

Google Data APIs

Interagindo com os serviços Google

Resumo DevMan

De que se trata o artigo:

Este artigo mostra o funcionamento básico do protocolo por trás das APIs de dados Google e apresenta a *Java Client Library*, biblioteca que pode ser usada para acessar as APIs Google a partir de um aplicativo Java.

Para que serve:

As APIs de dados servem para prover uma forma de uma aplicação terceira ser capaz de interagir com os diversos serviços Google. Com elas podemos escrever programas que utilizam e até mesmo modificam dados de serviços como *Google Calendar*, *YouTube*, etc.

Em que situação o tema é útil:

Como sabemos, reusar componentes e serviços de software sempre foi, e continua sendo, uma boa prática. A Google nos provê diversos serviços que já estão bastante maduros e populares. Com isso, as APIs de dados tornam-se extremamente úteis quando precisamos incorporar algum(ns) desses serviços às nossas aplicações.

Google Data APIs:

Os serviços Google estão cada vez mais presentes em nossa rotina. A Google disponibiliza APIs para permitir a interação entre vários de seus serviços e nossas aplicações. As APIs de dados (*GData APIs*) funcionam sobre um protocolo específico, o *GData API Protocol*.

Através deste protocolo podemos obter, inserir, atualizar e remover dados na web. Para cada operação basta enviar uma requisição HTTP utilizando o método correspondente. **GET** para ler, **POST** para criar, **PUT** para atualizar e **DELETE** para remover dados.

Os dados enviados e recebidos são em formato XML, assim como no *Atom*, um dos protocolos no qual o *GData* é baseado. Portanto, para interagir com as APIs de dados a aplicação deve ser capaz de enviar requisições HTTP e processar dados no formato XML.

A Google também disponibiliza bibliotecas clientes, em várias linguagens (incluindo Java), que abstraem o protocolo. Utilizando a *Java Client API*, toda a complexidade do protocolo é traduzida em classes e métodos, como já estamos acostumados a lidar. Além da biblioteca também é disponibilizado um plugin para Eclipse, o qual provê algumas facilidades como templates de projetos. Tendo posse da *Java Client Library* e do plugin para Eclipse, desenvolver aplicações que interagem com os serviços Google pode se tornar uma tarefa simples.

“Organizar toda a informação do mundo e torná-la universalmente acessível e útil”. Essa é a, nada modesta, missão da gigante Google. Como muitos sabem, já faz um bom tempo que o termo “Google” deixou de ser sinônimo apenas do buscador mais usado do planeta e passou a englobar toda uma gama de ferramentas e serviços dos mais variados tipos. Dentre os mais populares estão: *GMail*, *Orkut*, *YouTube*, *Maps*, *Calendar*, *Blogger*, *Picasa Web Albums*, entre outros. E no universo do desenvolvimento de software, a gigante não fica atrás e tam-

Conheça as APIs de dados Google e veja como criar aplicativos Java capazes de interagir com serviços como Google Calendar, Picasa Web Albums e YouTube

PAULO CÉSAR COUTINHO

bém oferece diversos serviços voltados ao desenvolvedor. Temos ferramentas como o *CodeSearch* para procurar por código-fonte público em diferentes linguagens, a *Google Code University* com vários cursos e tutoriais, o repositório de projetos do *Google Code* para hospedar projetos open-source, e, além desses e de outros serviços, também há diversas APIs que permitem ao desenvolvedor criar aplicações capazes de interagir com grande parte dos serviços Google: *Google Maps*, *OpenSocial*, *Google Web Toolkit*, *Android*, etc.

Neste último grupo, temos as APIs de dados Google (*Google Data APIs*), ou simplesmente *GData APIs*. Através do uso destas é possível, para uma aplicação terceira, manipular dados de vários dos serviços Google, como: *Calendar*, *Docs*, *YouTube*, *Picasa Web Albums*, *Blogger*, *Contacts* e muito mais, por meio de web services *REST*¹. Hoje daremos início a uma série de artigos sobre as APIs de dados. Neste primeiro artigo veremos como essas APIs funcionam e como utilizá-las numa aplicação Java.

