Communiquer *via* *Internet*

On s'attend généralement à ce que la plupart des terminaux Android, si ce n'est tous, intègrent un accès à Internet. Cet accès peut passer par le Wifi, les services de données cellulaires (EDGE, 3G, etc.) ou, éventuellement, un mécanisme totalement différent. Quoi qu'il en soit, la plupart des gens – en tout cas, ceux qui ont un accès données ou Wifi – peuvent accéder à Internet à partir de leur téléphone Android.

Il n'est donc pas étonnant qu'Android offre aux développeurs un large éventail de moyens leur permettant d'exploiter cet accès. Ce dernier peut être de haut niveau, comme le navigateur `WebKit` intégré que nous avons étudié au Chapitre 13. Mais il peut également intervenir au niveau le plus bas et utiliser des sockets brutes. Entre ces deux extrémités, il existe des API – disponibles sur le terminal ou *via* des JAR tiers – donnant accès à des protocoles spécifiques comme HTTP, XMPP, SMTP, etc.

Ce livre s'intéresse plutôt aux accès de haut niveau comme le composant `WebKit` et les API Internet car, dans la mesure du possible, les développeurs devraient s'efforcer de réutiliser des composants existants au lieu de créer leurs propres protocoles.

REST et relaxation

Bien qu'Android ne dispose pas d'API cliente pour SOAP ou XML-RPC, il intègre la bibliothèque `HttpComponents` d'Apache. Vous pouvez donc soit ajouter une couche SOAP/XML-RPC au-dessus de cette bibliothèque, soit l'utiliser directement pour accéder aux services web de type REST. Dans ce livre, nous considérerons les "services web REST" comme de simples requêtes HTTP classiques avec des réponses aux formats XML, JSON, etc.

Vous trouverez des didacticiels plus complets, des FAQ et des HOWTO sur le site web de `HttpComponents`[1] : nous ne présenterons ici que les bases en montrant comment consulter les informations météorologiques.

Opérations HTTP *via* `HttpComponents`

Le composant `HttpClient` de `HttpComponents` gère pour vous toutes les requêtes HTTP. La première étape pour l'utiliser consiste évidemment à créer un objet. `HttpClient` étant une interface, vous devrez donc instancier une implémentation de celle-ci, comme `DefaultHttpClient`.

Ces requêtes sont enveloppées dans des instances de `HttpRequest`, chaque commande HTTP étant gérée par une implémentation différente de cette interface (`HttpGet` pour les requêtes GET, par exemple). On crée donc une instance d'une implémentation de `HttpRequest`, on construit l'URL à récupérer ainsi que les autres données de configuration (les valeurs des formulaires si l'on effectue une commande POST *via* `HttpPost`, par exemple) puis l'on passe la méthode au client pour qu'il effectue la requête HTTP en appelant `execute()`.

Ce qui se passe ensuite peut être très simple ou très compliqué. On peut obtenir un objet `HttpResponse` enveloppant un code de réponse (200 pour OK, par exemple), des en-têtes HTTP, etc. Mais on peut également utiliser une variante d'`execute()` qui prend en paramètre un objet `ResponseHandler<String>` : cet appel renverra simplement une représentation `String` de la réponse. En pratique, cette approche est déconseillée car il est préférable de vérifier les codes de réponses HTTP pour détecter les erreurs. Cependant, pour les applications triviales comme les exemples de ce livre, la technique `ResponseHandler<String>` convient parfaitement.

Le projet `Internet/Weather`, par exemple, implémente une activité qui récupère les données météorologiques de votre emplacement actuel à partir du site Google Weather. Ces données sont converties en HTML puis passées à un widget `WebKit` qui se charge de les afficher. Nous laissons en exercice au lecteur la réécriture de ce programme pour qu'il utilise un `ListView`. En outre, cet exemple étant relativement long, nous ne présenterons

1. http://hc.apache.org/.

ici que les extraits de code en rapport avec ce chapitre ; les sources complets sont disponibles dans les exemples fournis avec ce livre[1].

Pour rendre tout ceci un peu plus intéressant, nous utilisons les services de localisation d'Android pour déterminer notre emplacement actuel. Les détails de fonctionnement de ce service sont décrits au Chapitre 33.

