e ao expirar deve-se obter um novo.
2- No exemplo, o método HTTP utilizado é o GET.
3- Representa a URL completa do recurso.
4- Representa todos os campos do recurso que serão buscados no servidor Rest do Facebook.
5- Exibe o resultado no formato JSON da consulta ao servidor REST.

4.3 Teste

Foram feitos testes em relação ao *unmarshalling* dos dados para verificar o desempenho tanto utilizando a API RestMB como utilizando a API Gson do Google.

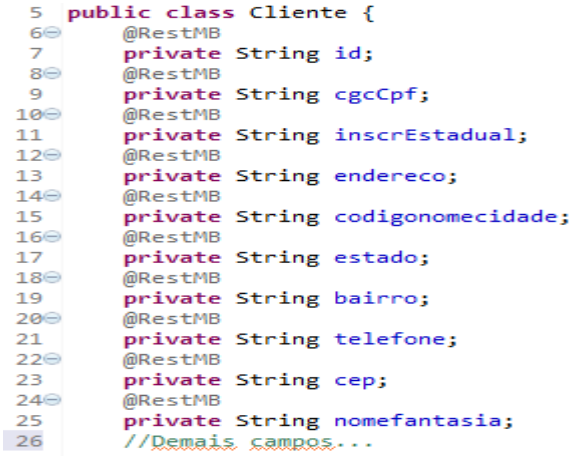
Os critérios utilizados para avaliação foram:

1- Consumir um recurso que retorne um objeto contendo 50 campos.
2- Consumir um recurso que retorne uma lista de objetos contendo 50 campos.
3- Consumir um recurso que retorne imagem.

4.3.1 Local

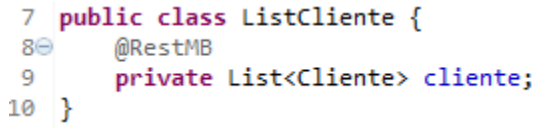
Para realizar o teste local foi criada uma classe de cliente conforme a figura 33 e uma classe que representa uma lista de clientes conforme a figura 34.

Figura 33 – Classe Cliente utilizando anotada pela API RestMB



Fonte: Jonathan Campos

Figura 34 – Classe Lista de objeto Cliente anotada pela API RestMB



Fonte: Jonathan Campos

Na figura 35 será exibido o trecho do código utilizado para realizar a busca de um recurso no servidor local chamado /cliente/042590.01 fazendo uso da API RestMB.

Figura 35 – Código utilizando RestMB

```
65        @Override
66        protected Void doInBackground(Void... params) {
67            long tempoInicial = System.currentTimeMillis();
68
69            DefaultClient client = new DefaultClient("http://192.168.0.164:8085/Server_TCC-2013/rest");
70
71            Cliente cliente = client.get("/cliente/042590.01", Cliente.class);
72
73            long tempoFinal = System.currentTimeMillis();
74            Log.d("TEMPO_TOTAL", "" + (tempoFinal - tempoInicial) );
75
76            return null;
77        }
```

Fonte: Jonathan Campos

Na figura 36 será exibido o trecho do código utilizado para realizar a busca de um recurso no servidor local /cliente/042590.01, porém a requisição será feita de forma manual e o *unmarshalling* de dados será realizado pela API Gson do Google.

Figura 36 – Código fazendo requisição manual

```
131       @Override
132       protected InputStream doInBackground(Void... params) {
133           InputStream in = null;
134           try {
135               long tempoInicial = System.currentTimeMillis();
136
137               URL url = new URL("http://192.168.0.164:8085/Server_TCC-2013/rest/cliente");
138               URLConnection conn = url.openConnection();
139
140               if(!(conn instanceof HttpURLConnection))
141                   throw new IOException("Não é uma conexão HTTP");
142
143               HttpURLConnection htc = (HttpURLConnection) conn;
144               htc.setRequestMethod("GET");
145               htc.connect();
146               int response = htc.getResponseCode();
147
148               if(response == HttpURLConnection.HTTP_OK)
149                   in = htc.getInputStream();
150
151               if(in != null) {
152                   final Gson gson = new Gson();
153                   final BufferedReader reader =
154                       new BufferedReader(new InputStreamReader(in));
155                   Cliente c = gson.fromJson(reader, Cliente.class);
156                   long tempoFinal = System.currentTimeMillis();
157                   Log.d("TEMPO_TOTAL", "" + (tempoFinal - tempoInicial) );
158               }
159           } catch (MalformedURLException e) {
160               // TODO Auto-generated catch block
161               e.printStackTrace();
162           } catch (IOException e) {
163               // TODO Auto-generated catch block
164               e.printStackTrace();
165           }
166           return in;
167       }
```

Fonte: Jonathan Campos

Na figura 37 será exibida uma parte do objeto JSON retornado do servidor local, o objeto JSON em seu formato original possui 50 campos.