GData API Protocol

A API de Dados Google define um protocolo padrão, baseado em XML, para leitura e escrita de dados na web. Esse protocolo também é baseado em outros dois já conhecidos: *RSS* e *Atom*, acrescidos do *Atom Publishing Protocol*. Através do protocolo *GData* podemos obter, criar, atualizar e remover dados usando métodos do protocolo HTTP²: GET, POST, PUT e DELETE, respectivamente. Sendo assim, para utili-

zar as APIs de dados, de um modo geral, nossa aplicação deve ser capaz de enviar requisições HTTP e processar dados no formato XML. As operações que precisamos utilizar dependem do que nossa aplicação se propõe a fazer. Se estivermos construindo uma aplicação de agenda telefônica, por exemplo, podemos querer fornecer ao usuário a opção de importar os dados dos seus contatos do *GMail*, e para isto uma operação de leitura (GET) será suficiente. Porém, se quisermos fazer o inverso, ou seja, permitir ao usuário adicionar os contatos da agenda à sua lista de contatos do *GMail*, precisaremos de uma operação para inserção (POST, de acordo com o protocolo *GData*) e assim por diante. Como nosso objetivo aqui é ter uma visão geral da API, veremos como o protocolo *GData* define cada uma das operações básicas.

☑ *Atom e RSS são formatos utilizados para publicação de Web Feeds. Atom Publishing Protocol é um protocolo baseado em HTTP que serve para criar e modificar recursos na web, seguindo o paradigma REST. Veja o artigo "Atom e Atom Publishing Protocol", Edição 57.*

Obtendo dados

Para recuperar dados do servidor basta, dado o endereço de um determinado *feed*³, enviar uma requisição HTTP GET para esse endereço. Vejamos um exemplo⁴:

```
GET /myFeed
```

Se tudo ocorrer bem, receberemos uma resposta semelhante à **Listagem 1**. Vamos analisar a estrutura básica do XML de res-

posta. Temos uma tag raiz chamada *Feed*, a qual possui um conjunto de tags internas. As tags *title*, *updated*, *id*, *author* e *link* são meta-dados com informações sobre o *feed*. Entretanto, temos também uma lista de tags *Entry*, cada qual com suas tags internas. Cada tag *Entry* representa uma entrada⁵ de dados, semelhante a uma linha numa tabela de um banco de dados, e é com elas que interagimos. As operações de ler, inserir, editar e remover são efetuadas sobre essas entradas.

É possível, também, filtrar as entradas que desejamos recuperar. Para isto basta acrescentarmos alguns parâmetros à URL requisitada. Dentre os parâmetros possíveis estão o *q* (*full text query*) que nos permite fazer uma busca textual usando regras semelhantes à busca do Google, e o *category* que nos dá a opção de recuperar as entradas pertencentes a uma ou mais categorias. Vejamos um exemplo de uma requisição utilizando o filtro de texto:

```
GET /myFeed?q=um%20outro
```

Lembre-se que o conteúdo dos parâmetros deve estar *URL-encoded*⁶, de acordo com o protocolo HTTP (no nosso exemplo, "%20" representa um espaço em branco). A **Listagem 2** mostra a resposta para nossa requisição com filtro de texto. Note que, desta vez, apenas as entradas que continham o texto procurado foram retornadas (apenas uma, em nosso caso). Mais detalhes e exemplos sobre o funcionamento dos filtros podem ser encontrados na documentação oficial (ver seção **Links**).

1 REST, REpresentational State Transfer, é um paradigma para construção de Web Services, caracterizado por utilizar os recursos do protocolo HTTP de forma ampla, e também por sua simplicidade e poder. O artigo "Web services: REST versus SOAP", da Edição 54, apresenta os web services REST de forma mais detalhada.

2 Os métodos definidos no protocolo HTTP são: GET, POST, PUT, DELETE, OPTIONS, CONNECT, HEAD e TRACE.

3 Web Feed, ou simplesmente Feed, é o termo usado para representar uma fonte de dados, geralmente atualizados com frequência, num formato bem definido que serve para alimentar um determinado serviço.

4 Os exemplos de requisições e respostas HTTP que utilizaremos no decorrer do artigo exibirão apenas os dados relevantes para o entendimento do protocolo *GData*. Fica implícito que numa requisição ou resposta real, temos mais informações referentes ao protocolo HTTP, como versão do protocolo e outros headers.

