Android

Utilizando Web Services no Google Android

Ramon Ribeiro Rabello
ramon.rabello@gmail.com
É graduado em Ciência da Computação pela Universidade da Amazônia (UNAMA). Trabalha com a tecnologia Java desde 2005. Possui experiência nas três especificações: JSE, JEE e JME. Trabalhou com desenvolvimento de aplicações móveis comerciais (M-Commerce / M-Payment). É um dos evangelistas de Android no Brasil, promovendo o uso da plataforma em cursos e palestras. Atualmente é mestrando da Universidade Federal de Pernambuco (UFPE) na área de Engenharia de Software, mais especificamente em Model-Driven Architecture (Arquitetura Dirigida a Modelos) e um dos membros do projeto ORCAS (*www.orcas.eu*).

De que se trata o artigo:

Utilização de um Web Service para consulta de CEPs na plataforma Android e integração com Google Maps para dar mais expressividade gráfica a aplicações mash-ups.

Para que serve:

Por meio da aplicação exemplo utilizada, demonstrar passo a passo como consumir Web Services em Android usando bibliotecas adicionais, como é o caso do KSOAP2, aproveitando a vantagem de se utilizar mapas para mostrar a localização.

Em que situação o tema é útil:

Consultar as informações referentes ao CEP informado (por exemplo, endereço, cidade, estado, etc), tendo a facilidade de visualizar esta localização no mapa por meio da API de Mapas em Android e do componente ItemizedOverlay, para construções de novas camadas para sobrepor MapViews.

Há muito que os Web Services vêm sendo utilizados em aplicações comerciais para resolver – acima de tudo – o problema da “Babel” dos sistemas, permitindo que vários sistemas escritos em linguagens distintas se comuniquem por meio de serviços (métodos) que são expostos para que outros módulos ou sistemas possam acessá-los. Para isso, a tecnologia XML (na verdade, uma variação mais avançada e robusta chamada XML Schema) associada ao protocolo SOAP são os

protagonistas deste cenário e utilizada como canal comum de comunicação e que constituem a arquitetura SOA (Service Oriented Architecture – Arquitetura Orientada à Serviço).

Provado o sucesso destes no universo das tecnologias do lado do servidor (como PHP, JSP, JSF, etc), tão logo a tecnologia tornou-se disponível para o mundo ubíquo por meio de implementação do protocolo SOAP para as várias linguagens de programação móvel (dentre elas, a plataforma Android).

Porém, a tecnologia teve que ser "enxugada" devido a características intrínsecas aos Web Services. Por exemplo, permitir trabalhar com estruturas de dados mais complexas, manipular várias linhas retornadas no XML de resposta a um serviço, carregar uma hierarquia de árvore era quase inviável devido às capacidades restritas dos dispositivos móveis referentes à capacidade de processamento e espaço de memória reduzido.

Este artigo demonstrará passo a passo como se acessar um Web Service utilizando a biblioteca KSOAP2 em Android. Para isso, será criada uma aplicação que acessa um Web Service público disponível em *www.maniezo.com.br* que será utilizado para consultar o CEP de determinada localidade e retorna informações adicionais, mostrando também a localização correspondente deste CEP no mapa usando o Google "Android" Maps.

Na época da escrita deste artigo, foram utilizados a IDE Eclipse 3.4 (codinome Ganymede, disponível em *www.eclipse.org/ganymede/*) juntamente com o plug-in para desenvolvimento de Android no Eclipse chamado ADT (Android Development Tools) e a versão 1.0 Release 2 (r2) do SDK para Windows (*http://code.google.com/android/download.html*), pode ser que atualmente exista. Atenção: muito cuidado para baixar a versão do SDK compatível com a versão do plug-in ADT, caso contrário poderá gerar erros indesejáveis em sua aplicação!

Nota DevMan: Compatibilidade entre versões do SDK

Um dos maiores problemas que estamos enfrentamos durante o desenvolvimento de uma aplicação em Android talvez seja este "bombardeio" de versões de releases do SDK que a plataforma tem sofrido durante este período de amadurecimento do Google Android, o que implicou severamente na parcial incompatibilidade entre as aplicações desenvolvidas em versões diferentes. Elas eram desenvolvidas em uma versão e quase sempre sofriam ajustes (renomeações de APIs, remoção de classes, etc) para que fosse possível rodá-la na versão mais recente.

Além disso, existe uma correspondência entre a versão de um SDK e a versão do plugin ADT. Por exemplo, o SDK 1.0 Release 2 (versão mais atual disponível durante a escrita deste artigo) requer a versão 0.8.0 do plugin Eclipse ADT para que o desenvolvimento de aplicações funcione corretamente.

Sendo assim, tenha em mente esta característica quando estiver desenvolvendo, adaptando ou migrando aplicações em Android que foram construídas utilizando versões de SDKs diferentes. Para baixar a última versão do SDK acesse *http://code.google.com/intl/pt-BR/android/download.html*. Para mais informações de como instalar/atualizar o plugin Eclipse ADT acesse *http://code.google.com/intl/pt-BR/android/intro/installing.html#installingplugin*.

Construindo a GUI da aplicação

Nossa aplicação será utilizada para buscar informações de determinada localidade de acordo om o CEP informado, sendo que essa busca será feita por meio do acesso a um Web Service. Ela será muito útil para sabermos mais informações das mais variadas localidades do Brasil (por enquanto, ficamos apenas com a implementação via emulador e esperamos ansiosamente para que o G1 – atualmente algum dispositivo com a plataforma Android chegue logo no Brasil!). A aplicação ainda mostra a localidade específica deste CEP no mapa de acordo com sua latitude/longitude.

Como ponto de partida para qualquer desenvolvimento de uma aplicação, criaremos um novo projeto Android. A **Figura 1** mostra as configurações (nome do projeto, da Activity, etc) que foram utilizadas para este projeto.

A aplicação é bem simples e é composta – essencialmente – de TextViews para visualizar textos, um EditText para entrada de texto (no nosso caso o CEP), um ImageButton para disparar eventos de cliques; e um MapView, que representa o mapa renderizado pelo Google Maps que exibirá a localização exata deste CEP.

LinearLayout é o layout padrão utilizado por todos os widgets. Dentro do primeiro layout percebemos a utilização de um ViewFlipper, que é uma das novidades na plataforma que une o conceito de animação (um outro tipo de recurso em Android) para ser aplicada às Views. Este componente será utilizado para animar os TextViews que informam como se utilizar nosso buscador, como numa espécie de letreiro digital que ficará sendo exibido continuamente no topo e no centro da tela do dispositivo, com o efeito de aparecimento (fade in, referente ao atributo android:fromAlpha=”0.0” que significa totalmente opaco) e desaparecimento (fade out, referente ao atributo android:toAlpha=”1.0” que significa totalmente transparente), com duração da animação de 100 milisegundos (android:duration=”100”), utilizando o efeito de aceleração (atributo android:interpolator), referenciado por meio de @anim:accelerate_interpolator em fade.xml (**Listagem 1**), que está em res/anim, diretório padrão para Android carregar recursos de animação. Ainda na tag <ViewFlipper>, encontramos o atributo android:flipInterval=”2000” informando que cada View terá a duração de 2 segundos (2000 milisegundos) para ser exibido na tela. O atributo android:padding diz respeito ao espaçamento entre o filho e o pai que, nesse caso, vai ser 10 pixels igualmente em todos os lados: cima, baixo, esquerda e direita.