Figura 37 – Exemplo de parte do retorno JSON


```
{
    "cliente": [
        {
            "id": "046005.01",
            "cgcCpf": "17777465000156",
            "inscrEstadual": "582959490111",
            "endereco": "R ATTILIO PEDRO CHERUBIM, 210",
            "codigonomecidade": "RIBEIRAO PRETO-SP",
            "estado": "SP",
            "bairro": "DOM BERNARDO JOSE MIELLE",
            "telefone": "36390108",
            "cep": "14057310",
            "nomefantasia": "MERCEARIA COELHO E LOPES LTDA - ME",
```


Fonte: Jonathan Campos

4.3.2 Remoto (Facebook)

Para realizar o teste remoto foi criada uma classe chamada Facebook conforme figura 38 e uma classe que representa uma lista Facebook conforme figura 39

Figura 38 – Classe representando dados do Facebook

```
public class Facebook {

    @RestMB("id")
    private long id;

    @RestMB("name")
    private String name;

    @RestMB("first_name")
    private String first_name;

    @RestMB("last_name")
    private String last_name;

    @RestMB("name_format")
    private String name_format;

    @RestMB("link")
    private String link;

    @RestMB("birthday")
    private String birthday;

    @RestMB("bio")
    private String bio;

    @RestMB("is_verified")
    private String is_verified;

    @RestMB("gender")
    private String gender;
```

Fonte: Jonathan Campos

Figura 39 – Classe representando uma lista de objetos Facebook

```
9       @RestMB
10      private List<Facebook> friends;
11
12      public List<Facebook> getFriends() {
13          return friends;
14      }
15  }
```

Fonte: Jonathan Campos

Na figura 40 será exibido o trecho do código utilizado para realizar a busca de um recurso no Facebook fazendo uso da API RestMB.

Figura 40 - Código fazendo requisição ao Facebook com o RestMB

```
82      @Override
83      protected Void doInBackground(Void... params) {
84          long tempoInicial = System.currentTimeMillis();
85          final String FIELDS = "id,name,first_name,last_name,name_format,link,birthday,bio,is_verified,gender";
86          final String ACCESS_TOKEN = "CAACEdEose0cBAI9ubFspzvMFJQIqqqzq0U75lYwFsqu1SDVMhYFkH2fgYNc6t443OZC3xgkhbK
87
88          Facebook faceFields =
89                  client.get(String.format("/100000564426722/?fields=%s&access_token=%s",
90                          FIELDS, ACCESS_TOKEN), Facebook.class);
91
92          long tempoFinal = System.currentTimeMillis();
93          Log.d("TEMPO_TOTAL", "" + (tempoFinal - tempoInicial) );
94
95          return null;
96      }
97  }
```

Fonte: Jonathan Campos

Na figura 41 será exibido o trecho do código utilizado para realizar a busca de um recurso no servidor no facebook, porém a requisição será feita de forma manual e o *unmarshalling* de dados será realizado pela API Gson do Google.