5 No decorrer desse artigo utilizaremos o termo "entrada de dados", ou apenas "entrada", para nos referirmos aos dados de um único elemento pertencente a uma coleção. No protocolo *GData* uma entrada está associada diretamente a uma tag *Entry*.

6 Codificado usando o padrão de codificação de URLs (*application/x-www-form-urlencoded*). Java possui uma classe *URLEncoder* que fornece métodos para esse propósito.

Inserindo novos dados

Para inserir uma nova entrada basta enviar um HTTP POST para o endereço do *feed*, colocando a representação XML da nova entrada no corpo⁷ da requisição. A **Listagem 3** mostra um exemplo de uma requisição POST para inserir uma nova entrada e a **Listagem 4** mostra um exemplo do que seria a resposta para essa requisição em caso de sucesso. Como podemos ver na resposta do servidor, além do código HTTP 201 (que representa sucesso na criação), recebemos também um conteúdo XML representando a entrada que acabamos de criar. Porém, perceba que alguns campos, como o ID e o link de edição (`<link rel="edit" ... />`), foram automaticamente adicionados à entrada. Esses dados são úteis para futuras operações sobre a entrada recém-criada, como veremos a seguir.

Atualizando uma entrada existente

Seguindo a mesma filosofia, para atualizar o valor de uma entrada basta enviar uma requisição HTTP PUT⁸ para o endereço de edição da entrada, colocando a representação XML atualizada no corpo. O endereço de edição, como vimos nos exemplos anteriores, é gerado automaticamente pelo servidor sempre que uma entrada é criada e também vem quando recuperamos uma entrada já existente. No nosso último exemplo (**Listagem 4**), veja que o atributo href da tag `<link rel="edit" .../>` contém o endereço de edição da entrada (no nosso exemplo, `http://example.com/myFeed/3/1/`). A **Listagem 5** mostra um exemplo de requisição para atualizar uma entrada. Nesse caso estamos alterando o título e o conteúdo da entrada.

A **Listagem 6** mostra o resultado da requisição apresentado na **Listagem 5**. Veja que, assim como quando criamos uma nova entrada, recebemos, além do código de sucesso (200), a própria entrada atualizada. Aqui devemos atentar principalmente para

⁷ Ao contrário de uma requisição HTTP GET, onde os dados são enviados como parâmetros na própria URL, numa requisição HTTP POST os dados são enviados no corpo, também chamado de payload, da requisição.

⁸ Se estivermos numa rede com o firewall configurado para bloquear alguns métodos HTTP e o método PUT estiver bloqueado, podemos usar um POST no lugar. Basta acrescentarmos o header X-HTTP-Method-Override: PUT para indicar que o método HTTP deve ser sobrescrito.

Listagem 1. Exemplo de resposta de uma requisição GET simples de um feed.

```
200 OK

<?xml version="1.0"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>JM Test GData</title>
  <updated>2008-09-23T08:30:00-03:00</updated>
  <id>http://www.example.com/myFeed</id>
  <author>
    <name>Java Magazine</name>
  </author>
  <link href="/myFeed" rel="self"/>
  <entry>
    <id>http://www.example.com/id/1</id>
    <link rel="edit" href="http://www.example.com/myFeed/1/1/">
    <updated>2008-09-24T08:30:00-03:00</updated>
    <author>
      <name>João</name>
      <email>joao@gmail.com</email>
    </author>
    <title type="text">Minha primeira entrada</title>
    <content type="text">Conteúdo da entrada</content>
  </entry>
  <entry>
    <id>http://www.example.com/id/2</id>
    <link rel="edit" href="http://www.example.com/myFeed/2/1/">
    <updated>2008-09-25T08:30:00-03:00</updated>
    <author>
      <name>Maria</name>
      <email>maria@gmail.com</email>
    </author>
    <title type="text">um outro título</title>
    <content type="text">um outro conteúdo</content>
  </entry>
</feed>
```

