soluções mobile

prof. Thyerri Mezzari

Android: Permissions e Requisições HTTP +Volley 🏐

Permissions

<uses-permission>

Por que pedir permissões?

Um dos pontos de projeto centrais da arquitetura de segurança do SO Android é que, por padrão, nenhum app tem permissão de realizar nenhuma operação que prejudique outros apps, o sistema operacional ou o usuário.

Isso abrange a leitura e a gravação de dados privados do usuário (como contatos ou e-mails), leitura ou gravação dos arquivos, realização de acesso de rede (internet), manter a tela do dispositivo ativo etc.

Apps para Android precisam solicitar permissão para acessar dados confidenciais do usuário (como contatos e SMS), bem como recursos específicos do sistema (como câmera, galeria de fotos e Internet). Dependendo do recurso, o sistema pode conceder a permissão automaticamente ou pedir ao usuário que aprove a solicitação.

Permissions

Obrigatoriamente um app precisa divulgar as permissões necessárias incluindo tags *<uses-permission>* no arquivo manifesto do app.

Por exemplo um aplicativo que precise enviar mensagens SMS de maneira programática teria que adicionar a seguinte linha em seu ***AndroidManifest.xml***:

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

Caso seu app precise consumir algum recurso on-line, baixar algo ou fazer alguma requisição:

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

Permissions

Caso seu app necessite permissões de nível normais no ***AndroidManifest.xml*** (ou seja, permissões que não apresentam muito risco à privacidade do usuário nem à operação do dispositivo), o sistema concederá automaticamente essas permissões.

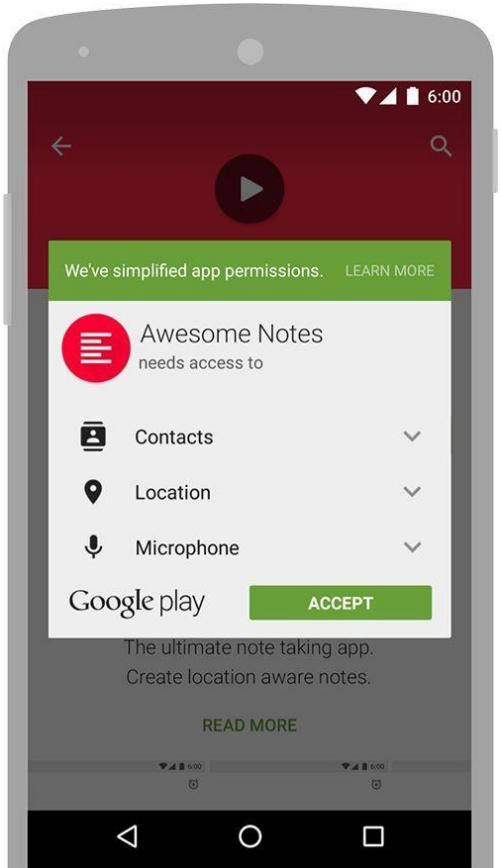
Caso seu app use permissões perigosas no manifesto (ou seja, permissões que possam afetar a privacidade do usuário ou a operação do dispositivo), como a permissão *SEND_SMS*, o usuário precisa concordar explicitamente em conceder essas permissões.

Segue um link de consulta rápida para ver quais são algumas permissões normais (sem bronca) e permissões perigosas (que precisa solicitar ao usuário):

https://stackoverflow.com/a/36937109

Solicitando Permissões

Se o dispositivo estiver executando o **Android 5.x** (API de nível 22) ou anterior, ou o ***targetSdkVersion*** do app for 22 ou anterior, durante a execução em qualquer versão do Android, o sistema solicitará automaticamente que o usuário conceda todas as permissões perigosas para seu app no momento da instalação.