Lorsqu'un emplacement a été trouvé – soit au lancement, soit parce que nous avons bougé –, nous récupérons les données de Google Weather *via* la méthode `updateForecast()` :

```
private void updateForecast(Location loc) {
  String url = String.format(format, ""
        + (int) (loc.getLatitude() * 1000000), ""
        + (int) (loc.getLongitude() * 1000000));
    HttpGet getMethod = new HttpGet(url);
    try {
      ResponseHandler<String> responseHandler =
            new BasicResponseHandler();
      String responseBody = client.execute(getMethod,
                                    responseHandler);
      buildForecasts(responseBody);
      String page = generatePage();
      browser.loadDataWithBaseURL(null, page, "text/html",
                                  "UTF-8", null);
    } catch (Throwable t) {
      Toast.makeText(this, "La requete a echouee: " +
                           t.toString(), 4000).show();
    }
  }
}
```

La méthode `updateForecast()` prend un objet `Location` en paramètre, obtenu *via* le processus de mise à jour de la localisation. Pour l'instant, il suffit de savoir que `Location` dispose des méthodes `getLatitude()` et `getLongitude()`, qui renvoient, respectivement, la latitude et la longitude.

L'URL Google Weather est stockée dans une ressource chaîne à laquelle nous ajoutons en cours d'exécution la latitude et la longitude. Nous construisons un objet `HttpGet` avec cette URL (l'objet `HttpClient` a été créé dans `onCreate()`) puis nous exécutons cette méthode. À partir de la réponse XML, nous construisons la page HTML des prévisions que nous transmettons au widget `WebKit`. Si l'objet `HttpClient` échoue avec une exception, nous indiquons l'erreur à l'aide d'un toast.

1. Reportez-vous à la page dédiée à cet ouvrage sur le site www.pearson.fr.

Traitement des réponses

La réponse que l'on obtient est dans un certain format – HTML, XML, JSON, etc. – et c'est à nous, bien sûr, de choisir l'information qui nous intéresse pour en tirer quelque chose d'utile. Dans le cas de WeatherDemo, nous voulons extraire l'heure de la prévision, la température et l'icône (qui représente les conditions météorologiques) afin de nous en servir pour produire une page HTML.

Android fournit :

- trois analyseurs XML, l'analyseur DOM classique du W3C (`org.w3c.dom`), un analyseur SAX (`org.xml.sax`) et l'analyseur pull présenté au Chapitre 19 ;
- un analyseur JSON (`org.json`).

Lorsque cela est possible, vous pouvez bien sûr utiliser du code Java tiers pour prendre en charge d'autres formats – un analyseur RSS/Atom, par exemple. L'utilisation du code tiers a été décrite au Chapitre 21.

Pour WeatherDemo, nous utilisons l'analyseur DOM du W3C dans notre méthode `buildForecasts()` :

```
void buildForecasts(String raw) throws Exception {
    DocumentBuilder builder =
      DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document doc =
      builder.parse(new InputSource(new StringReader(raw)));
    NodeList forecastList =
      doc.getElementsByTagName("forecast_conditions");
    for (int i = 0; i < forecastList.getLength(); i++) {
      Element currentFore = (Element) forecastList.item(i);

      // Retrouvons le jour de la semaine
      String day =currentFore.getElementsByTagName("day_of_week")
                             .item(0).getAttributes()
                             .item(0).getNodeValue();
      String lowTemp = currentFore.getElementsByTagName("low")
                             .item(0).getAttributes()
                             .item(0).getNodeValue();
      String highTemp = currentFore.getElementsByTagName("high")
                              .item(0).getAttributes()
                              .item(0).getNodeValue();
      String icon = currentFore.getElementsByTagName("icon")
                             .item(0).getAttributes()
                             .item(0).getNodeValue();
      Forecast f = new Forecast();
      f.setDay(day);
      f.setLowTemp(lowTemp);
      f.setHighTemp(highTemp);
      f.setIcon(icon);
      forecasts.add(f);
    }
  }
```

Le code HTML est lu comme un `InputStream` et est fourni à l'analyseur DOM. Puis on recherche les éléments `forecast_conditions` et l'on remplit un ensemble de modèles `Forecast` en incluant la date, la température et l'URL de l'icône qui sera affichée en fonction du temps.