Listagem 2. Exemplo de resposta de uma requisição GET utilizando filtros.

```
200 OK

<?xml version="1.0"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title>JM Test GData</title>
  <updated>2008-09-23T08:30:00-03:00</updated>
  <id>http://www.example.com/myFeed</id>
  <author>
    <name>Java Magazine</name>
  </author>
  <link href="/myFeed" rel="self"/>
  <entry>
    <id>http://www.example.com/id/2</id>
    <link rel="edit" href="http://www.example.com/myFeed/2/1/">
    <updated>2008-09-25T08:30:00-03:00</updated>
    <author>
      <name>Maria</name>
      <email>maria@gmail.com</email>
    </author>
    <title type="text">um outro título</title>
    <content type="text">um outro conteúdo</content>
  </entry>
</feed>
```

Listagem 3. Exemplo de requisição POST para inserir uma nova entrada.

```
POST /myFeed

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <author>
    <name>Java Magazine</name>
    <email>jm@gmail.com</email>
  </author>
  <title type="text">Nova Entrada</title>
  <content type="text">Inserindo uma nova entrada...</content>
</entry>
```

Listagem 4. Exemplo de resposta indicando que uma nova entrada foi criada com sucesso.

```
201 CREATED

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.example.com/id/3</id>
  <link rel="edit" href="http://example.com/myFeed/3/1/">
  <updated>2008-09-28T09:15:50-03:00</updated>
  <author>
    <name>Java Magazine</name>
    <email>jm@gmail.com</email>
  </author>
  <title type="text">Nova Entrada</title>
  <content type="text">Inserindo uma nova entrada...</content>
</entry>
```

Listagem 5. Exemplo de requisição PUT para atualiza uma entrada existente.

```
PUT /myFeed/3/1

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.example.com/id/3</id>
  <link rel="edit" href="http://example.com/myFeed/3/1/">
  <updated>2008-09-28T09:15:50-03:00</updated>
  <author>
    <name>Java Magazine</name>
    <email>jm@gmail.com</email>
  </author>
  <title type="text">Nova Entrada Atualizada</title>
  <content type="text">Inserindo uma nova entrada atualizada...</content>
</entry>
```

Listagem 6. Exemplo de resposta indicando que uma nova entrada foi criada com sucesso.

```
200 OK

<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.example.com/id/3</id>
  <link rel="edit" href="http://example.com/myFeed/3/2/">
  <updated>2008-09-28T09:15:50-03:00</updated>
  <author>
    <name>Java Magazine</name>
    <email>jm@gmail.com</email>
  </author>
  <title type="text">Nova Entrada Atualizada</title>
  <content type="text">Inserindo uma nova entrada atualizada...</content>
</entry>
```

o endereço de edição que foi atualizado pelo servidor. Note que o final mudou de /3/1/ para /3/2/ indicando que uma nova versão da entrada foi criada. Os números representam o *ID* e a versão da entrada respectivamente. Então, como atualizamos a entrada, sua versão foi incrementada. Para mais detalhes sobre o mecanismo de versionamento de entradas, consulte a documentação oficial (ver seção **Links**).

Removendo uma entrada

Por fim, para remover uma entrada, basta enviar uma requisição HTTP DELETE⁹ para o endereço de edição da entrada. Se-

⁹ Assim como no caso do PUT, se o firewall não permitir executar requisições HTTP DELETE, podemos enviar um POST acrescentando o header X-HTTP-Method-Override: DELETE

gue um exemplo para remover a entrada que editamos anteriormente:

```
DELETE /myFeed/3/2
```

Note que como estamos removendo uma entrada, não é necessário enviar nada no corpo da requisição. Nesse caso a resposta simplesmente indica o sucesso da operação, como vemos a seguir:

```
200 OK.
```

Formas de Autenticação

Para utilizar a maioria dos serviços Google, precisamos possuir uma conta válida e estar logados nessa conta. Para utilizar as APIs de dados não é diferente. Precisamos informar o login e senha do usuário ao serviço de autenticação e este nos retornará, em caso de sucesso, um *token* de acesso que utilizaremos para nos

identificar nas demais requisições. O *token* é gerado pelo servidor e pode servir para um ou mais serviços, dependendo de como solicitado.