Figura 41 - Código fazendo requisição ao Facebook sem RestMB

```
75      @Override
76      protected Facebook doInBackground(Void... params) {
77          final String FIELDS = "id,name,first_name,last_name,name_format,link,birthday,bio,is_verified,gender";
78          final String ACCESS_TOKEN = "CAACEdEose0cBAI9ubFspzvMFJQIqqqzq0U75lYwFsqu1SDVMhYFkH2fgYNc6t443OZC3xgkhl
79          Facebook face = null;
80          tempoInicial = System.currentTimeMillis();
81          InputStream in = null;
82          try {
83              URL url = new URL(String.format("/100000564426722/?fields=%s&access_token=%s",
84                                              FIELDS, ACCESS_TOKEN));
85              URLConnection conn = url.openConnection();
86
87              if(!(conn instanceof HttpURLConnection))
88                  throw new IOException("Não é uma conexão HTTP");
89
90              HttpURLConnection htc = (HttpURLConnection) conn;
91              htc.setRequestMethod("GET");
92              htc.connect();
93              int response = htc.getResponseCode();
94
95              if(response == HttpURLConnection.HTTP_OK)
96                  in = htc.getInputStream();
97
98              if(in != null) {
99                  Gson gson = new Gson();
100                 final BufferedReader reader =
101                     new BufferedReader(new InputStreamReader(in));
102                 face = gson.fromJson(reader, Facebook.class);
103                 tempoFinal = System.currentTimeMillis();
104                 Log.d("TEMPO_TOTAL", "" + (tempoFinal - tempoInicial) );
105             }
106         } catch (MalformedURLException e) {
107             // TODO Auto-generated catch block
108             e.printStackTrace();
109         } catch (IOException e) {
110             // TODO Auto-generated catch block
111             e.printStackTrace();
112         }
113         return face;
114     }
```

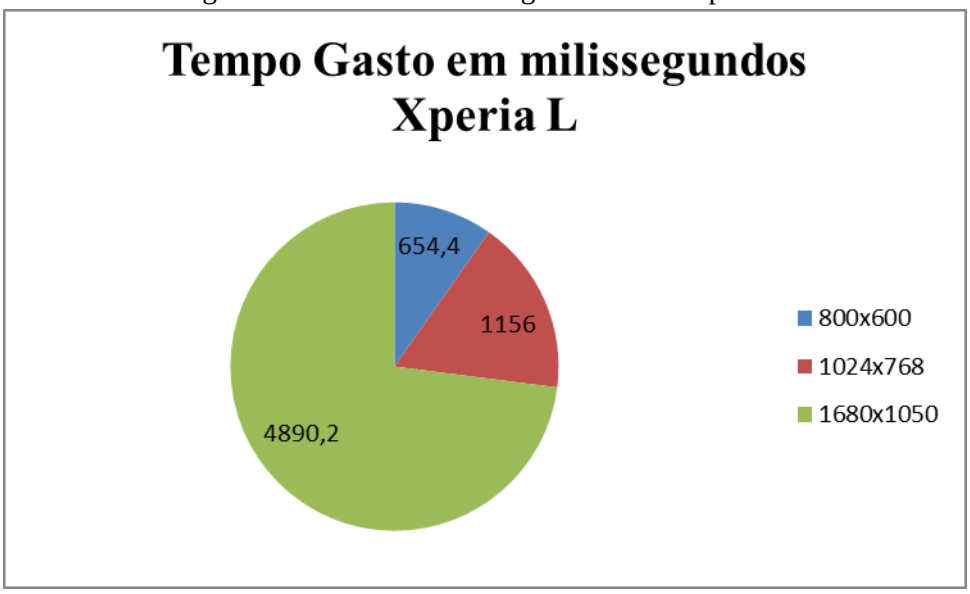
Fonte: Jonathan Campos

Para realizar uma requisição que retornasse imagem foram utilizadas imagens idênticas, porém com resoluções diferentes conforme listado abaixo:

800x600, 1024x768 e 1680x1050.

Para cada imagem foram realizados 5 testes medindo o tempo de download, após isso foi feita uma média do tempo gasto conforme gráfico abaixo.

Figura 42 – Consumo de Imagens RestMB Xperia L



Fonte: Jonathan Campos

Análise dos Resultados

Para avaliação dos resultados foram realizados dois ambientes de testes conforme explicado no tópico 4 deste trabalho e também o consumo de imagens com diferentes resoluções.

Tabela 4 – Resultado de unmarshalling de dados de um Objeto

	Desempenho	**Produtividade**
RestMB	Bom	**Bom**
Gson	**Bom**	**Regular**

Fonte: Jonathan Campos

Tabela 5 - Resultado de unmarshalling de dados de uma lista de Objetos

	Desempenho	**Produtividade**
RestMB	Regular	**Bom**
Gson	**Bom**	**Regular**

Fonte: Jonathan Campos

Foi possível notar que a API realizou de forma aceitável os testes para consumo de objetos pequenos e imagens, porém a mesma não se mostrou eficiente ao realizar o *unmarshalling* de uma lista com vários objetos, o teste realizado foi feito retornando uma lista com 100 objetos contendo 50 campos cada.