Solicitando Permissões

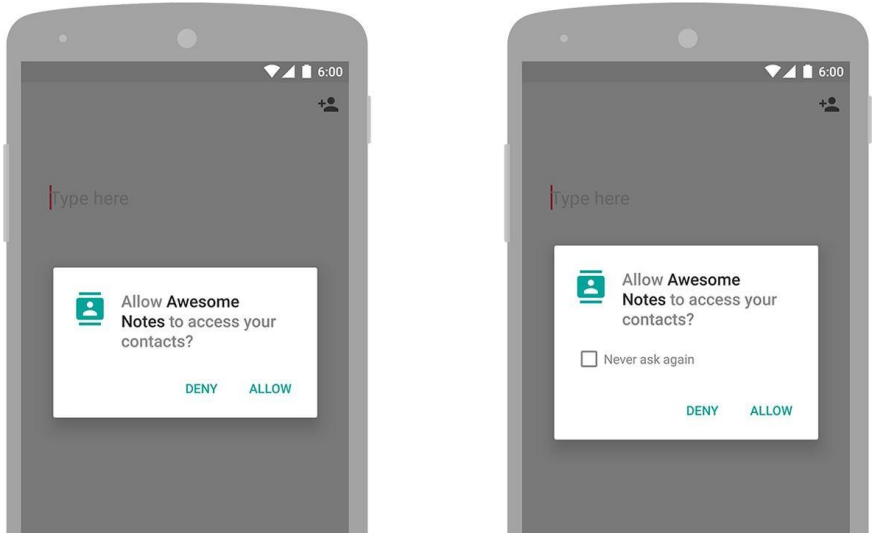
Se o dispositivo estiver executando o **Android 6.x** (API de nível 23) ou mais recente, e o ***targetSdkVersion*** do app for 23 ou versões mais recentes, o usuário não receberá nenhuma notificação sobre permissões do app no momento da instalação. O app precisa pedir que o usuário conceda permissões perigosas no momento da execução do aplicativo.

Quando o app solicitar permissão, o usuário verá uma caixa de diálogo do sistema informando qual grupo de permissões o app está tentando acessar. **A caixa de diálogo inclui os botões Negar e Permitir**.

Se o usuário negar a solicitação de permissão, na próxima vez que o app solicitar a permissão, a caixa de diálogo terá uma caixa de seleção que, quando marcada, indica que o usuário não quer que a permissão seja solicitada novamente.

Solicitando Permissões

Se o usuário marcar a caixa Não perguntar novamente e tocar em Negar, o sistema não pedirá mais a permissão ao usuário se o app tentar solicitar a mesma permissão posteriormente.



Solicitando Permissões

Os princípios básicos para solicitar permissões são os seguintes:

- Solicite permissões no contexto/momento, quando o usuário começar a interagir com o recurso que precisa da permissão.
- Não bloqueie o usuário. Sempre ofereça a opção de cancelar um fluxo relacionado as necessidades de permissões.
- Se o usuário negar ou revogar uma permissão necessária para um recurso, faça uma *downgrade* suave para que ele possa continuar usando o app, possivelmente desativando o recurso que requer a permissão e oferecendo algo mais básico.

Solicitando Permissões

Depois de ter adicionado no arquivo ***AndroidManifest.xml*** a tag ***<uses-permission>*** referente a permissão que você deseja utilizar, caso você esteja necessitando de um recurso considerado "perigoso" será necessário programaticamente solicitar a permissão do usuário para aquela consulta.

Sempre que for usar um recurso de câmera, por exemplo, inicie o código verificando primeiro se o usuário já não concedeu aquela permissão anteriormente através do método `ContextCompat.checkSelfPermission()`. Esse método retorna ***PERMISSION_GRANTED*** ou ***PERMISSION_DENIED***, dependendo se o app já têm ou não a permissão.

Solicitando Permissões

Se o método **`ContextCompat.checkSelfPermission()`** retornar ***PERMISSION_DENIED***, chame **`shouldShowRequestPermissionRationale()`**. Se este método retornar true, mostre uma "tela" explicativa para o usuário, uma tela que descreva o motivo pelo qual o recurso que o usuário quer ativar precisa de uma permissão específica.