Atualmente as APIs de dados suportam três tipos de autenticação: *ClientLogin*, *AuthSub* e *OAuth*. Para aplicações desktop o método *ClientLogin* é o mais indicado. Já para aplicações web é indicado escolher entre os métodos *AuthSub* e *OAuth*. Vejamos, de forma breve, as principais características de cada um dos meios de autenticação:

- **ClientLogin:** É a forma mais simples. O usuário envia seu login, senha e o ID¹⁰ do(s) serviço(s) com o(s) qual(is) deseja interagir. A principal desvantagem é a segurança, pelo fato do login e senha terem que ser enviados ao servidor pela aplicação cliente, mas isso é amenizado pelo uso de um *token* que é gerado e utilizado nas demais requisições, ou seja, as credenciais do usuário só trafegam na rede uma única vez;

- **AuthSub:** Permite que aplicações web obtenham o *token* de acesso sem precisar manipular as credenciais do usuário. É enviada uma requisição, para o serviço de autenticação *AuthSub*, contendo um parâmetro next que representa a URL para qual o serviço deve retornar. A partir daí o usuário é encaminhado para a página de login da própria Google e, uma vez “logado” é redirecionado para a URL informada no parâmetro next.

- **OAuth:** Padrão aberto para autenticação em aplicações web. Semelhante ao *AuthSub* em vários aspectos. Utiliza certificados digitais para aumentar a segurança.

Serviços Específicos

Como já vimos anteriormente, vários serviços Google suportam o protocolo *GData* e para cada serviço é disponibilizada uma API específica, todas baseadas no protocolo *GData*. Atualmente, as APIs disponíveis são:

- Google Apps APIs;
- Google Base Data API;
- Blogger Data API;
- Google Calendar Data API;
- Google Code Search Data API;
- Google Contacts Data API;

¹⁰ Cada serviço possui um identificador único no formato string (Ex: cl para Calendar, youtube para o YouTube). A lista completa pode ser encontrada na documentação oficial (ver seção **Links**).

Listagem 7. Exemplo de resposta indicando que uma nova entrada foi criada com sucesso.

```
<entry xmlns='http://www.w3.org/2005/Atom' xmlns:gd='http://schemas.google.com/g/2005'>
  <category scheme='http://schemas.google.com/g/2005#kind'
    term='http://schemas.google.com/contact/2008#contact'/>
  <title>Java Magazine</title>
  <content>Meu contato preferido.</content>
  <gd:email rel='http://schemas.google.com/g/2005#work' primary='true'
    address='jm@gmail.com'/>
  <gd:email rel='http://schemas.google.com/g/2005#home'
    address='jm@jm.com.br'/>
  <gd:phoneNumber rel='http://schemas.google.com/g/2005#work' primary='true'>
    (11)3333-4444
  </gd:phoneNumber>
  <gd:phoneNumber rel='http://schemas.google.com/g/2005#home'>
    (11)2222-3333
  </gd:phoneNumber>
  <gd:phoneNumber rel='http://schemas.google.com/g/2005#mobile'>
    (11) 6666-7777
  </gd:phoneNumber>
  <gd:im rel='http://schemas.google.com/g/2005#home'
    protocol='http://schemas.google.com/g/2005#G00GLE_TALK'
    address='jm@gmail.com'/>
  <gd:postalAddress rel='http://schemas.google.com/g/2005#work'
    primary='true'>
    Avenida ABCDE Nº 100, 66555-777
  </gd:postalAddress>
  <gd:postalAddress rel='http://schemas.google.com/g/2005#home'>
    Rua FGHI Nº 200
    São Paulo-SP, 33333-444
  </gd:postalAddress>
  <gd:organization>
    <gd:orgName>Java Magazine</gd:orgName>
    <gd:orgTitle>Presidente</gd:orgTitle>
  </gd:organization>
</entry>
```