Seguindo, temos um outro LinearLayout – na direção horizontal – que contém um TextView com o texto “CEP:” (referenciado por @string/cep_desc). o EditText para entrada do CEP (@+id/etCEP) e um ImageButton (@+id/imgBtnPesquisar) que possui as mesmas funcionalidades do Button, exceto que em vez de um texto, uma imagem é mostrada (no nosso caso uma imagem de uma lupa por meio da referência @drawable/search).

Logo abaixo, temos também um LinearLayout que contém os TextViews que mostrarão as informações da localidade do CEP informado. Essas informações serão preenchidas no momento que a busca for realizada com sucesso. Essas informações são respectivamente: o Estado (@+id/tvEstado), a Cidade (@+id/tvCidade), o Bairro (@+id/tvBairro), o tipo do logradouro (@+id/tvTipoLogradouro), o Logradouro (@+id/tvLogradouro) e Complemento (@+id/tvComplemento). A **Listagem 2** mostra o arquivo main.xml que representa a GUI da aplicação.

Logo abaixo temos um RelativeLayout que contém um MapView que exibirá a localização exata (latitude/longitude) de acordo com o CEP fornecido; e um LinearLayout que será utilizado para adicionarmos um controle de zoom em nosso mapa. Um detalhe muito importante: a partir da versão 0.9 do SDK da plataforma, para que possamos utilizar MapViews é obrigatório assinarmos, por meio de uma chave privada, a aplicação e adicionarmos umas tags específicas no AndroidManifest.xml, caso contrário sua aplicação irá abortar quando estiver prestes a ser executada (mais detalhes sobre este processo obrigatório, ver **Nota 1**).

Listagem 1. fade.xml

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
       android:interpolator="@android:anim/accelerate_interpolator"
       android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="100" />
```

Listagem 2. main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:id="@+id/rootLayout">

        <ViewFlipper android:id="@+id/flipper"
                android:layout_width="fill_parent" android:layout_height="wrap_content"
                android:flipInterval="2000" android:padding="10px">

                <TextView android:layout_width="fill_parent"
```

```
                android:layout_height="wrap_content" android:gravity="center_horizontal"
                android:textSize="16px" android:text="Entre com o CEP," />

        <TextView android:layout_width="fill_parent"
                android:layout_height="wrap_content" android:gravity="center_horizontal"
                android:textSize="16px" android:text="clique na lupa ao lado ou" />

        <TextView android:layout_width="fill_parent"
                android:layout_height="wrap_content" android:gravity="center_horizontal"
                android:textSize="16px" android:text="pressione ENTER" />
    </ViewFlipper>

    <LinearLayout android:orientation="horizontal"
        android:layout_width="wrap_content" android:layout_height="wrap_content">

        <TextView android:id="@+id/tvCepDesc" android:layout_width="wrap_content"
                android:layout_height="wrap_content" android:text="@string/cep_desc" />

        <EditText android:id="@+id/etCEP" android:layout_width="240px"
                android:hint="Digite o CEP" android:layout_height="wrap_content" />

        <ImageButton android:id="@+id/imgBtnPesquisar"
                android:layout_width="wrap_content" android:layout_height="wrap_content"
                android:src="@drawable/search" />

    </LinearLayout>

    <LinearLayout android:orientation="vertical"
        android:layout_width="fill_parent" android:layout_height="wrap_content">

        <TextView android:id="@+id/tvEstado" android:layout_width="wrap_content"
                android:textSize="12px" android:layout_height="wrap_content"
                android:text="@string/estado" />

        <TextView android:id="@+id/tvCidade" android:layout_width="wrap_content"
                android:textSize="12px" android:layout_height="wrap_content"
                android:text="@string/cidade" />

        <TextView android:id="@+id/tvBairro" android:layout_width="wrap_content"
                android:textSize="12px" android:layout_height="wrap_content"
                android:text="@string/bairro" />

        <TextView android:id="@+id/tvTipoLogradouro"
                android:layout_width="wrap_content" android:textSize="12px"
                android:layout_height="wrap_content" android:text="@string/tipo_logradouro" />

        <TextView android:id="@+id/tvLogradouro"
                android:layout_width="wrap_content" android:textSize="12px"
                android:layout_height="wrap_content" android:text="@string/logradouro" />

        <TextView android:id="@+id/tvComplemento"
                android:layout_width="wrap_content" android:textSize="12px"
                android:layout_height="wrap_content" android:text="@string/complemento" />

    </LinearLayout>

    <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent" android:layout_height="fill_parent">

        <com.google.android.maps.MapView
                android:id="@+id/map" android:layout_width="fill_parent"
                android:layout_height="fill_parent" android:apiKey="{api_key_gerada}"
                android:clickable="true" />

        <LinearLayout android:id="@+id/map_zoom"
                android:layout_width="wrap_content" android:layout_height="wrap_content"
                android:layout_alignParentBottom="true"
                android:layout_centerHorizontal="true" />
    </RelativeLayout>
</LinearLayout>
```

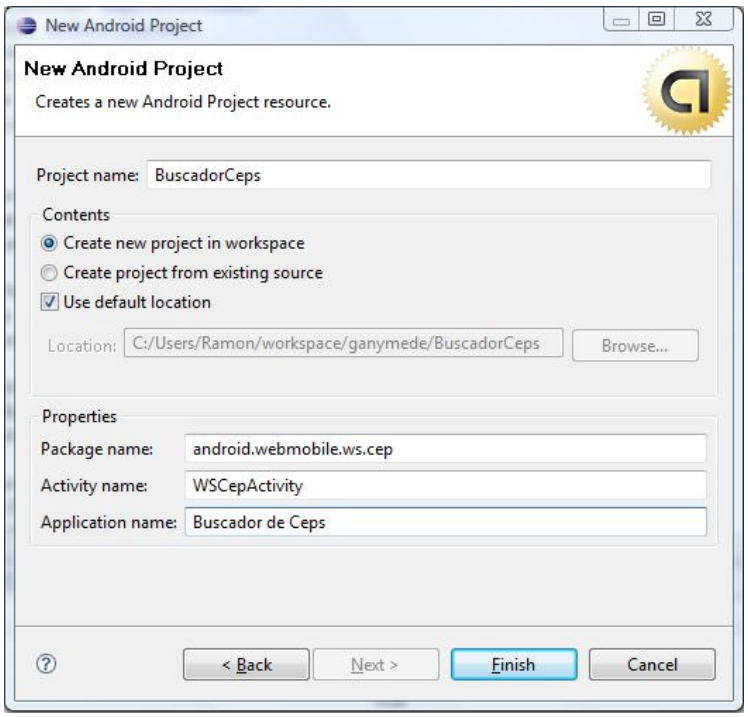

Figura 1. Configurações iniciais do projeto.

