Dispositivos Móveis

Aula 14 - Internet

Apresentação

E chegamos à nossa última aula com funcionalidades novas para nossa aplicação. Na próxima aula, veremos como compilar a aplicação que tenhamos desenvolvido e disponibilizar no Google Play, passando por todos os passos necessários antes da publicação.

Na aula de hoje, veremos como utilizar a Internet no desenvolvimento de aplicações Android. Como sabemos, a Internet está presente na grande maioria dos dispositivos Android, seja através de conexões 3G, 4G ou redes Wi-Fi. Hoje veremos como utilizar a classe WebView e também revisaremos a utilização do *browser* em nossa aplicação. Além disso, veremos como utilizar conexões HTTP dentro do Android para fazer requisições. Veremos os dois principais tipos de requisição, GET e POST e também como poderemos fazer para evitar a perda de desempenho da aplicação quando executando essas requisições.

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você deverá ser capaz de:

- Utilizar a classe WebView dentro de sua aplicação.
- Lançar o *browser* para navegação externa quando necessário.
- Fazer requisições web em sua aplicação através do HttpUrlConnections.
- Utilizar AsyncTasks para fazer requisições assíncronas.

Acessando Páginas da Internet

A WebView é a View responsável por exibir páginas da Web dentro de sua aplicação. Ela simula o comportamento de um *browser*, porém, pode ser adicionada a sua Activity como uma View qualquer, permitindo que você posicione e adeque o espaço que o conteúdo mostrado terá dentro de sua aplicação. Note que o WebView não é um *browser* e, assim, algumas funcionalidades que estariam disponíveis no *browser* podem não estar disponíveis no WebView.

Para sermos capazes de utilizar o WebView em nossa aplicação, como já vimos em outros exemplos que utilizam conectividade, precisamos declarar no Manifest a permissão necessária para tal. Isso pode ser feito utilizando a seguinte linha:

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

É importante notarmos que o WebView tem seu uso planejado apenas para exibir páginas simples e que, por esse motivo, não costumam fornecer ao usuário interação com essas páginas. Widgets do *browser* não são carregados para o WebView e algumas interações com a página são ignoradas quando estamos utilizando essa view.

Com essas características, podemos perceber que o WebView serve mais para exibir pequenas páginas em sua aplicação do que para substituir completamente o *browser*. Exibir o contrato final com o usuário ou alguma espécie de guia que pode mudar constantemente são exemplos de pequenas páginas que se encaixariam bem dentro de um WebView. Mas, caso um *browser* completo, com mais funcionalidades, seja realmente necessário em sua aplicação, podemos utilizar o código mostrado na **Listagem 1** para iniciar um *browser* externo completo.

```
1 Uri uri = Uri.parse("http://10.0.2.2:8080");
2 Intent intent = new Intent(Intent.ACTION_VIEW, uri);
3
4 startActivity(intent);
```

Listagem 1 - Lançando um Intent para o *browser* externo do Android.

Na aula de Intent, já tínhamos visto como fazer esse acesso, então não deve ser nenhum problema lembrar como funciona. Uma vez lançado, o *browser* funcionará como o normal do Android, permitindo ao usuário navegar entre páginas, acessar *links*, executar JavaScript, entre outras funcionalidades. Ao apertar a tecla voltar no *browser*, o usuário navegará voltando, até que esteja na página inicial. Quando esta for atingida, a tecla voltar levará o usuário de volta a Activity que lançou o *browser* dentro de sua aplicação. Agora que já vimos brevemente o funcionamento do *browser*, vamos entender como adicionar o WebView em sua aplicação.

Como qualquer outro componente, o WebView pode ser declarado no XML ou programaticamente. A **Listagem 2** nos mostra como declarar o WebView no XML e a **Listagem 3** como criar um WebView que ocupará a Activity inteira, de maneira programática.

```
1 <Webview
2    android:id="@+id/Webview1"
3    android:layout_height="match_parent"
4    android:layout_width="match_parent" />
```

Listagem 2 - Declarando o WebView no XML.

```
1 @Override
2 public void onCreate(Bundle savedInstanceState) {
3    super.onCreate(savedInstanceState);
4    WebView webview = new WebView(this);
5    setContentView(webview);
6 }
```

Listagem 3 - Criando um WebView que ocupa toda a janela da Activity.