Após explicar ao usuário porque você precisa acessar aquele recurso do smartphone dele, você poderá utilizar o método **`requestPermissions`** para mostrar a caixa de diálogo do Android aonde o usuário poderá autorizar ou negar aquela permissão.

Solicitando Permissões

O resultado/resposta do usuário sobre a permissão irá chegar em um método chamado **`onRequestPermissionsResults`** de sua *Activity* utilizada no momento. Dentro deste método você poderá tomar um ação de continuidade de sua necessidade caso o usuário permita o acesso OU decidir o que fazer caso a permissão não seja concedida pelo usuário.

Para mais detalhes sobre este processo sugiro uma consulta no link oficial:
https://developer.android.com/training/permissions/requesting#manage-request-code-yourself

Requisições HTTP

android.permission.INTERNET

Conectando-se a rede

Assim como vimos em nossas aulas, banco de dados/persistência de dados são atividades bem comuns em aplicativos, o acesso e consumo de dados on-line através de requisições (normalmente HTTP) também é classificado com uma atividade rotineira. Quanto maior o porte do app e a utilidade maior a chance de você precisar usar um serviço HTTP/REST.

Conectando-se a rede

Para iniciar essa implementação de recursos on-line, a primeira coisa que temos que fazer é adicionar duas permissões a seu AndroidManifest.xml:

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

As permissões de acesso a internet (android.permission.INTERNET) e o controle de estado da rede (android.permission.ACCESS_NETWORK_STATE) formam o conjunto base de estratégias para você navegar.

A primeira irá lhe dar o "poder" de consumir dados de qualquer fonte on-line (desde que o servidor alvo também permita). A segunda irá lhe dar a capacidade de checar se o smartphone/tablet do usuário está com acesso a internet no momento (online x offline).

Conectando-se a rede

Após estabelecer as permissões básicas, o próximo passo para seu app é definir quais estratégias e formas de conexão você irá explorar.

Iremos ver nos próximos slides dois métodos, o primeiro mais "clássico" e nativo e o segundo através da lib HTTP do Google chamada Volley.

Por fim, também iremos consumir algumas APIs REST disponíveis gratuitamente na internet ao longo dos exemplos da aula:

https://jsonplaceholder.typicode.com/

https://pokeapi.co/

Não bloqueie a Thread Principal

Algo que vale destacar é que em operações online não devemos nunca ***bloquear a Thread Principal***. Voltando a nossa aula sobre *Threads*, *AsyncTasks e Services* sempre que o app precisar consumir algo que está em um servidor remoto não podemos garantir "agilidade" nem velocidade, pois parte da operação irá depender do estado da conexão do usuário (ex: 3G ruim/lento).

Neste caso a lógica é que na *Thread Principal de UI* nós iremos mostrar algum diálogo/mensagem indicando que algo está sendo "carregado" e em uma thread secundária que irá levar o tempo necessário (pouco ou muito) para processar a requisição on-line iremos aguardar o final da operação. Uma vez que a requisição chegue ao fim, podemos retornar a Main Thread e exibir para nosso usuário os dados que vieram do servidor on-line.

Como vimos em nossas aulas passadas, um bom caminho no consumo de dados "nativo" para não bloquear a Main Thread é o uso de **AsyncTask**.

Consumindo dados on-line

Para fazer nossa primeira requisição apenas com tecnologias disponíveis nativamente na Android SDK devemos iniciar em nossa **Activity** com a criação de uma **AsyncTask** básica.

```kotlin
// kotlin
private inner class NetworkDataTask : AsyncTask<String, Int, String>() {
    override fun onPreExecute() {
      // método para mostramos algum "carregando"
    }

    override fun doInBackground(vararg urls: String): String {
      // método para consumo na rede e requisição
    }

    override fun onPostExecute(result: String) {
      // após os dados terem sidos recebidos do servidor
    }
}
```

Consumindo dados on-line

A ***AsyncTask*** da tela anterior será inserida dentro de uma ***Activity***, o foco neste momento é o método ***doInBackground*** que irá realizar operações em uma segunda Thread sem bloquear a principal. Começaremos criando um objeto do tipo URL que será nosso ponto de partida:

```
// kotlin
val url:URL = URL(...)
```