Incrementando a Activity

Aqui vai um lembrete essencial: pelo fato de mostrarmos um mapa em nossa Activity, devemos em vez de estender simplesmente Activity (criado por padrão), alteramos para WSCepActivity herdar de MapActivity, haja vista que este tipo de Activity já possui funcionalidades (threads) implementadas para trabalhar com mapas, implementar um cache de níveis dos mapas, acessar arquivos em disco, etc.

Começando o desenvolvimento de nossa MapActivity, temos inicialmente o método onCreate() que é chamado no momento da criação de qualquer Activity. Dentro dele, chamamos o método onCreate() da superclasse e o método setContentView() carrega toda a GUI definida em main.xml por meio da referência R.layout.main. Depois, recuperamos em forma de classes os widgets definidos em main.xml por meio do método findViewById() passando o inteiro que cada um representa na classe de recursos R.java.

Depois, utilizamos a classe utilitária AnimationUtils e o método loadAnimation() para carregar o efeitos já pré-definidos em Android, que no nosso caso será de aparecimento (android.R.anim.fade_in) e desaparecimento (disponível em android.R.anim.fade_out) dos TextViews que serão manipulados por <ViewFlipper> (variável flipper). Configuramos o efeito de aparecimento em setInAnimation() e desaparecimento em setOutAnimation(), passando respectivamente os objetos Animations (variáveis in e out), retornados pelo método loadAnimation(). Por fim, chamamos flipper.startFlipping() para iniciarmos a animação.

Agora, para adicionarmos os controles de zoom no MapView, obtemos o LinearLayout por meio da referência R.id.map_zoom passado para findViewById(). Chamamos o método addView() e passamos como parâmetro um objeto ZoomControls, retornado por mapView.getZoomControls(), e configuramos a altura e largura deste objeto como LayoutParams.WRAP_CONTENT, informando que ambas devem ser ajustadas de acordo com o conteúdo do componente.

Queremos que a busca do CEP ocorra de acordo com dois eventos: ou clicando no ImageButton ou pressionando a tecla “ENTER”. Para isso, devemos permitir que tanto ImageButton quanto EditText sejam registrados para escutar eventos de clique – por meio da interface

View.OnClickListener e método onClick() - ou de pressionamento de teclas, implementando View.OnKeyListener e o método onKey(). Finalmente, registramos o ImageButton (variável imgBtnPesquisar) para escutar eventos de clique por meio do método setOnClickLister() e para o EditText (variável etCEP), chamamos o método setOnKeyListener() para capturar eventos de pressionamento de teclas. Ambos os métodos recebem como parâmetro a referência "this" informando que é a nossa própria Activity que implementará as interfaces de tratamento de eventos.

Para dar mais riqueza à aplicação, adicionamos um menu que permite mudar o modalidade do mapa (tráfego ou satélite). Para isso, sobrescrevemos o método onCreateOptionsMenu() e onOptionsItemSelected(). O primeiro método inicializa o menu de opções da Activity. Nele, criamos um SubMenu que é retornado quando chamamos addSubMenu(), que recebe por parâmetro quatro objetos que representam, respectivamente, o id do grupo, id do item, a ordem desse item no menu e o titulo. Logo em seguida, chamamos setIcon() para alterarmos o ícone (referenciado por android.R.drawable.ic_menu_mapmode) desse SubMenu. Logo abaixo, adicionamos neste SubMenu os MenuItems que representarão os modos de visualização do mapa: tráfego e satélite. Para capturarmos os cliques neste itens de menu e alterarmos o modo do mapa, codificamos a lógica dentro de onOptionsItemSelected() que recebe como parâmetro o item (MenuItem) selecionado.

Finalmente nessa parte inicial, temos o método sobrescrito isRouteDisplayed() que retorna um booleano para informar se algum tipo de rota está sendo visualizada. A **Listagem 3** a nossa MapActivity com as devidas alterações. A **Listagem 4** exibe o arquivo AndroidManifest.xml, já configurado com as tag <uses-library> que faz referência a biblioteca "com.google.android.maps" e as de permissão <uses-permission> necessárias para acesso à internet (android.permission.INTERNET) e para ter acesso a outros tipos de protocolos mais granulados, como Wi-Fi, CellID, etc (android.permission.ACCESS_COARSE_LOCATION), sendo que esta segundo é requerida no momento de obtenção da latitude/longitude da localidade. A **Listagem 5** mostra o arquivo strings.xml que representa as Strings estáticas utilizadas na aplicação; e a **Figura 2** exibe nossa aplicação em seu estágio inicial**.**

Listagem 3. WSCepActivty.java

```
/* package & imports */

public class WSCepActivity extends MapActivity implements View.OnClickListener, View.OnKeyListener {

  /* declaração de variáveis*/

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    flipper = (ViewFlipper) findViewById(R.id.flipper);
    Animation in = AnimationUtils.loadAnimation(this, android.R.anim.fade_in);
    Animation out = AnimationUtils.loadAnimation(this, android.R.anim.fade_out);

    flipper.setInAnimation(in);
    flipper.setOutAnimation(out);
    flipper.startFlipping();

    /* inicializando widgets */
    etCEP = (EditText) findViewById(R.id.etCEP);
    tvEstado = (TextView) findViewById(R.id.tvEstado);
    tvCidade = (TextView) findViewById(R.id.tvCidade);
    tvBairro = (TextView) findViewById(R.id.tvBairro);
    tvTipoLograd = (TextView) findViewById(R.id.tvTipoLogradouro);
    tvLograd = (TextView) findViewById(R.id.tvLogradouro);
    tvComplemento = (TextView) findViewById(R.id.tvComplemento);
    imgBtnPesquisar = (ImageButton) findViewById(R.id.imgBtnPesquisar);
    mapView = (MapView) findViewById(R.id.map);

    LinearLayout zoomLayout = (LinearLayout) findViewById(R.id.map_zoom);
    zoomLayout.addView(mapView.getZoomControls(), new ViewGroup.LayoutParams(
        LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT));
```

```
    imgBtnPesquisar.setOnClickListener(this);
    etCEP.setOnKeyListener(this);
  }

  @Override
  protected boolean isRouteDisplayed() {
    return false;
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    SubMenu subMenu = menu.addSubMenu(0,1,2,"Modo de
Mapa").setIcon(android.R.drawable.ic_menu_mapmode);
    subMenu.add(0,2,1,"Tráfego");
    subMenu.add(0,3,2,"Satélite");
    return true;
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {

    switch (item.getItemId()){
      case 2:
        mapView.setTraffic(true);
        mapView.setSatellite(false);
        break;
      case 3:
        mapView.setTraffic(false);
        mapView.setSatellite(true);
        break;
    }
    return true;
  }
}
```

Listagem 4. AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
       package="android.webmobile.ws.cep" android:versionCode="1"
       android:versionName="1.0.0">

       <uses-permission android:name="android.permission.INTERNET" />
       <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