Ambos os códigos produzirão o mesmo resultado, que é um WebView de tela inteira. Perceba que um WebView inicializado dessa maneira não exibirá nenhuma página. Para isso, precisaremos carregar uma página para o WebView. Para fazermos isso, devemos utilizar o método *loadUrl(),* como visto na **Listagem 4**.

```
public class MainActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        WebView mWebview = (WebView) findViewById(R.id.Webview1);
        mWebview.loadUrl("http://10.0.2.2:8080");
    }
}
```

Listagem 4 - Carregando um URL no seu WebView

Perceba que tanto nesse exemplo quanto no primeiro apresentado, vemos a URL 10.0.2.2. Ela é o caminho utilizado pelo emulador para levar o desenvolvedor ao localhost (ou 127.0.0.1) da máquina que está o hospedando. Isso é necessário, pois o emulador não tem acesso direto ao IP da máquina. Para que seja possível acessá-lo, o Android fornece essa interface para facilitar a interação.

Um último ponto importante que deve ser ressaltado sobre o WebView é que ele permite também a execução de código HTML direto. Essa interface não é capaz de trabalhar com qualquer tipo de código HTML, mas pode ser bastante interessante quando se trata de códigos mais simples. Na **Listagem 5**, podemos ver um exemplo de como isso é feito.

```
public class MainActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        WebView mWebview = (WebView) findViewById(R.id.Webview1);
        String pontuacao = "<html><body>Sua pontuacao foi de <b>10</b> pontos.</body></html>";
        mWebview.loadData(pontuacao, "text/html", null);
    }
}
```

Listagem 5 - Carregando HTML no WebView

Perceba que também é possível trazer esse código de um arquivo que tenha um HTML um pouco mais robusto definido. Basta que tenhamos uma *String* para o *loadData()*. A referência desse método é capaz de fornecer maiores informações sobre o que é possível e o que não é de se fazer dentro desse método.

Pronto! Com esses conhecimentos já somos capazes de criar WebViews ou lançar *browsers* externos de nossas aplicações. Agora que já sabemos como exibir conteúdo da internet, vamos aprender como podemos consumir o conteúdo fornecido por uma página. Para tal, utilizaremos métodos de conexão HTTP para gerar uma tela de *login* como exemplo de comunicação entre o Android e páginas da internet.

Vídeo 02 - Uso Avançado do Webview

Atividade 01

1. Crie uma Activity que possua dois WebViews, um carregando a página do Google e outro exibindo um texto com formatação HTML.

Conexões HTTP

Internet, *login*, busca de dados armazenados remotamente, *ranking* de pontuação online... Esses são apenas alguns exemplos do que podemos adicionar à nossa aplicação a partir do momento que aprendemos a utilizar conexões HTTP e HTTPS feitas diretamente de nossa aplicação.

O Android dispõe de três métodos para fazer conexões HTTP. Podemos utilizar a API padrão do Java através da classe java.net.HttpUrlConnection, o *framework Commons HttpClient* do Apache, com a classe org.apache.http.client.HttpClient, ou ainda através da classe android.net.http.AndroidHttpClient, que existe a partir da versão 2.2 do Android. Todos esses métodos tem o mesmo objetivo final: comunicação via HTTP.