Depois devemos utilizar a classe nativa ***HttpURLConnection*** para abrir uma conexão do tipo GET com o servidor e iniciar o consumo de dados:

```
// kotlin
val urlConnection:HttpURLConnection = url.openConnection() as HttpURLConnection
urlConnection.setRequestMethod("GET")
urlConnection.connect()
```

Consumindo dados on-line

Após conexão aberta teremos que manter o **buffer** de conexão ativo até todos os dados serem recebidos, teremos um buffer de input (requisição) e um buffer de leitura (resposta), mais ou menos assim:

```
// kotlin
// buffer requisição
val inputStream:InputStream = urlConnection.getInputStream()
// buffer de resposta
val bufferedReader:BufferedReader = BufferedReader(InputStreamReader(inputStream))
// aqui iremos "acumular" o conteúdo (body) de nossa requisição sem os headers de retorno
val bufferData:StringBuffer = StringBuffer()
```

Consumindo dados on-line

Na sequência, devemos trabalhar com a ideia de não sabermos o "tamanho da resposta" podemos ter um retorno de *1 ou milhares de linhas* vindas de nosso servidor, portanto enquanto nosso buffer de leitura (bufferedReader) tiver "dando" retorno, iremos considerar que o conteúdo de nossa requisição (bufferData) ainda não está completo.

A ideia é trabalhar com um *loop* acumulativo:

```
// kotlin
bufferedReader.use {
    var inputLine = it.readLine()
    while (inputLine != null) {
        bufferData.append(inputLine)
        inputLine = it.readLine()
    }
}
```

Consumindo dados on-line

Por fim, uma vez que nosso loop de consumo tenha chegado ao fim, teremos um corpo de retorno completo acumulado em nossa variável *bufferData* sendo que o próximo passo será processar isso e utilizar em nosso app em prol de nosso usuário.

Por exemplo, se a URL que estamos consumindo on-line retornar um **JSON**, podemos transformar nossa string em **JSON** assim:

```
// kotlin
val jsonData = JSONObject(bufferData.toString())
```

Consumindo dados on-line

Aaaa já ia quase me esquecendo, também não podemos esquecer de encerrar nossa conexão com o servidor para liberar memória do app e também desonerar o consumo de dados:

```
// kotlin
if (urlConnection != null) {
    urlConnection.disconnect();
}
```

Consumindo dados on-line

Veremos agora a implementação completa da requisição detalhada nos últimos slides com ***AsyncTask + HttpURLConnection*** em um exemplo demonstrado no *Android Studio*.

Google Volley 🏐

Conhecendo o Volley 🏐

Volley é uma biblioteca HTTP que facilita a criação de redes para apps Android de maneira mais rápida.

O **Volley** se destaca em operações do tipo "chamada de procedimento remoto" (RPC) usadas para preencher uma UI, como buscar uma página de resultados de pesquisa como dados estruturados ou uma lista de items online.

Ele se integra facilmente a qualquer protocolo e vem pronto para uso com strings em texto plano e ***JSON***.

Vale observar que o Volley não é adequado para operações grandes de download ou streaming de dados volumosos, porque ele mantém todas as respostas na memória durante a análise. Para grandes operações de download, o Google indica usar como alternativa a classe ***`DownloadManager`***.

Instalando o Volley 🏐

A maneira mais fácil de adicionar o **Volley** ao seu projeto é adicionar a seguinte dependência ao arquivo *build.gradle* do app:

```
dependencies {
    ...
    implementation 'com.android.volley:volley:1.2.1'
}
```

Lembrando que mesmo usando o **Volley** ainda iremos precisar das permissões de `INTERNET` e `ACCESS_NETWORK_STATE` para que nosso app rode sem problemas.