La méthode `generatePage()` produit à son tour un tableau HTML rudimentaire contenant les prévisions :

```
String generatePage() {
  StringBuffer bufResult = new StringBuffer("<html><body><table>");

  bufResult.append("<tr><th width=\"50%\">Jour</th>"
    +  "<th>Basse</th><th>Haute</th><th>Tendance</th></tr>");

  for (Forecast forecast : forecasts) {
    bufResult.append("<tr><td align=\"center\">");
    bufResult.append(forecast.getDay());
    bufResult.append("</td><td align=\"center\">");
    bufResult.append(forecast.getLowTemp());
    bufResult.append("</td>");
    bufResult.append("</td><td align=\"center\">");
    bufResult.append(forecast.getHighTemp());
    bufResult.append("</td><td><img src=\"");
    bufResult.append(forecast.getIcon());
    bufResult.append("\"></td></tr>");
  }
  bufResult.append("</table></body></html>");
  return (bufResult.toString());
}
```

La Figure 22.1 montre le résultat obtenu.

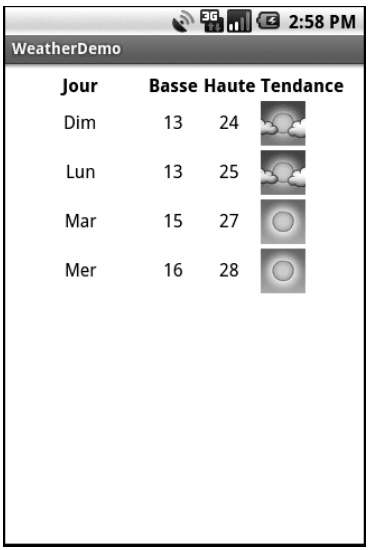

Figure 22.1
L'application `WeatherDemo`.

Autres points importants

Si vous devez utiliser SSL, n'oubliez pas que la configuration de `HttpClient` n'inclut pas SSL par défaut car c'est à vous de décider comment gérer la présentation des certificats SSL – les acceptez-vous tous aveuglément, même ceux qui sont autosignés ou qui ont expiré ? Préférez-vous demander confirmation à l'utilisateur avant d'accepter les certificats un peu étranges ?

De même, `HttpClient` est conçu par défaut pour être utilisé dans une application monothread, bien qu'il puisse aisément être configuré pour travailler dans un contexte multithread.

Pour tous ces types de problèmes, la meilleure solution consiste à consulter la documentation et le support disponibles sur le site web de `HttpComponents`.

Partie IV

Intentions (Intents)

23

Création de filtres d'intentions

Pour l'instant, nous ne nous sommes intéressés qu'aux activités ouvertes directement par l'utilisateur à partir du lanceur du terminal, ce qui est, évidemment, le moyen le plus évident de lancer une activité et de la rendre disponible à l'utilisateur. Dans la plupart des cas, c'est de cette façon que l'utilisateur commencera à utiliser votre application.

Dans de nombreuses situations, le système Android repose sur un grand nombre de composants étroitement liés. Ce que vous pouvez obtenir dans une interface graphique *via* des boîtes de dialogue, des fenêtres filles, etc. est généralement traité par des activités indépendantes. Bien que l'une d'elles puisse être "spécifique" puisqu'elle apparaît dans le lanceur, les autres doivent toutes être accessibles... d'une façon ou d'une autre.

Elles le sont grâce aux intentions.

Une intention est essentiellement un message que l'on passe à Android pour lui dire "je veux que tu fasses... quelque chose". Ce "quelque chose" dépend de la situation – parfois, on sait parfaitement de quoi il s'agit (ouvrir l'une de nos autres activités, par exemple) mais, d'autres fois, on ne le sait pas.

Dans l'absolu, Android ne se consacre qu'aux intentions et à leurs récepteurs. Maintenant que nous avons vu comment créer des activités, plongeons-nous dans les intentions afin de pouvoir créer des applications plus complexes tout en étant de "bons citoyens d'Android".

Quelle est votre intention ?

Lorsque sir Tim Berners-Lee a conçu le protocole de transfert hypertexte, HTTP, il a défini un ensemble de verbes et d'adresses sous la forme d'URL. Une adresse désigne une ressource : une page web, une image ou un programme qui s'exécute sur un serveur, par exemple. Un verbe précise l'action qui doit s'appliquer à cette adresse : GET pour la récupérer, POST pour lui envoyer des données de formulaire afin qu'elle les traite, etc.

Les intentions sont similaires car elles représentent une action et un contexte. Bien qu'elles permettent de définir plus d'actions et de composants de contexte qu'il n'y a de verbes et de ressources HTTP, le concept est le même.

Tout comme un navigateur web sait comment traiter une paire verbe + URL, Android sait comment trouver les activités ou les autres applications qui sauront gérer une intention donnée.