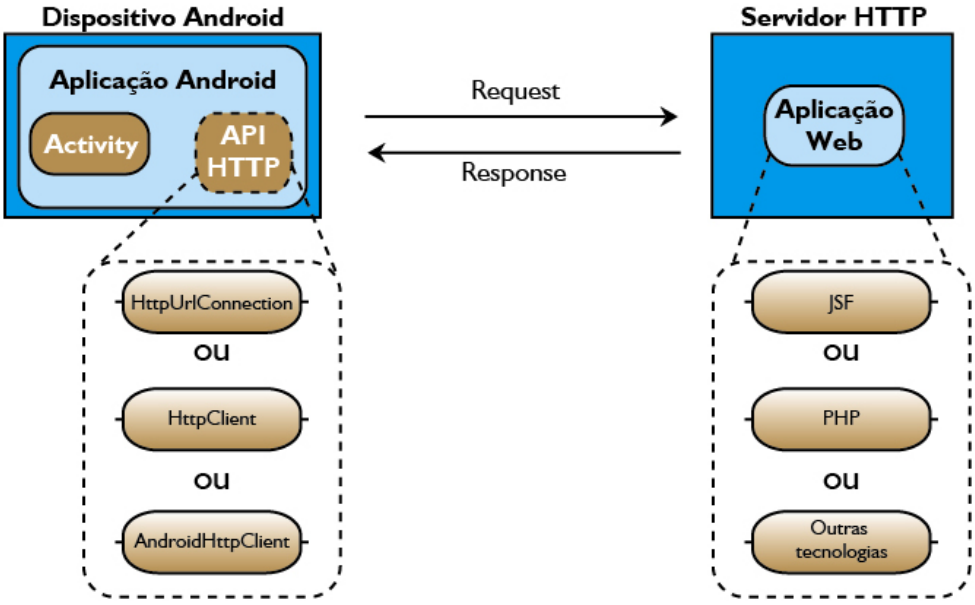
Cada método entre os apresentados utiliza uma maneira diferente para fazer sua comunicação. No escopo dessa aula, estudaremos o primeiro deles, o HttpUrlConnection. Através dessa classe, padrão na API Java, podemos fazer todas

as conexões que nos forem necessárias e acreditamos que o estudo dessa classe servirá como base para qualquer outra, uma vez entendida a lógica utilizada por trás da API.

Antes de olharmos o código do lado Android da comunicação, vamos ver o que precisamos do lado do servidor. Para a comunicação do Android com o servidor, é necessário que o servidor entenda as requisições recebidas e também que ele seja capaz de responder do jeito que é esperado pelo requisitante. Esse protocolo a ser utilizado pelas chamadas deve ser definido por você, desenvolvedor, e deve mudar o mínimo possível durante o ciclo de vida de sua aplicação, para evitar problemas com clientes que não estejam atualizados.

Para que haja uma comunicação bem sucedida entre o cliente e o servidor, é importante notar que os dois lados precisam implementar códigos que suportem requisições HTTP. O que está por trás dessas requisições não é importante no caso dessa comunicação. Não importa se o servidor está sendo executado baseado em Java, em PHP ou em qualquer outra linguagem. O importante para que a comunicação funcione é que os dois lados possam implementar corretamente o protocolo HTTP e entendam as requisições que lhe forem feitas, sejam elas do tipo GET, POST ou qualquer outra. A **Figura 1** ilustra a abstração, mostrando que o que importa na comunicação é realmente o protocolo da comunicação.

Figura 01 - Comunicação HTTP entre um dispositivo Android e o servidor.



O método GET, no protocolo HTTP, é responsável por requisições que buscam dados do servidor. É possível passar alguns parâmetros com o GET para obter resultados diferentes, como veremos dentro do nosso exemplo. Vejamos agora a **Listagem 6**, que demonstra o código do Servlet que receberá as requisições do Android e retornará uma mensagem de boas vindas, que posteriormente será lida no Android.

```
@WebServlet("/ServletHello")
public class ServletHello extends HttpServlet {
    public ServletHello() {
        super();
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Servl
        PrintWriter out = response.getWriter();
        String mobileId = request.getParameter("id");

        String responseText = "";
        if(mobileId != null){
            System.out.println("Cliente identificado: "+mobileId);
            responseText = "Ola "+mobileId+" ! Sou seu servidor HTTP";
        }
        else{
            System.out.println("Cliente anonimo!");
            responseText = "Ola ! Sou seu servidor HTTP";
        }
        out.println(responseText);
        out.flush();
        System.out.println("Resposta enviada...");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
                throws ServletException, IOException {
            doGet(req, resp);
    }
}
```

Listagem 6 - Servlet que receberá as requisições de nosso exemplo.

Perceba que a primeira linha do exemplo indica o caminho que deveremos utilizar dentro de nosso servidor para chegar até o Servlet. Como estaremos executando o servidor em nossa máquina e acessando pela interface local, o endereço para acessar esse Servlet será o http://10.0.2.2:8080/nome-do-app/ServletHello. É importante sempre lembrar de utilizar o endereço 10.0.2.2 ao invés de 127.0.0.1, ou mesmo localhost, quando estivermos executando a aplicação num emulador Android.