Requisição Simples com Volley 

A base de uma requisição HTTP feita via **Volley** passará sempre por uma "fila de requisições" (*`RequestQueue`*), na sequência a ideia é adicionar um tipo de requisição a fila e esperar a mesma ser concluída com métodos de resposta padrão e resposta com erro.

```kotlin
// kotlin
// Fila padrão de requisições
val queue = Volley.newRequestQueue(this)
// URL a ser solicitada
val url = "http://www.google.com"

// Solicitação simples (GET), apenas uma string/corpo inteiro
val stringRequest = StringRequest(Request.Method.GET, url,
        Response.Listener<String> { response ->
            // SUCESSO!
        },
        Response.ErrorListener { /* ERRO! */ })

// Adicionando nossa requisição a fila
queue.add(stringRequest)
```

Trabalhando com JSON - Volley 

Para trabalhar com ***JSON*** junto ao **Volley**, mantemos a mesma ideia de fila de requisições, porém ao invés de solicitar uma *StringRequest*, podemos pedir uma **JsonObjectRequest** *(caso o corpo seja um objeto JS complexo)* ou uma **JsonArrayRequest** *(caso o corpo seja um array de items JS)*.

```kotlin
// kotlin
val url = "http://my-json-feed"
val jsonObjectRequest = JsonObjectRequest(Request.Method.GET, url, null,
  Response.Listener { response ->
      // no caso de um JsonObjectRequest a resposta será um JSONObject
      // no caso de um JsonArrayRequest a resposta será um JSONArray
  },
  Response.ErrorListener { error ->
      // TODO: Handle error
  }
)
```

Trabalhando com JSON - Volley 

Também é possível enviarmos dados via **POST** para o servidor/API:

```
// kotlin
val url = "https://jsonplaceholder.typicode.com/posts"

val jsonBody = JSONObject()
jsonBody.put("title", "Android Volley Demo")
jsonBody.put("body", "BNK")
jsonBody.put("userId", "1")

val jsonObjectRequest = JsonObjectRequest(Request.Method.POST, url, jsonBody,
  Response.Listener { response ->
      // no caso de um JsonObjectRequest a resposta será um JSONObject
      // no caso de um JsonArrayRequest a resposta será um JSONArray
  },
  Response.ErrorListener { error ->
      // TODO: Handle error
  }
)
```

Padrão Singleton - Volley 🏐

Se seu app faz uso constante da rede, provavelmente é mais eficiente configurar uma única instância de *RequestQueue* que dure a vida útil do app.

A abordagem recomendada é implementar uma classe de ***singleton*** que encapsule *RequestQueue* e outros recursos do Volley.

Um conceito importante é que o *RequestQueue* precisa ser instanciado com o contexto *Application*, não um contexto *Activity*. Isso garante que a *RequestQueue* tenha uma duração equivalente à vida útil do app, em vez de ser recriada toda vez que a atividade é criada novamente *(por exemplo, quando o usuário gira o dispositivo)*.

Padrão Singleton - Volley

```kotlin
// kotlin
class VolleyQueue constructor(context: Context) {
    companion object {
        @Volatile
        private var INSTANCE: VolleyQueue? = null
        fun getInstance(context: Context) =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: VolleyQueue(context).also {
                    INSTANCE = it
                }
            }
    }
    val requestQueue: RequestQueue by lazy {
        // applicationContext is key, it keeps you from leaking the
        // Activity or BroadcastReceiver if someone passes one in.
        Volley.newRequestQueue(context.applicationContext)
    }
    fun <T> addToRequestQueue(req: Request<T>) {
        requestQueue.add(req)
    }
}
```

Padrão Singleton - Volley 🏐

E depois na hora de usar o padrão estabelecido na classe anterior, podemos chamar isto de qualquer *Activity* ou *Fragment*:

```
// kotlin
// no caso de um Fragment seria getInstance(context) ou getInstance(getActivity())
VolleyQueue.getInstance(this).addToRequestQueue(stringRequest)
```

Brincando com Volley 

Veremos agora a implementação completa das requisições detalhadas baseadas no uso da lib **Volley** em um exemplo demonstrado no *Android Studio*.

obrigado 🚀