       <application android:icon="@drawable/icon" android:label="@string/app_name">

              <uses-library android:name="com.google.android.maps" />

              <activity android:name="WSCepActivity" android:label="@string/app_name">
                     <intent-filter>
                            <action android:name="android.intent.action.MAIN" />
                            <category android:name="android.intent.category.LAUNCHER" />
                     </intent-filter>
              </activity>
       </application>
</manifest>
```

Listagem 5. strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Buscador de CEPs</string>
    <string name="cep_desc">CEP:</string>
    <string name="estado">Estado:</string>
    <string name="cidade">Cidade:</string>
    <string name="bairro">Bairro:</string>
    <string name="tipo_logradouro">Tipo Logradouro:</string>
    <string name="logradouro">Logradouro:</string>
    <string name="complemento">Complemento:</string>
</resources>
```

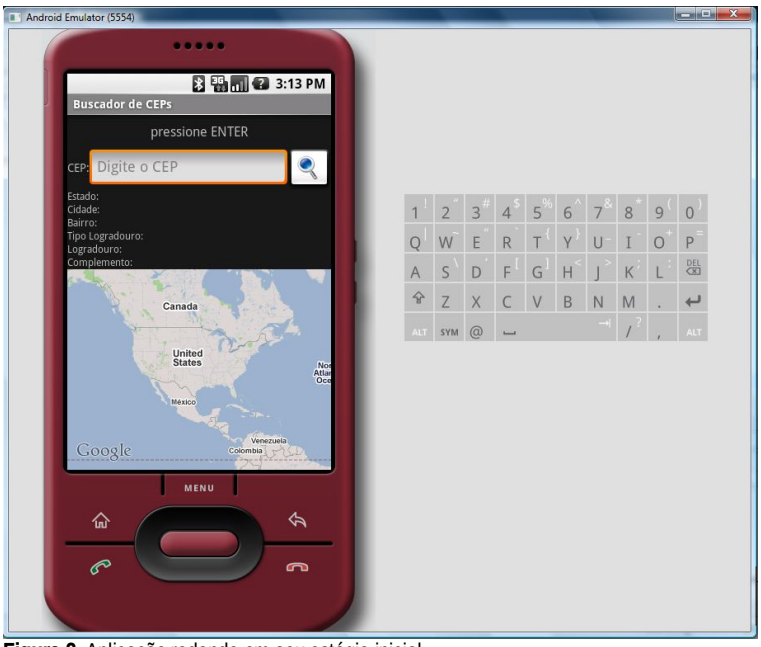

Figura 2. Aplicação rodando em seu estágio inicial.

Nota 2: Obtendo uma API Key para ativar o uso de MapViews

A partir da versão 0.9 do SDK, os MapViews e as aplicações, para poderem rodar nos dispositivos em Android, obrigatoriamente devem ser assinados. Isto torna-se necessário, pois eles permitem acesso aos dados do Google Maps. Sendo assim, é preciso se registrar ao serviço do Google Maps e aceitar aos Termos de Serviços para que seu MapView possa obter qualquer tipo de dados retornados pelo Google Maps. E isso se aplica tanto durante o desenvolvimento das aplicações no emulador quanto para uma aplicação que irá rodar normalmente em um dispositivo que rode Android (como o HTC G1 T-Mobile, atualmente o único dispositivo móvel disponível comercialmente rodando a plataforma do Google).

Registrar um MapView a fim de se obter uma chave privada válida – comumente chamada de API Key – é simples, de graça e pode ser seguido basicamente em dois passos:

1. Registrar o fingerprint (impressão digital) MD5 do certificado que você utilizará para assinar a sua aplicação. Depois, deixe que o próprio serviço de registro do Google Maps forneça a chave gerada para ser utilizada nas aplicações que foram assinadas com o certificado que você utilizou.
2. Adicionar em cada MapView utilizado, uma referência para a API Key gerada (via XML ou via código), desde que esta chave tenha sido a que foi gerada pelo fingerprint que você utilizou durante o processo de registro.

É possível que você crie seu próprio certificado (da mesma maneira que utilizávamos para assinatura de Applets nos tempos áureos de Java...) utilizando a ferramenta "keytool" que vem integrada em qualquer SDK de Java. Em termos de desenvolvimento, Android já disponibiliza um certificado – um arquivo chamado debug.keystore (conhecido também como Debug Certificate) - que fica armazenado num diretório específico que varia de plataforma para plataforma. A **Tabela 1** mostra o diretório em que o arquivo debug.keystore está armazenado de acordo com a plataforma. Porém, para deploy das aplicações em dispositivos, é necessário que outro certificado seja utilizado. Mas se você estiver usando o Eclipse/ADT para desenvolvimento, o local deste arquivo pode ser obtido indo em Windows > Preferences > Android > Build.

Plataforma	Diretório
Windows Vista	C:\Users\<user>\AppData\Local\Android\debug.keystore
Windows XP	C:\Documents and Settings\<user>\Local Settings\Application Data\Android\debug.keystore
OS X e Linux	~/.android/debug.keystore

Tabela 1. Correspondência entre a plataforma e o diretório de debug.keystore.

Como estamos desenvolvendo no emulador e não iremos fazer o deploy de uma aplicação em nenhum dispositivo (por enquanto) real, utilizaremos o próprio Debug Certificate para gerarmos a API Key para utilizarmos no nosso MapView.

Sendo assim, para obtermos um fingerprint MD5 para nosso certificado, executamos o seguinte comando em um prompt de comando:

```
keytool -list -alias androiddebugkey –keystore
<caminho_para_debug_keystore>.keystore –storepass android –keypass android
```

Depois disso, copiamos o fingerprint MD5 e acessamos o site de registro da Android Maps API, que está disponível em *http://code.google.com/intl/pt-BR/android/maps-api-signup.html.* Leia o termo de serviço, aceite e cole no único campo de texto o fingerprint MD5 obtido. Depois disso, você será redirecionado para outro site que mostra a chave que você deve utilizar nos seus MapViews. Simples, não? Agora você escolhe: adicionar a chave via XML ou via código, passando para o construtor de MapView a chave gerada. Para a nossa aplicação, iremos adicionar a chave – por meio do atributo android:apiKey - via XML. O site que contém a chave gerada já disponibiliza um exemplo de como seria a utilização de um MapView via XML. Se você preferir, pode copiar e colar o exemplo. A Listagem 2 já utiliza o MapView via XML, somente deve ser alterado o valor de android:apiKey para a chave que foi gerada anteriormente. Para mais detalhes de como gerar o seu prório certificado, assinar a sua aplicação e outras informações necessárias para o deploy de aplicações em Android, acessar *http://code.google.com/intl/pt-BR/android/devel/sign-publish.html*.

Entendendo o funcionamento do Web Service de CEP

Antes de codificarmos a lógica de consumo do WS de busca de CEPs (também possui a função de cálculo de frete, mas como o próprio site informa, só existe implementação para busca de CEPs), devemos realizar um cadastro no site *www.maniezo.com.br* (na parte direita da tela numa área com o título “Webservice” e clicando em “leia mais”), pois para o acesso será necessário informar o nome de usuário e a senha como parâmetros para o WS. Feito o cadastro, na mesma tela, será apresentada a URL de acesso WSDL - *http://www.maniezo.com.br/webservice/soap-server.php?wsdl* - que contém os serviços e a descrição completa do formato das mensagens de envio e resposta e um pequeno exemplo da utilização do acesso a este em PHP.