Dentro da classe, percebemos que além de ela estender HttpServlet, ela deve implementar o método *doGet(request, result)*, que será responsável por processar as requisições GET que forem recebidas. Além disso, o método *doPost(request, result)* também está ilustrado no exemplo, pois as requisições POST também são de suma importância para a comunicação bem sucedida entre cliente e servidor. Páginas de *login* costumam utilizar esse tipo de requisição para realizar a autenticação do usuário, pois os dados não são passados explicitamente, pelo endereço da solicitação.

Com esse Servlet implementado e rodando dentro de um servidor na máquina local, podemos partir para o lado Android da comunicação. Nesse exemplo, estaremos apenas enviando uma requisição a esse Servlet e então exibindo a mensagem recebida ao usuário, utilizando um Toast. Vejamos a **Listagem 7**, que contém o código responsável por executar essa requisição.

```
public class HTTPURLCommunication extends Activity {
    private static String ENDERECO = "http://10.0.2.2:8080/aula14/ServletHello";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Toast.makeText(this, connect("Aluno"), Toast.LENGTH_LONG);
    }
    public static String connect() {
        HttpURLConnection connection = null;
        BufferedReader rd = null;
        String line = null;
        URL serverAddress = null;
    try {
        serverAddress = new URL(ENDERECO);
        connection = (HttpURLConnection) serverAddress.openConnection();
        connection.setRequestMethod("GET");
        connection.connect();
        rd = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        line = rd.readLine();
    } catch (MalformedURLException e) {
        Log.e("ERROR", e.getMessage(), e);
    } catch (IOException e) {
        Log.e("ERROR", e.getMessage(), e);
    }
    return line;
}
public static String connect(String id) {
    HttpURLConnection connection = null;
    BufferedReader rd = null;
    String line = null;
    String request = "?id="+id;
    URL serverAddress = null;
    try {
        serverAddress = new URL(ENDERECO+request);
        connection = (HttpURLConnection) serverAddress.openConnection();
        connection.setRequestMethod("GET");
        connection.connect();
        rd = new BufferedReader(new InputStreamReader( connection.getInputStream()));
        line = rd.readLine();
    } catch (MalformedURLException e) {
        Log.e("ERROR", e.getMessage(), e);
    } catch (IOException e) {
        Log.e("ERROR", e.getMessage(), e);
    }
    return line;
    }
}
```

Listagem 7 - Código do cliente para excutar uma requisição GET ao ServletHello.

Na Activity apresentada na **Listagem 7**, implementamos apenas o método *onCreate()* para inicializar a Activity e fazer a requisição, através do método *connect(id)*, que passa um ID para o servidor que é retornado junto a resposta. Esse parâmetro, em situações reais, é muito importante, pois pode ser através desse ID que a consulta ao banco será feita, ou alguma consulta de outro tipo que necessite de um parâmetro por parte de quem fez a requisição.

O método *connect()* é o método responsável por fazer a conexão. Como podemos ver, ele utiliza a API HttpUrlConnection para tal. O primeiro passo para fazer a requisição bem sucedida é criar um objeto do tipo URL que conterá o endereço da requisição a ser feita, juntamente do parâmetro da requisição, se for necessário, como mostrado no método *connect(id)*. Uma vez criado o URL, precisamos abrir uma conexão HTTP com o servidor com o qual faremos a comunicação. Para isso, criamos um objeto do tipo HttpUrlConnection e então requisitamos ao objeto URL que abra a conexão com o endereço que foi configurado. Ao abrir essa conexão, o URL retornará um objeto que pode ser transformado em um HttpUrlConnection, através de um *cast*.

Uma vez que tenhamos o objeto HttpUrlConnection com a conexão aberta, devemos configurar o método de requisição a ser utilizado, através do método *setRequestMethod(),* e então conectar ao servidor. Para isso, utilizamos o método *connect()* do objeto HttpUrlConnection.