Com relação às funcionalidades comparando com outras soluções.

Gson: É uma biblioteca Java que pode ser utilizada para realizar a conversão de objetos Java para a representação JSON e também pode ser utilizado para converter uma string JSON para um objeto Java equivalente (GSON, 2013). Possui uma documentação completa e uma vasta lista de exemplos na Web.

Android Asynchronous Http Client: É um cliente Http assíncrono construído utilizando as bibliotecas do Apache HttpClient, seus pedidos são feitos fora do seguimento da UI, não sendo necessário que o desenvolvedor se preocupe em criar uma thread para chamar o método executor de requisição ao servidor (SMITH, 2013).

Volley: É uma biblioteca apresentada por Ficus Kirkpatrick no Google I/O 2013 que promete deixar a comunicação de rede nas aplicações Android mais fácil e o mais importante mais rápida, a biblioteca gerencia o processamento e armazenamento em cache de solicitações de rede. A biblioteca fornece métodos prontos para realizar a comunicação com um servidor Rest com isso gerando economia na quantidade de código escrito pelo desenvolvedor (KIRKPATRICK, 2013).

Abaixo tabela com a exibição dos recursos disponíveis pela API RestMB comparado a outras soluções disponíveis no mercado.

Tabela 6 – Comparação com outras Soluções

	RestMB	Gson	Loopj	Volley
Suporte a paginação (Nativo)	Não	Não	Não	Sim
Requisição http Assíncronas	Não	Não	Sim	Sim
Uso de Anotações	Sim	Sim	Não	Não
Autenticação	Sim	Não	Sim	Sim
Unmarshalling/Marshalling	Sim	Sim	Não	Sim (Gson)
RESTful	Sim	Não	Não	Sim
Tratamento de Erro	Sim	Sim	Sim	Sim
Dependência de API	Não	Não	Não	Sim

Fonte: Jonathan Campos

Na Tabela 6 é possível observar as funcionalidades que a API RestMB oferece bem como outras soluções disponíveis no mercado, porém no quesito custo/benefício para o desenvolvedor as soluções mais robustas para o desenvolvimento de aplicações Restful seriam RestMB e Volley.

Em relação ao uso do RestMB o ponto fraco se comparado ao Volley é que o RestMB não possui suporte a paginação e requisição assíncrona, porém, seu ponto forte é fazer uso de anotações e não depender de APIs de terceiros para realizar o marshall e unmarshall dos dados.

Em relação ao Volley o mesmo possui suporte a paginação e requisição assíncrona, porém não faz uso de anotações e depende da biblioteca GSON para funcionar.

Realizando o comparativo entre as APIs RestMB e Volley é possível notar que ambas são APIs Restful e podem ser utilizadas de maneira eficiente para o desenvolvimento de aplicações Android REST.

5. Conclusão

O objetivo desse trabalho foi de desenvolver uma API que pudesse realizar o processo de requisição a um servidor baseado em Rest, para assim facilitar a integração entre aplicações desenvolvidas para Android e Web Services seguindo os princípios Rest.

O desenvolvimento foi realizado utilizando a linguagem Java, fazendo uso de alguns recursos muito úteis tal como *reflections* para realizar o *marshalling* de dados, além disso, foram criadas classes e interfaces a fim de encapsular os principais métodos do protocolo HTTP.

Em relação à plataforma Android foi possível notar o quanto a mesma pode ser utilizada para se integrar e interagir com outros sistemas e realizar a troca de informações de maneira ágil. Também nota-se o interesse cada vez maior de empresas na utilização de Web Services para deixar seus sistemas prontos para interagir com outras aplicações.

Em relação à implementação para plataforma Android foi possível identificar que a mesma possui uma documentação bem elaborada, completa e de fácil aprendizado, além de vários exemplos encontrados em sites e fórums, muitos deles disponíveis inclusive em português.

Para o teste de um servidor REST remoto foi realizado integração com o Facebook, consumindo alguns recursos visando medir o desempenho da aplicação em realizar o ciclo completo de uma requisição.

Para o teste local foi utilizado o Jersey que é uma biblioteca utilizada para se construir *Web Services Restful* na plataforma Java (JERSEY, 2013), no teste local foram realizados testes utilizando os mesmos critérios usados nos testes do servidor remoto (Facebook).