O parâmetro de consulta deste Web Service possui um formato que deve ser seguido. Para isso, deve-se passar como parâmetro o seguinte esquema: “CEP#login#senha#”. Estes campos, inclusive o caractere especial “#”, devem ser passados exatamente nesta ordem. Se tudo ocorrer em perfeita ordem, o resultado obtido deve ser uma String no formato: TipoLogradouro#Logradouro#Complemento#Bairro#Cidade#Estado#. Como uma forma de um guia de consulta, a **Tabela 1** mostra os dados retornados pelo Web Service em forma de um array (indicado por resultado[i]), onde cada informação está contida em uma posição.

Posição no Array	Descrição
resultado[0]	Tipo do Logradouro (Ex: Rua, Avenida)
resultado[1]	Logradouro (Endereço)
resultado[2]	Complemento

resultado[3]	Bairro
resultado[4]	Cidade
resultado[5]	Estado no formato de siglas (Ex: PA)

Tabela 1. Guia de consulta do formato de retorno do Web Service.

Configurando KSOAP2 em Android

Aqui iremos configurar o nosso projeto para utilizar a biblioteca KSOAP2, muito conhecida em Java ME para consumo de aplicações. Porém, antes vale ressaltar uma característica relevante da plataforma: como Android já vem integrado no mínimo com o Tiger (codinome do Java 5.0), conseguimos facilmente rodar bibliotecas que já funcionam para Java SE, como é o caso do KSOAP2.

Acessando o site da biblioteca – *http://sourceforge.net/projects/ksoap2/*, percebemos que a plataforma possui três tipos de releases: ksoap2-j2me-core-<versão>.jar, ksoap2-j2me-extras-<versão>.jar, ksoap2-j2me-nodeps-<versão>.jar e ksoap2-j2se-full-<versão>.jar. As primeiras versão, como podemos observar, é para Java ME, o que não irá funcionar em Android devido às determinadas APIs de Java ME estarem ausentes, o que gerará um RuntimeException no momento que o Web Service estiver sendo consumido. Sendo assim, iremos escolher versão completa para Java SE que funciona perfeitamente em Android (na época da escrita deste artigo, a versão disponível para KSOAP2 era a 2.1.2).

Depois de termos baixado a biblioteca, agora iremos integrá-la em nossa aplicação para que possamos acessar as suas APIs de acesso à Web Services. Para isto basta a adicionarmos no Build Path do projeto. Seguiremos os seguintes passos no Eclipse (para desenvolvedores mais avançados, favor seguir para a próxima seção):

- Clicar com o botão direito em cima do nome do projeto
- Apontar para "Build Path > Configure Build Path…"
- Clicar no botão "Add JARs…" se a biblioteca estiver no classpath do projeto ou "Add External JARs...", se a sua aplicação estiver em um outro local no disco
- Selecionar o JAR e clicar em "Abrir"

Pronto. Agora já podemos ter acesso a toda API do KSOAP2.

Consumindo Web Services com KSOAP2

Agora, criaremos em nosso projeto uma outra classe - WSConnection,Java (**Listagem 6**) - que será responsável por conter a regra de negócio necessária para o consumo do Web Service, bem como realizar a pesquisa de CEP, tratar o resultado obtido, etc.

Começamos declarando três variáveis públicas finais estáticas que representam, respectivamente, a URL do WebService (URL) o nome da operação responsável por buscar o CEP (OPERATION), o namespace do WS (NAMESPACE), o nome de usuário (USERNAME) e a senha (PASSWORD).

Seguindo, declaramos o método estático pesquisarCEP(), passando como parâmetro uma String que representará o CEP informado pelo usuário. Depois, começamos a utilizar a primeira das principais classes da bliblioteca – SoapObject (variável request) - que representará o encapsulamento da requisição que será feita ao Web Service, ou seja, o que vai "dentro" do envelope. Ela recebe como parâmetro uma String representando o namespace referente ao WS e o nome da operação, que no nosso caso é "traz_cep".

Em seguida, criamos um objeto SoapSerializationEnvelope (variável envelope) que é a abstração de um envelope SOAP e passamos como parâmetro a versão do protocolo SOAP que iremos utilizar (SoapEnvelope.VER11). Chamamos envelope.setOutputSoapObject(request) para encapsularmos "request" como corpo do envelope SOAP a ser enviado.

Agora criamos um StringBuffer que irá formatar o parâmetro no esquema correto aceito pelo WS. Adicionamos - por meio do método addProperty() - a String “dados_cep” que será uma espécie de chave para seu valor - o parâmetro formatado. Instanciamos um objeto HttpTransportSE (SE é uma referência para Standard Edition de Java uma vez que estamos usando a versão de KSOAP2 para JSE) e passamos como parâmetro a URL para acessar o Web Service. Finalmente, chamamos o método call() de HttpTransportSE (variável httpTransport), passando uma string em branco (não precisaremos de nenhuma SoapAction) e o envelope que construímos anteriormente e esperamos a resposta do servidor. Assim que a resposta é retornada, o valor obtido é recuperado pela chamada envelope.getResponse().

O resultado pode vir em dois formatos: uma String com os valores da localidade, todos concatenados (mais informações do formato retornado, revisitar a seção “Entendendo o funcionamento do Web Service de CEP”) caso o CEP seja válido ou a String “#####” (na verdade é como se o campo de cada informação de localidade estivesse vazio) caso o CEP não seja encontrado. Por fim, instanciamos o objeto CepBean (**Listagem 7**) para encapsularmos os dados retornados da consulta.

Listagem 6. WSConnection.java

```
package android.webmobile.ws.cep;

/* imports */

public class WSConnection {

  private static final String URL = "http://www.maniezo.com.br/webservice/soap-server.php";
  private static final String OPERATION = "traz_cep";
  private static final String NAMESPACE = "http://www.maniezo.com.br/soap-server.php";
  private static final String USERNAME = "ramonrabello";
  private static final String PASSWORD = "trip22";

  public static Object pesquisarCEP(String cep) {

    SoapObject request = new SoapObject(NAMESPACE, OPERATION);
    SoapSerializationEnvelope envelope = new SoapSerializationEnvelope(
        SoapEnvelope.VER11);
    envelope.setOutputSoapObject(request);

    StringBuffer params = new StringBuffer();
    params.append(cep.concat("#"));
    params.append(USERNAME.concat("#"));
    params.append(PASSWORD.concat("#"));

    request.addProperty("dados_cep", params.toString());
    HttpTransportSE httpTransport = new HttpTransportSE(URL);

    try {
      httpTransport.call("", envelope);
      String response = (String) envelope.getResponse();

      if (response.equals("#####"))
        return null;
      else {
        String[] dadosCep = response.split("#");
        CepBean cepBean = new CepBean();
        cepBean.setTipoLogradouro(dadosCep[0]);
        cepBean.setLogradouro(dadosCep[1]);
        cepBean.setComplemento(dadosCep[2]);
        cepBean.setBairro(dadosCep[3]);
        cepBean.setCidade(dadosCep[4]);
        cepBean.setUf(dadosCep[5]);
        cepBean.setCep(cep);
        return cepBean;
      }
    }
    catch (IOException ioex) {
      return null;

    }
    catch (XmlPullParserException e) {
      return null;
```

```
    }
  }
}
```