Após a conexão ser realizada, se esta tiver sucesso, poderemos receber dados através de um objeto BufferedReader. Utilizando o método *getInputStream()* do HttpUrlConnection, podemos ler uma sequência de dados que será enviada pelo servidor. Essa sequência, quando passada por um InputStreamReader torna-se compreensível e pode ser armazenada em *buffer* através de um objeto BufferedReader. Perceba que é importante passar por esses dois passos para que tenhamos acesso correto aos dados que estejam sendo enviados pelo servidor.

Uma vez que tenhamos acesso a um objeto do tipo BufferedReader, podemos utilizar o método *getLine()* para pegar uma linha de conteúdo do que tenha sido enviado, no tipo string. Uma vez que tenhamos esse objeto do tipo string, podemos trabalhar com o que o servidor nos tenha retornado. No caso da Activity apresentada, esse valor é utilizado em um Toast para ser exibido ao usuário.

Perceba que durante o processo de conexão alguns erros podem ocorrer. Esses erros, exceções do tipo IOException ou MalformedURLException, precisam ser tratados utilizando *try/catch* ou então lançados para cima. Caso sejam lançados, é necessário ter certeza que alguém acima de sua Activity será capaz de tratar esses erros, ou eles poderão gerar o encerramento inesperado de sua aplicação.

Também é importante notar que fazer conexões HTTP leva tempo. Caso conexões sejam feitas como no exemplo exibido, dentro da *thread* principal, é muito provável que haja uma grande pausa na aplicação, o que pode gerar desconforto para o usuário e, principalmente, uma Activity Not Responding (ANR) que pode levar ao encerramento da aplicação. Para evitar esse tipo de perda de *desempenho*, o Android disponibiliza ao usuário uma interface para realizar requisições assíncronas. Vejamos agora como essa interface funciona.

Vídeo 03 - API Apache HTTP

Implementando AsyncTasks

O AsyncTask é uma interface que o Android fornece para que os desenvolvedores possam lançar operações bloqueantes em uma nova *thread*, na qual executará a requisição e retornará o resultado à *thread* principal, sem que essa seja bloqueada e sem a necessidade do desenvolvedor tratar eventos de comunicação assíncrona.

A utilização dessa classe é simples. Implemente uma subclasse privada em sua Activity que implemente os métodos *doInBackground()* e *onPostExecute().* Esses métodos são os responsáveis por executar as requisições em uma *thread* separada em *background* e atualizar a *thread* principal, respectivamente. Uma vez implementados, basta chamar o método *execute()* dessa subclasse em sua Activity e a requisição será processada, de forma assíncrona. Vejamos a **Listagem 8** para

entender melhor como funciona esse processo. Implementaremos o mesmo código do exemplo anterior, apenas omitindo os métodos de conexão que se manterão iguais.

```
1  public class HTTPURLCommunication extends Activity {
2     private static String ENDERECO ="http://10.0.2.2:8080/aula14/ServletHello";
3
4     Context context = getApplicationContext();
5     @Override
6     protected void onCreate(Bundle savedInstanceState) {
7        super.onCreate(savedInstanceState);
8        setContentView(R.layout.main);
9        new Requisicoes().execute("Aluno");
10    }
11    public static String connect() {
12       ...
13    }
14    public static String connect(String id) {
15       ...
16    }
17
18    private class Requisicoes extends AsyncTask<String, Void, String> {
19       protected String doInBackground(String... id) {
20          return connect(id[0]);
21       }
22       protected void onPostExecute(String result) {
23          Toast.makeText(context, result, Toast.LENGTH_LONG).show();
24       }
25    }
26 }
```

Listagem 8 - Fazendo requisições em segundo plano através da classe AsyncTask.

Perceba que a classe AsyncTask precisa de três tipos em sua declaração. Esses tipos são o tipo dos parâmetros, o tipo do progresso e o tipo do resultado final. O tipo do progresso raramente é utilizado. Ele serve para determinar que tipo será retornado quando houver progresso na requisição que a *thread* está executando. Já os outros dois parâmetros são extremamente importantes. O primeiro deles indica o tipo dos parâmetros que o método *doInBackground()* receberá para utilizar na requisição. O último declara o tipo do objeto que será retornado pelo método *doInBackground().* O retorno desse método é automaticamente passado como parâmetro para o método *onPostExecute()*, então, podemos dizer que esse valor é também o tipo do parâmetro do método *onPostExecute().* Com essas informações, você deve ser capaz de implementar requisições a serem executadas em segundo plano, da maneira que todas as conexões dentro de uma aplicação devem ser feitas.