Trabalhos Futuros

Acredita-se ser uma proposta de trabalho futuro aprimorar o algoritmo para efetuar o *unmarshall* de dados de uma lista de objetos complexos, bem como a melhoria na biblioteca para também realizar o *marshall* e *unmarshall* de informações no formato XML, além disso novos testes deverão ser feitos afim de testar o *unmarshall* de objetos aninhados e verificar seu desempenho.

Sugere-se ainda, como trabalhos futuros, outras formas de autenticação tais como OAuth 2.0 e autenticação Digest e também a melhoria na biblioteca para trabalhar de maneira assíncrona, não necessitando assim que o desenvolvedor tenha que criar uma thread para executar os métodos da API RestMB.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDROID, Disponível em: *http://developer.android.com/tools/index.html* Acessado em Setembro 2013.

BLOCH, Joshua, Effective Java, 2Ed. California: Sun Microsystems, 2008.

BOYD, Ryan, Getting Started with OAuth 2.0, 1Ed. Sebastopol: O'Reilly, 2012.

BURKE, Bill, RESTful Java with JAX-RS, 1Ed. Sebastopol: O'Reilly, 2009.

CHAPPELL, David; JEWELL, Tyler. Java Web Services, 1Ed. Sebastopol: O'Reilly 2002.

CERAMI, Ethan, Web Services Essentials - Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. O'Reilly, 2002.

ENGLANDER, Robert. Java and Soap, 1Ed. Sebastopol: O'Reilly, 2002.

FIELDING, Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, Tese de Doutorado, University of california, Irvine, 2000, Disponível em: *http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.* Acessado em Abril de 2013.

GOURLEY, David; TOTTY, Brian, HTTP: The Definitive Guide, 1Ed. Sebastopol: O'Reilly, 2002.

HARDT, Dick, The OAuth 2.0 Authorization Framework, Disponível em: *http://tools.ietf.org/html/rfc6749.html#section-1.1* Acessado em Outubro 2013.

JERSEY, Disponível em: *https://jersey.java.net/* acessado em Novembro 2013.

KALIN, Martin. Java Web Services Up and Running, 1Ed. Sebastopol: O'Reilly, 2009.

KANKANAMGE, Charitha, Web Services Testing with soapUI, 1Ed. Birmingham: Packt Publishing, 2012.

KIRKPATRICK, Fícus, Disponível em: *http://commondatastorage.googleapis.com/io-2013/presentations/110%20-%20Volley-%20Easy,%20Fast%20Networking%20for%20Android.pdf* acessado em Novembro 2013.

LECHETA, Ricardo R. Google Android: Aprenda a Criar Aplicações Para Dispositivos Móveis com Android SDK. Novatec, 2009.

LEE, Wei Meng, Introdução ao Desenvolvimento de Aplicativos para o Android, 1Ed. CIDADE: Ciência Moderna, 2011.

ORACLE, disponível em *http://docs.oracle.com/javase/tutorial/java/annotations/* acessado em setembro 2013.

RICHARDSON, Leonard; RUBY, Sam, RESTful Web Services. [S.l.]: O'Reilly Media, 2007.
SILVEIRA, Paulo et al. INTRODUÇÃO À ARQUITETURA E DESIGN DE SOFTWARE: Uma visão sobre a plataforma Java. Rio de Janeiro: Elsevier, 2012.

SAUDATE, Alexandre, SOA aplicado Integrando com web services e além, 1Ed. São Paulo: Casa do Código, 2012.

SIERRA, Katty, Certificação Sun para Programador Java 6 - Guia de Estudos - Editora: Alta Books, 2006

SMITH, James, Disponível em: *http://loopj.com/android-async-http/* acessado em 22 novembro 2013.

SQLite, Disponível em: *http://www.sqlite.org/* acessado em Junho 2013.

The RESTful CookBook, Disponível em: *http://restcookbook.com*/ acessado em Maio 2013.

TIDWELL, Doug; SNELL, James; KULCHENKO, Pavel, Programming Web Services with SOAP, 1Ed. Sebastopol: O'Reilly, 2001.

WEBBER, Jim; PARASTATIDIS, Savas; ROBINSON, Ian, REST in Practice, 1Ed. Sebastopol: O'Reilly, 2010.

W3C, Disponível em *http://www.w3.org/TR/soap12-part0/* acessado em 10 novembro 2013.