Listagem 8. Classe exemplo gerada pelo plugin.

```
import com.google.gdata.client.photos.PicasawebService;
import com.google.gdata.data.photos.AlbumEntry;
import com.google.gdata.data.photos.GphotoEntry;
import com.google.gdata.data.photos.UserFeed;
import com.google.gdata.util.AuthenticationException;
import com.google.gdata.util.ServiceException;

import java.io.IOException;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.List;

public class PicasaWebAlbums {

    public static void main(String[] args) {

        try {
            // Create a new Picasa Web Albums service
            PicasawebService myService = new PicasawebService("My Application");
            myService.setUserCredentials(args[0], args[1]);

            // Get a list of all entries
            URL metafeedUrl = new URL("http://picasaweb.google.com/data/feed/api/
            user/" + args[0] + "?kind=album");
            System.out.println("Getting Picasa Web Albums entries...\n");
            UserFeed resultFeed = myService.getFeed(metafeedUrl, UserFeed.class);
            List<GphotoEntry> entries = resultFeed.getEntries();
            for (int i = 0; i < entries.size(); i++) {
                GphotoEntry entry = entries.get(i);
                System.out.println("\t" + entry.getTitle().getPlainText());
            }
            System.out.println("\nTotal Entries: " + entries.size());
        } catch (AuthenticationException e) {
            e.printStackTrace();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (ServiceException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- Google Finance Portfolio Data API;
- Google Health Data API;
- Google Notebook Data API;
- Google Spreadsheets Data API;
- Picasa Web Albums Data API;
- Google Documents List Data API;
- Webmaster Tools Data API;
- YouTube Data API.

Vimos o protocolo e as APIs dos serviços. Mas como uma entrada de um serviço real se parece? É apenas aquilo que acabamos e ver? Porque nos exemplos que vimos até agora temos sempre as mesmas informações (ID, author, title, content, etc). Essas informações parecem não ser suficientes. E é isso mesmo, essas não são as únicas informações contidas numa entrada real. O Atom, principal protocolo no qual o GData é baseado, é um protocolo extensível que permite o acréscimo de tags específicas. E o que acontece na realidade é que uma entrada de um determinado serviço (Contacts Data API, por exemplo) contém, além das tags padrão do Atom, um conjunto de tags específicas desse serviço, permitindo assim representar todas as características dessa entrada.

A **Listagem 7** mostra um exemplo de uma entrada representando um contato. Note que, além das tags padrão do *Atom*, temos várias *tags* específicas, todas prefixadas pelo namespace *gd*, indicando que são uma extensão do protocolo. Para entender melhor o significado de cada tag é preciso consultar a documentação específica dos serviços.

Java Client Library

Agora que vimos o funcionamento básico do protocolo *GData*, estamos prontos para testar algum dos serviços. Mas como faremos para enviar requisições HTTP? Qual API Java faz isto? E qual biblioteca usaremos para montar e parsear os XMLs? A resposta é, podemos fazer como bem entendermos, usando as APIs e bibliotecas que desejarmos. Porém, como se já não bastasse disponibilizar as APIs de dados, a Google também nos provê um conjunto de bibliotecas clientes escritas em várias linguagens (Java, .NET, PHP, Python, Objective-C e JavaScript), além das bibliotecas de terceiros que podemos encontrar na Internet. Então por que não utilizar a biblioteca Java (*Java Client Library*)? Baixe

a versão mais recente (1.21 atualmente) da *Java Client Library* no repositório do Google Code (ver seção **Links**) e descompacte os *jars* (localizados na pasta *lib*) numa pasta de sua preferência (“C:\gdata”, por exemplo). Para alguns serviços ou funcionalidades será necessário também baixar algumas dependências: *mail.jar* da JavaMail API, *activation.jar* do JavaBeans Activation Framework e *servlet.jar* da Servlet API (Os links para download encontram-se na seção **Links**)¹¹. Baixe os arquivos das dependências e salve-os numa pasta de sua preferência (“C:\gdata\dependencies”, por exemplo).

E para facilitar ainda mais nossas vidas, também nos é disponibilizado um plugin do *GData* para Eclipse. Além de oferecer diversos esqueletos de projeto para cada API de dados, o plugin oferece também a opção de baixar as dependências automaticamente. Para instalá-lo execute o procedimento normal de instalação e atualização de plugins do Eclipse (*Help>Software Updates*, etc) usando o seguinte link: <http://gdata-java-client-eclipse-plugin.googlecode.com/svn/update-site>.