Composantes des intentions

Les deux parties les plus importantes d'une intention sont l'action et ce qu'Android appelle les "données". Elles sont quasiment analogues aux verbes et aux URL de HTTP – l'action est le verbe et les "données" sont une `Uri` comme `content://contacts/people/1`, représentant un contact dans la base de données des contacts. Les actions sont des constantes, comme `ACTION_VIEW` (pour afficher la ressource), `ACTION_EDIT` (pour l'éditer) ou `ACTION_PICK` (pour choisir un élément disponible dans une `Uri` représentant une collection, comme `content://contacts/people`).

Si vous créez une intention combinant `ACTION_VIEW` avec l'`Uri content://contacts/people/1` et que vous la passiez à Android, ce dernier saura comment trouver et ouvrir une activité capable d'afficher cette ressource.

Outre l'action et l'`Uri` des "données", vous pouvez placer d'autres critères dans une intention (qui est représentée par un objet `Intent`) :

- Une catégorie. Votre activité "principale" appartient à la catégorie `LAUNCHER`, pour indiquer qu'elle apparaît dans le menu du lanceur. Les autres activités appartiendront probablement aux catégories `DEFAULT` ou `ALTERNATIVE`.
- Un type MIME indiquant le type de ressource sur laquelle vous voulez travailler si vous ne connaissez pas une `Uri` collection.

- Un composant, c'est-à-dire la classe de l'activité supposée recevoir cette intention. Cette utilisation des composants évite d'avoir besoin des autres propriétés de l'intention, mais elle rend cette dernière plus fragile car elle suppose des implémentations spécifiques.
- Des "Extras", c'est-à-dire un `Bundle` d'autres informations que vous voulez passer au récepteur en même temps que l'intention et dont ce dernier pourra tirer parti. Les informations utilisables par un récepteur donné dépendent du récepteur et sont (heureusement) bien documentées.

La documentation d'Android consacrée à la classe `Intent` contient les listes des actions et des catégories standard.

Routage des intentions

Comme on l'a mentionné précédemment, si le composant cible a été précisé dans l'intention, Android n'aura aucun doute sur sa destination – il lancera l'activité en question. Ce mécanisme peut convenir si l'intention cible se trouve dans votre application, mais n'est vraiment pas recommandé pour envoyer des intentions à d'autres applications car les noms des composants sont globalement considérés comme privés à l'application et peuvent donc être modifiés. Il est préférable d'utiliser les modèles d'`Uri` et les types MIME pour identifier les services auxquels vous souhaitez accéder.

Si vous ne précisez pas de composant cible, Android devra trouver les activités (ou les autres récepteurs d'intentions) éligibles pour cette intention. Vous aurez remarqué que nous avons mis "activités" au pluriel car une activité peut très bien se résoudre en plusieurs activités. Cette approche du routage est préférable au routage implicite.

Essentiellement, trois conditions doivent être vérifiées pour qu'une activité soit éligible pour une intention donnée :

1. L'activité doit supporter l'action indiquée.
2. L'activité doit supporter le type MIME indiqué (s'il a été fourni).
3. L'activité doit supporter toutes les catégories nommées dans l'intention.

La conclusion est que vous avez intérêt à ce que vos intentions soient suffisamment spécifiques pour trouver le ou les bons récepteurs, mais pas plus.

Tout ceci deviendra plus clair à mesure que nous étudierons quelques exemples.

Déclarer vos intentions

Tous les composants Android qui souhaitent être prévenus par des intentions doivent déclarer des filtres d'intention afin qu'Android sache quelles intentions devraient aller vers quel composant. Pour ce faire, vous devez ajouter des éléments `intent-filter` au fichier `AndroidManifest.xml`.

Le script de création des applications Android (`activityCreator` ou son équivalent IDE) fournit des filtres d'intention à tous les projets. Ces déclarations sont de la forme :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.android.skeleton">
    <application>
        <activity android:name=".Now" android:label="Now">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Notez la présence de l'élément `intent-filter` sous l'élément `activity`. Il annonceles choses suivantes :

- Cette activité est l'activité principale de cette application.
- Elle appartient à la catégorie `LAUNCHER`, ce qui signifie qu'elle aura une icône dans le menu principal d'Android.

Cette activité étant l'activité principale de l'application, Android sait qu'elle est le composant qu'il doit lancer lorsqu'un utilisateur choisit cette application à partir du menu principal.

Vous pouvez indiquer plusieurs actions ou catégories dans vos filtres d'intention afin de préciser que le composant associé (l'activité) gère plusieurs sortes d'intentions différentes.