É importante notar que sempre que precisarmos utilizar o valor retornado pela requisição, deveremos fazer isso dentro do método *onPostExecute()*, pois como a requisição será executada em outra *thread*, a *thread* principal vai continuar e pode terminar por utilizar o valor antes de ele terminar de ser obtido na outra *thread*, o que vai gerar um erro. Já no caso do valor ser utilizado somente dentro do *onPostExecute()*, podemos ter certeza de que ele só será chamado após a execução completa da requisição solicitada. Com esse conhecimento sobre como as requisições web devem ser feitas dentro de uma aplicação Android, encerramos a nossa aula. Até a próxima, que será nossa última!

Vídeo 04 - Restful Webservices

Atividade 02

1. Crie um Servlet que responda, no método GET, a uma requisição que passa um parâmetro e recebe a string "OK" se esse parâmetro for a string "Aluno".

2. Crie uma Activity que envie a esse Servlet uma requisição passando o parâmetro "Aluno" e mostrando uma mensagem de sucesso se a resposta for "OK".

3. Separe a requisição da *thread* principal utilizando um AsyncTask.

Leitura Complementar

- **Building Web Apps in WebView.** Disponível em: <https://developer.android.com/guide/webapps/webview>. Acesso em: 09 maio de 2018.

- **HttpUrlConnection.** Disponível em: <http://developer.android.com/HttpURLConnection.html>. Acesso em: 08 jun. 2015.

- **Processes and Threads.** Disponível em: <http://developer.android.com/processes-and-threads.html>. Acesso em: 08 jun. 2015.

Resumo

Nesta aula, vimos diversos aspectos relacionados à internet no Android. Começamos a aula vendo como utilizar um WebView e como lançar o *browser* externo a nossa aplicação. Estudamos o funcionamento de um WebView para ajudar a decidir quando cada caso é pertinente e deve ser utilizado. Também vimos como executar pequenos pedaços de códigos HTML no WebView. Em seguida, na segunda parte da aula, estudamos uma das APIs utilizadas para requisições HTTP feitas pelo Android. Vimos como ela funciona e como implementar, através de exemplos de código tanto para o lado servidor quando para o lado cliente. Vimos também o problema que requisições web podem gerar se feitas diretamente da classe principal. Para evitar isso, vimos, na terceira parte da aula, como fazer requisições assíncronas. Estudamos brevemente a classe AsyncTask e a utilização de seus métodos para fazer requisições em *threads* separadas da *thread* principal. Vimos também como essa interface nos deixa interagir tanto com a *thread* que executará a requisição quanto com a *thread* principal sem que tenhamos que lidar com mensagens entre *threads*. Espero que tenha sido uma aula muito produtiva para todos, pois ela tratou de um assunto, sem dúvida, muito interessante e importante. Até mais.

Autoavaliação

1. Avalie as diferentes maneiras de abrir uma página Web a partir de uma aplicação Android e descreva situações onde cada uma das abordagens é mais indicada.

2. Explique como uma aplicação Android poderia se comunicar com uma aplicação feita em PHP ou .NET. Há diferenças para aplicação Android se o servidor é desenvolvido em uma das tecnologias? Justifique.

Referências

Android Developers, 2015. Disponível em: <>. Acesso em 08 jun. 2015.

DIMARZIO, J. **Android: A Programmer's Guide**. McGraw-Hill, 2008. Disponível em: <>. Acesso em: 08 jun. 2015.

LECHETA, RICARDO R.**GOOGLE ANDROID** - APRENDA A CRIAR APLICAÇOES. 2. ed. NOVATEC, 2010.

LECHETA, RICARDO R.**GOOGLE ANDROID PARA TABLETS**. NOVATEC, 2012.

MEIER, R. **Professional Android 2 Application Development.** John Wiley e Sons, 2010. Disponível em: <>. Acesso em: 08 jun. 2015.