☑ *As instruções para construção do exemplo são para o Eclipse, para mostrar a facilidade extra do plugin. Mas é possível desenvolver o exemplo em qualquer outro IDE Java, bastando incluir manualmente no projeto os jars das APIs utilizadas e escrever todo o código do zero, sem auxílio dos esqueletos gerados pelo plugin.*

Aplicação Exemplo

Com os arquivos da biblioteca Java baixados e o *plugin* do Eclipse instalado, vamos fazer um teste básico utilizando a API do *Picasa Web Albums*. No Eclipse, escolha a opção *File>New>Project*, selecione *Google Data>Google Data Project*, escreva “*GData Picasa Test*” no nome do projeto, escolha o template *Picasa Web Albums*. Por fim, selecione as pastas onde estão os *jars* da *Java Client Library* e também a pasta onde se encontram as dependências. O projeto será criado juntamente com uma classe, gerada

¹¹ Estas três APIs fazem parte da plataforma Java EE, portanto, os *jars* mencionados não são necessários para aplicações que rodem em containers Java EE. O JavaBeans Activation Framework também é incluído no Java SE 6.

automaticamente pelo plugin, contendo um exemplo de código que recupera a lista de álbuns do usuário. A **Listagem 8** mostra o código da classe gerada pelo plugin. Note que o login e senha devem ser passados por linha de comando, portanto configure os argumentos no Eclipse ou, se preferir, substitua diretamente as variáveis `args[0]` e `args[1]` pelas strings correspondentes ao seu login e senha, respectivamente.

Analisando o código do exemplo, temos primeiramente a criação do objeto representando o serviço que desejamos utilizar, *PicasawebService* no nosso caso. Em seguida usamos o método `setUserCredentials()` para informar nosso login e senha, necessários para ter acesso ao serviço. Montamos a URL para acessar a lista de álbuns do usuário de acordo com o padrão definido pela API. No caso do *Picasa Web Albums* a URL deve ter o seguinte formato: <http://picasaweb.google.com/data/feed/api/user/<username>?kind=album>. Basta substituir “<username>” pelo login do usuário. Depois disso chamamos o método `getFeeds()` do serviço informando a URL e o tipo de *feed* que desejamos (com.google.gdata.data.photos.UserFeed, em nosso exemplo). Em seguida iteramos sobre as entradas do nosso *feed*, recuperadas através de uma chamada ao método `getEntries()` do *UserFeed*, e exibimos seu título.

Execute o programa e se tudo ocorrer bem, uma lista com os títulos de seus álbuns será exibida no console. Com isso concluímos nossa primeira experiência com o protocolo *GData*.

Conclusões

Neste artigo vimos o funcionamento básico do protocolo *GData*, usado pelas APIs de dados Google. Como podemos perceber, é um protocolo bastante simples de entender. Utilizamos os métodos HTTP para indicar as operações que desejamos realizar e XML para representar os dados com/sobre os quais as operações serão executadas. Também demos uma olhada na *Java Client Library* e no plugin para Eclipse. Por fim, testamos, com um exemplo prático, o funcionamento de um serviço real. Com isso, vimos que usar as APIs de dados pode se tornar uma tarefa fácil, quando utilizamos a *Java Client Library*. Nos próximos artigos veremos os serviços específicos em maiores detalhes.

<http://code.google.com/apis/gdata/>
Site oficial das Google Data APIs.

<http://code.google.com/apis/gdata/reference.html#Queries>

Mostra as opções de filtro disponíveis e como utilizá-las.

<http://code.google.com/apis/gdata/reference.html#Optimistic-concurrency>

Mostra o mecanismo de versionamento de entradas.

http://code.google.com/apis/base/faq_gdata.html#clientlogin

Lista com os IDs de cada serviço para usar no ClientLogin.

<http://code.google.com/p/gdata-java-client/downloads/list>

Site para download da Java Client Library.

<http://java.sun.com/products/javamail/downloads/index.html>

Site para download da JavaMail API (arquivo mail.jar).

<http://java.sun.com/products/javabeans/jaf/downloads/index.html>

Site para download do JavaBeansActivationFramework. (arquivo activation.jar).

<http://tomcat.apache.org/download-60.cgi>

Site para download do Tomcat (arquivo *servlet-api.jar*).



Paulo César M. N. A. Coutinho

pcmnac@gmail.com

é graduado em Tecnologia em Sistemas de Informação pelo Centro Federal de Educação Tecnológica de

Pernambuco (CEFETPE). Trabalha como Engenheiro de Sistemas no Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R) com desenvolvimento móvel em C/C++ e Java. Também vem trabalhando com Java para web e desenvolvimento de componentes open-source em projetos pessoais. Possui as certificações SCJP 5 e SCWCD 1.4.

Dê seu feedback sobre esta edição!

A Java Magazine tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/javamagazine/feedback