Listagem 7. CepBean.java

```
package android.webmobile.ws.cep;

public class CepBean {

  private String cep;
  private String uf;
  private String cidade;
  private String logradouro;
  private String complemento;
  private String tipoLogradouro;
  private String bairro;

  /* gets & sets */
}
```

Views, Threads e Handlers em Android

Voltando para WSCepActivity (**Listagem 8**), criamos o método pesquisarCep() que recebe como parâmetro uma String que é o CEP a ser pesquisado e retorna um Object, que no nosso caso será o bean CepBean. Para maior conforto para o usuário, queremos exibir uma mensagem enquanto os dados da localidade não forem obtidos. Para isso, utilizamos a classe ProgressDialog. Chamamos ProgressDialog.show() para mostrarmos esta mensagem e atribuímos à variável progressDialog. Esse método recebe 5 parâmetros: o título, a mensagem, um booleano para indicar se este ProgressDialog é indeterminado (não irá mostrar uma barra crescente informando o status do progresso) e outro booleano informando se este componente pode ser interrompido (clicando no botão voltar, por exemplo) ou não.

É uma boa prática de programação enviarmos para threads separadas partes de código que possa demorar um tempo significativo para a completude do processamento. Dessa forma, assim que o método pesquisarCep() for chamado, instanciamos uma nova thread passando para seu construtor a referência “this” pois a nossa Activity irá implementar Runnable e o método run() que será chamado por start().

O método run() possui o cerne da lógica da aplicação de busca de CEPs. Começamos validando a entrada do CEP, criticando caso o campo de CEP esteja em branco, caso contrário, a busca do CEP deve ser realizada chamando WSConnection.pesquisarCEP().

Uma regra geral: para toda e qualquer mensagem que deva ser mostrada por meio do componente AlertDialog, criamos um objeto Bundle (uma espécie de mapeamento chave/valor) e configuramos o título e a mensagem do diálogo que será mostrado. Depois obtemos uma Message chamando Message.obtain(), passamos o objeto Bundle (variável bundle) para setData() de Message e chamamos o método sendMessage() de um handler, passando o objeto Message que encapsula os dados presentes em bundle. Para que tudo isso? Pois Android, por questão de segurança e arquitetura, não permite que Views e Viewgroups sejam manipulados (instanciados ou seus métodos sejam chamados) fora de sua thread de criação (lembrem-se que estamos dentro de run()). Sendo assim, devemos utilizar Handlers, que são objetos que permitem enviar e processar Messages e Runnables associados com threads de uma MessageQueue (cada nova thread criada possui um MessageQueue).

Basicamente, Handlers são utilizados por dois motivos: escalonar Messages ou Runnables para serem executados em algum momento futuramente ou enfileirar uma ação para ser realizada em uma Thread diferente da sua (que criou o objeto que está sendo utilizado). No nosso caso, temos dois handlers, um para alterar os textos (handler) dos TextViews com as informações da localidade e outro para mostrar os alertas de diálogos (variável handleError). Estes handlers são instanciados assim que ou sendEmptyMessage() ou sendMessage() é chamado. E pó fim, o método handleMessage() é um callback chamado por estes para alterarem os Views. Resumindo: Handlers são uma forma eficiente de implementarmos ações que devam ser executadas por Threads

diferentes da qual a classe foi criada. Para mais detalhes sobre Handlers, ler a API deste componente disponível em *http://code.google.com/intl/pt-BR/android/reference/android/os/Handler.html*.

Listagem 8. WSCepActivity.java

```
/* package e imports */
public class WSCepActivity extends MapActivity implements View.OnClickListener,
    View.OnKeyListener, Runnable
{

  /* declaração de variáveis */

  /* restante do código */

  private void pesquisarCep() {
    progressDialog = ProgressDialog.show(this, "Buscando CEP",
        "Por favor, Aguarde...", true, false);
    Thread t = new Thread(this);
    t.start();
  }

  public void run() {

    if (etCEP.getText().toString().equals("")) {
      Bundle bundle = new Bundle();
      bundle.putString("titulo", "CEP em branco");
      bundle.putString("msg", "Por favor, informe o CEP");
      Message m = Message.obtain();
      m.setData(bundle);
      handlerError.sendMessage(m);
    }
    else {

      cepBean = (CepBean) WSConnection.pesquisarCEP(etCEP.getText().toString().trim());

      if (cepBean != null) {

        try {
          Geocoder geocoder = new Geocoder(this);
          String local = cepBean.getTipoLogradouro() + " " + cepBean.getLogradouro() + ", " +
cepBean.getCidade();
          Address localidade = geocoder.getFromLocationName(local, 1).get(0);

          if (localidade != null) {
            latitude = localidade.getLatitude() * 1E6;
            longitude = localidade.getLongitude() * 1E6;
            geoPoint = new GeoPoint(latitude.intValue(), longitude.intValue());
            mapController = mapView.getController();
            mapController.setZoom(17);
            mapController.animateTo(geoPoint);
            marker = getResources().getDrawable(R.drawable.pin);
            if (mapView.getOverlays().size() > 0)
               mapView.getOverlays().clear();
            mapView.getOverlays().add(new LocationCepOverlay(marker));
            handler.sendEmptyMessage(0);
          }
          else {
            Bundle bundle = new Bundle();
            bundle.putString("titulo", "Localidade não encontrada");
            bundle.putString("msg","Não foi possível encontrar uma localidade no mapa para este
cep.");
            Message m = Message.obtain();
            m.setData(bundle);
            handlerError.sendMessage(m);
          }
        }
        catch (IOException ex) {
          Bundle bundle = new Bundle();
          bundle.putString("titulo", "Erro de IO");
          bundle.putString("msg", "Erro ao obter localidade. Mensagem: " + ex.getMessage());
          Message m = Message.obtain();
          m.setData(bundle);
          handlerError.sendMessage(m);
        } catch (Exception ex) {
```

```
        Bundle bundle = new Bundle();
        bundle.putString("titulo", "Erro Geral");
        bundle.putString("msg", "Ocorreu um erro geral. Mensagem: "
            + ex.getMessage());
        Message m = Message.obtain();
        m.setData(bundle);
        handlerError.sendMessage(m);
      }
    }
    else {
      Bundle bundle = new Bundle();
      bundle.putString("titulo", "CEP não encontrado");
      bundle.putString("msg","Desculpe, o CEP não pôde ser encontrado. Tente novamente.");
      Message m = Message.obtain();
      m.setData(bundle);
      handlerError.sendMessage(m);
    }
  }
}

//Handler para atualizar os dados da localidade do cep
private Handler handler = new Handler() {

  public void handleMessage(Message msg) {
    tvEstado.setText(getString(R.string.estado) + cepBean.getUf());
    tvCidade.setText(getString(R.string.cidade) + cepBean.getCidade());
    tvBairro.setText(getString(R.string.bairro) + cepBean.getBairro());
    tvTipoLograd.setText(getString(R.string.tipo_logradouro)
        + cepBean.getTipoLogradouro());
    tvLograd.setText(getString(R.string.logradouro) + cepBean.getLogradouro());
    tvComplemento.setText(getString(R.string.complemento) + cepBean.getComplemento());
    progressDialog.dismiss();
  }

};

// Handler para mostrar mensagens de erro
private Handler handlerError = new Handler() {

  public void handleMessage(Message msg) {

    if (progressDialog.isShowing()) progressDialog.dismiss();

    tvEstado.setText(getString(R.string.estado));
    tvCidade.setText(getString(R.string.cidade));
    tvBairro.setText(getString(R.string.bairro));
    tvTipoLograd.setText(getString(R.string.tipo_logradouro));
    tvLograd.setText(getString(R.string.logradouro));
    tvComplemento.setText(getString(R.string.complemento));

    String titulo = msg.getData().getString("titulo");
    String mensagem = msg.getData().getString("msg");
    alertDialog = new AlertDialog.Builder(WSCepActivity.this).create();
    alertDialog.setButton("OK", new DialogInterface.OnClickListener() {

      @Override
      public void onClick(DialogInterface dialog, int which) {
        alertDialog.dismiss();
      }
    });

    alertDialog.setTitle(titulo);
    alertDialog.setMessage(mensagem);
    alertDialog.setIcon(android.R.drawable.ic_dialog_alert);
    alertDialog.show();
  }
};
```