Il est fort probable que vous voudrez également que vos activités secondaires (non `MAIN`) précisent le type MIME des données qu'elles manipulent. Ainsi, si une intention est destinée à ce type MIME – directement ou indirectement *via* une `Uri` référençant une ressource de ce type –, Android saura que le composant sait gérer ces données.

Vous pourriez, par exemple, déclarer une activité de la façon suivante :

```
<activity android:name=".TourViewActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.commonsware.tour" />
  </intent-filter>
</activity>
```

Celle-ci sera alors lancée par une intention demandant l'affichage d'une `Uri` représentant un contenu `vnd.android.cursor.item/vnd.commonsware.tour`. Cette intention pourrait provenir d'une autre activité de la même application (l'activité principale, par exemple) ou d'une autre application qui connaît une `Uri` que cette activité peut gérer.

Récepteurs d'intention

Dans les exemples que nous venons de voir, les filtres d'intention étaient configurés sur des activités. Cependant, lier les intentions à des activités n'est parfois pas exactement ce dont on a besoin :

- Certains événements système peuvent nous obliger à déclencher une opération dans un service plutôt qu'une activité.
- Certains événements peuvent devoir lancer des activités différentes en fonction des circonstances, où le critère repose non pas uniquement sur l'intention elle-même, mais sur un autre état (si l'on obtient l'intention *X* et que la base de données contienne *Y*, on lance l'activité *M* ; si la base ne contient pas *Y*, on lance l'activité *N*, par exemple).

Dans ces situations, Android offre un récepteur d'intention défini comme une classe qui implémente l'interface `BroadcastReceiver`. Les récepteurs d'intention sont des objets conçus pour recevoir des intentions – notamment celles qui sont diffusées – et pour effectuer une action impliquant généralement le lancement d'autres intentions pour déclencher une opération dans une activité, un service ou un autre composant.

L'interface `BroadcastReceiver` ne possède qu'une seule méthode `onReceive()`, que les récepteurs d'intention doivent donc implémenter pour y effectuer les traitements qu'ils souhaitent en cas de réception d'une intention. Pour déclarer un récepteur d'intention, il suffit d'ajouter un élément `receiver` au fichier `AndroidManifest.xml` :

```
<receiver android:name=".MaClasseReceptriceDIntention/>
```

Un récepteur d'intention ne vit que le temps de traiter `onReceive()` – lorsque cette méthode se termine, l'instance est susceptible d'être supprimée par le ramasse-miettes et ne sera pas réutilisée. Ceci signifie donc que les fonctionnalités de ces récepteurs sont un peu limitées, essentiellement pour éviter l'appel de fonctions de rappel. Ils ne peuvent notamment pas être liés à un service ni ouvrir une boîte de dialogue.

La seule exception est lorsque le `BroadcastReceiver` est implémenté sur un composant qui a une durée de vie assez longue, comme une activité ou un service : dans ce cas, le récepteur vivra aussi longtemps que son "hôte" (jusqu'à ce que l'activité soit stoppée, par exemple). Cependant, dans cette situation, vous ne pouvez pas déclarer le récepteur dans `AndroidManifest.xml` : il faut appeler `registerReceiver()` dans la méthode `onResume()` de l'activité pour annoncer son intérêt pour une intention, puis appeler `unregisterReceiver()` dans sa méthode `onPause()` lorsque vous n'avez plus besoin de ces intentions.

Attention à la pause

Il y a un petit problème lorsque l'on utilise des objets `Intent` pour transmettre des messages : ceci ne fonctionne que lorsque le récepteur est actif. Voici ce que précise la documentation de `BroadcastReceiver` à ce sujet :

Si vous enregistrez un récepteur dans votre implémentation d'`Activity.onResume()`, il faut le désinscrire dans `Activity.onPause()` (vous ne recevrez pas d'intention pendant la pause et cela évite une surcharge inutile du système). N'effectuez pas cette désinscription dans `Activity.onSaveInstanceState()`, car cette méthode n'est pas appelée lorsque l'utilisateur revient dans son historique.

Vous pouvez donc utiliser les intentions pour transmettre des messages aux condition suivantes :

- Votre récepteur ne se soucie pas de manquer des messages lorsqu'il est inactif.
- Vous fournissez un moyen pour que le récepteur récupère les messages qu'il a manqués pendant qu'il était inactif.

Aux Chapitres 30 et 31, nous verrons un exemple de la première condition, où le récepteur (le client du service) utilise des messages reposant sur des intentions lorsqu'elles sont disponibles, mais pas quand le client n'est pas actif.