Retornando dados geográficos com Geocoder

A partir do momento que obtemos os dados referentes ao CEP (logradouro, cidade, estado, etc) desejado, podemos agora saber a exata localização deste CEP para mostrarmos no mapa. Para isso, utilizaremos o componente android.location.Geocoder (um dos principais componentes da API de Localização) que é o responsável por retornar informações geográficas (latitude/longitude, por

exemplo) de acordo com determinada descrição de localização (este processo é comumente chamado de “geocoding” ou “geocodificação”) ou o vice-versa (chamado de “reverse geocoding” ou “geocodificação reversa”).

No nosso exemplo, na **Listagem 9**, usamos o conceito de geocodificação no momento que chamamos o método getFromLocationName(locationName,maxResults) que retorna uma lista de endereços (List<Address>) de acordo com o filtro passado, onde . locationName representa a(s) localização(ões) que queremos pesquisar; e o inteiro maxResults utilizado para limitar o número máximo de ocorrências encontradas. Como um CEP é específico para somente uma localidade, passamos “1” como valor para maxResults.

Depois, obtemos a latitude (variável latitude) e longitude (variável longitude) do ponto representado pelo CEP. Não esquecendo que multiplicar pela constante 1E6 (um milhão), pois os métodos getLatitude() e getLongitude() de Address (ambos com valor de retorno do tipo double) - são retornados em graus, o que influencia drasticamente na precisão do ponto geográfico a ser mostrado no mapa. Agora iremos encapsular os valores da latitude e longitude por meio da classe GeoPoint, porém seus dois parâmetros devem ser convertidos para inteiros por meio do método intValue(). Depois, obtemos o objeto MapController (variável mapController) responsável por manipular MapViews (zoom, animar em determinada posição do mapa, etc) chamando mapView.getController(). Após, configuramos o nível de zoom para 17 (o nível varia de 1 a 21, inclusive) por meio da chamada mapController.setZoom(17) e fazemos com que o MapView seja animado de acordo com a localização que obtivemos deste cep, passando para o método animateTo() de mapController o objeto GeoPoint.

Listagem 9. WSCepActivity.java

```
/* package & imports */

public class WSCepActivity extends MapActivity implements View.OnClickListener,
    View.OnKeyListener, Runnable
{
  /* código omitido */
  public void run() {

        if (etCEP.getText().toString().equals("")) {
      /* código omitido */
    }
    else {

      cepBean = (CepBean) WSConnection.pesquisarCEP(etCEP.getText().toString().trim());

      if (cepBean != null) {

        try {
          Geocoder geocoder = new Geocoder(this);
          String local = cepBean.getTipoLogradouro() + " "
                       + cepBean.getLogradouro() + ", " +
                         cepBean.getCidade();
          Address localidade = geocoder.getFromLocationName(local, 1).get(0);

          if (localidade != null) {
            latitude = localidade.getLatitude() * 1E6;
            longitude = localidade.getLongitude() * 1E6;
            geoPoint = new GeoPoint(latitude.intValue(), longitude.intValue());
            mapController = mapView.getController();
            mapController.setZoom(17);
            mapController.animateTo(geoPoint);
          }
          else {
            /* código omitido */
          }
        }
        catch (IOException ex) {
          /* código omitido */
        } catch (Exception ex) {
          /* código omitido */
        }
      }
      else {
```

```
        /* código omitido */
      }
    }
  }
}
```

Mostrando a localização do CEP com ItemizedOverlay

Agora só está faltando mostrarmos a localização deste CEP no mapa. Para isso iremos utilizar o componente ItemizedOverlay (subclasse da antiga classe Overlay) que permite visualizar listas de OverlayItems que são os objetos que serão renderizados no mapa. Ele possui as mesmas funcionalidades do componente Overlay, porém com as seguintes vantagens e facilidades para se trabalhar com aplicações com mapas:

- tratar aspectos de direção de desenho (por exemplo, se este deve ser desenhado de norte-a-sul no mapa)
- definir limites de desenho do item (qual a área que será ocupada por um item)
- utilizar uma objeto gráfico (qualquer Drawable) para marcar cada ponto no mapa
- gerenciar evento de foco nos itens
- disparar eventos de toque e mudança de foco de itens por meio de listeners

Voltando para o nosso exemplo, criamos uma classe privada LocationCepOverlay que estende ItemizedOverlay<OverlayItem> para configurarmos o que e onde será mostrado o marcador, que no nosso caso vai ser uma imagem de um alfinete de mapa - que está na pasta res/drawable - e será visualizado nas coordenadas específicas da localidade do CEP.

Continuando, declaramos uma List (variável overlayItems) que conterá todos os OverlayItems e nos será útil para outros métodos sobrescritos. No método construtor, chamamos o construtor da superclasse que recebe também um Drawable que será o marcador padrão que será utilizado. Instanciamos um OverlayItem e passamos como parâmetro, respectivamente, o ponto geográfico (representado pela variável geoPoint) que foi obtido durante a geocodificação, uma String para o título e outra para a descrição deste elemento. Depois, adicionamos este item em overlayItems e chamamos o método populate(). Esse método serve como um utilitário para ItemizedOverlays (e suas subclasses) e recomenda-se chamá-lo assim que algum dado (OverlayItem) estiver disponível. O método createItem() recebe como parâmetro um inteiro que corresponde o índice do OverlayItem na lista e retona um OverlayItem na posição i. Ele é intimamente ligado a populate(), uma vez que somente createItem() é chamado por ele para popular os OverlayItems a fim de serem visualizados no mapa.

Seguindo, sobrescrevemos o método draw(), definido em Overlay que é utilizado para desenhar os objetos bráficos na tela (utilizando Canvas) e recebe como parâmetro três objetos: um Canvas, representando a tela que será desenhada com os objetos gráficos (imagens, formas, etc), o nosso MapView que irá ser sobreposto por este Overlay; e um booleano informando se este Overlay deve projetar sombras. Depois, chamamos o métodoso draw() da superclasse e boundCenterBottom() que recebe como parâmetro um Drawable (marker). Este método é útil para centrarmos o centro da imagem no pixel(0,0) e ele é específico para ser utilizado por aplicações que mostram a localização no mapa por meio de itens com aspectos de alfinete (conhecidas como pin-based applications em Inglês).

Sobrescrevemos por fim o método size() para retornarmos a quantidade de OverlayItems na lista pois populate() também chama este método. Para retornarmos nossa imagem como um Drawable, chamamos getResources().getDrawable() e passamos para o segundo método o inteiro com o identificador de nossa imagem na classe utilitária de recursos R.java (R.drawable.pin) e atribuímos para marker, que é um Drawable. Testamos se existe algum OverlayItem na lista e a limpamos chamando mapView.getOverlays().clear() para garantir que só haverá um OverlayItem sendo mostrado no mapa, que será a localidade do CEP. Porém, para que possamos visualizar o alfinete no

mapa, precisamos adicionar LocationCepOverlay ou nosso MapView, realizamos o comando mapView.getOverlays().add() e passamos para add() uma instância de nosso ItemizedOverlay para podermos visualizar o alfinete na exata posição geográfica. A **Listagem 10** mostra o código referente ao nosso ItemizedOverlay e as alterações necessárias para visualização do item no mapa. A **Figura 3** mostra um exemplo da nossa aplicação em execução utilizando o ItemizedOverlay que criamos.

Listagem 9. WSCepActivity.java

```
/* package & imports */

public class WSCepActivity extends MapActivity implements View.OnClickListener,
    View.OnKeyListener, Runnable
{

  /* declaração de variáveis */

  /* código omitido */
     public void run() {

        try {
          localidade = geocoder.getFromLocationName(cepBean.getTipoLogradouro() + " " +
cepBean.getLogradouro() + ", " + cepBean.getCidade(), 1).get(0);

          if (localidade != null) {
            /* código omitido */
            marker = getResources().getDrawable(R.drawable.pin);
            mapView.getOverlays().add(new LocationCepOverlay(marker));
            handler.sendEmptyMessage(0);
          }
          else {
            /* código omitido */
          }
        }
        catch (IOException ex) {
          /* código omitido */
        } catch (Exception ex) {
          /* código omitido */
        }
     }
  }

  /* restante do código */
  private class LocationCepOverlay extends ItemizedOverlay<OverlayItem> {
    List<OverlayItem> overlayItems = new ArrayList<OverlayItem>();

    public LocationCepOverlay(Drawable defaultMarker) {
      super(defaultMarker);
      OverlayItem item = new OverlayItem(geoPoint, "Dados da Localidade", cepBean.getLogradouro());
      overlayItems.add(item);
      populate();
    }

    @Override
    protected OverlayItem createItem(int i) {
      return overlayItems.get(i);
    }

    @Override
    public void draw(Canvas canvas, MapView mapView, boolean shadow) {
      super.draw(canvas, mapView, shadow);
      boundCenterBottom(marker);
    }

    @Override
    public int size() {
      return overlayItems.size();
    }
  }
}
```

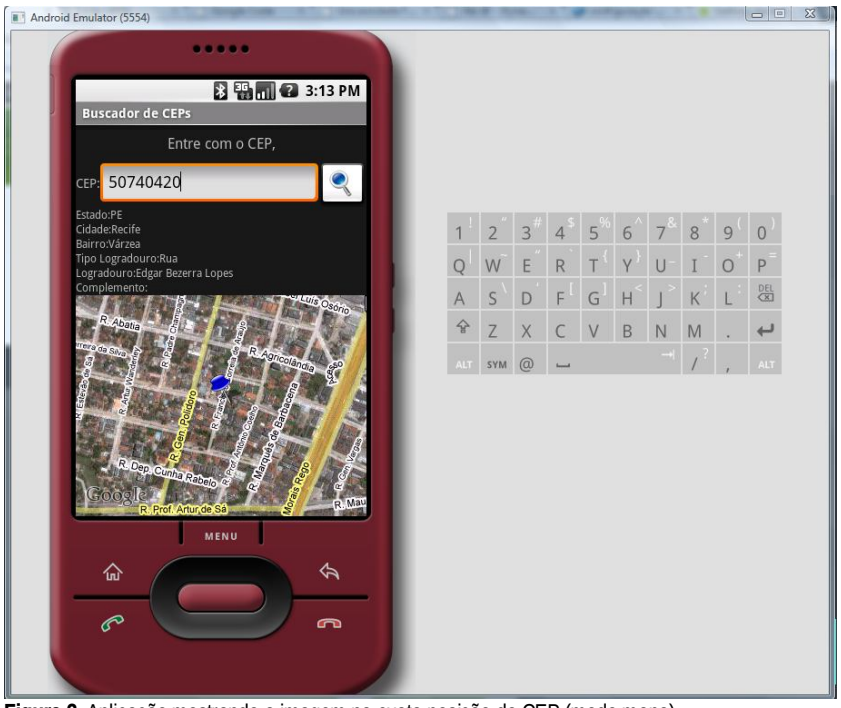

Figura 3. Aplicação mostrando a imagem na exata posição do CEP (modo mapa).

Conclusões

Como você pôde perceber, Android ainda não possui uma API que já venha embutida que trate especificamente de Web Services. Sendo assim, para ser possível trabalharmos com WS, Android possui a capacidade de integrar as mesmas bibliotecas de Java SE, como vimos com a versão de KSOAP2, em aplicações comerciais reais.

A plataforma Android, por mais que ainda esteja muito precoce (1 ano e alguns meses na época deste artigo), tem provado que a cada release do SDK, as funcionalidades que foram apresentadas no dia do seu anúncio ao público mundial tornam-se realidade tanto para nós desenvolvedores quanto para os usuários finais do dispositivo. Até breve, androiders!

Links

Eclipse Ganymede
http://www.eclipse.org/ganymede/
Site do SDK do Android
http://code.google.com/android/
Obtendo chave privada (API Key) para MapViews
http://code.google.com/intl/pt-BR/android/toolbox/apis/mapkey.html
Registrando uma API Key para MapView
http://code.google.com/android/maps-api-signup.html
Web Service de Busca de CEP
http://www.maniezo.com.br
API do componente Handler
http://code.google.com/intl/pt-BR/android/reference/android/os/Handler.html
API do componente Geocoder
http://code.google.com/intl/pt-BR/android/reference/android/location/Geocoder.html
API do componente ItemizedOverlay
http://code.google.com/android/reference/com/google/android/maps/ItemizedOverlay.html