android developers

Intents and Intent Filters

In this document
Intent Objects
Intent Resolution
  Intent filters
  Common cases
  Using intent matching
Note Pad Example

Key classes
`Intent`
`IntentFilter`
`BroadcastReceiver`
`PackageManager`

Three of the core components of an application — activities, services, and broadcast receivers — are activated through messages, called *intents*. Intent messaging is a facility for late run-time binding between components in the same or different applications. The intent itself, an `Intent` object, is a passive data structure holding an abstract description of an operation to be performed — or, often in the case of broadcasts, a description of something that has happened and is being announced. There are separate mechanisms for delivering intents to each type of component:

- An Intent object is passed to `Context.startActivity()` or `Activity.startActivityForResult()` to launch an activity or get an existing activity to do something new. (It can also be passed to `Activity.setResult()` to return information to the activity that called `startActivityForResult()`.)
- An Intent object is passed to `Context.startService()` to initiate a service or deliver new instructions to an ongoing service. Similarly, an intent can be passed to `Context.bindService()` to establish a connection between the calling component and a target service. It can optionally initiate the service if it's not already running.
- Intent objects passed to any of the broadcast methods (such as `Context.sendBroadcast()`, `Context.sendOrderedBroadcast()`, or `Context.sendStickyBroadcast()`) are delivered to all interested broadcast receivers. Many kinds of broadcasts originate in system code.

In each case, the Android system finds the appropriate activity, service, or set of broadcast receivers to respond to the intent, instantiating them if necessary. There is no overlap within these messaging systems: Broadcast intents are delivered only to broadcast receivers, never to activities or services. An intent passed to `startActivity()` is delivered only to an activity, never to a service or broadcast receiver, and so on.

This document begins with a description of Intent objects. It then describes the rules Android uses to map intents to components — how it resolves which component should receive an intent message. For intents that don't explicitly name a target component, this process involves testing the Intent object against *intent filters* associated with potential targets.

Intent Objects

An `Intent` object is a bundle of information. It contains information of interest to the component that receives the intent (such as the action to be taken and the data to act on) plus information of interest to the Android system (such as the category of component that should handle the intent and instructions on how to launch a target activity). Principally, it can contain the following:

Component name

The name of the component that should handle the intent. This field is a `ComponentName` object — a combination of the fully qualified class name of the target component (for example "`com.example.project.app.FreneticActivity`") and the package name set in the manifest file of the application where the component resides (for example, "`com.example.project`"). The package part of the component name and the package name set in the manifest do not necessarily have to match.

The component name is optional. If it is set, the Intent object is delivered to an instance of the designated class. If it is not set, Android uses other information in the Intent object to locate a suitable target — see Intent Resolution, later in this document.

The component name is set by setComponent(), setClass(), or setClassName() and read by getComponent().

Action

A string naming the action to be performed — or, in the case of broadcast intents, the action that took place and is being reported. The Intent class defines a number of action constants, including these:

Constant	Target component	Action
ACTION_CALL	activity	Initiate a phone call.
ACTION_EDIT	activity	Display data for the user to edit.
ACTION_MAIN	activity	Start up as the initial activity of a task, with no data input and no returned output.
ACTION_SYNC	activity	Synchronize data on a server with data on the mobile device.
ACTION_BATTERY_LOW	broadcast receiver	A warning that the battery is low.
ACTION_HEADSET_PLUG	broadcast receiver	A headset has been plugged into the device, or unplugged from it.
ACTION_SCREEN_ON	broadcast receiver	The screen has been turned on.
ACTION_TIMEZONE_CHANGED	broadcast receiver	The setting for the time zone has changed.

See the Intent class description for a list of pre-defined constants for generic actions. Other actions are defined elsewhere in the Android API. You can also define your own action strings for activating the components in your application. Those you invent should include the application package as a prefix — for example: "com.example.project.SHOW_COLOR".

The action largely determines how the rest of the intent is structured — particularly the data and extras fields — much as a method name determines a set of arguments and a return value. For this reason, it's a good idea to use action names that are as specific as possible, and to couple them tightly to the other fields of the intent. In other words, instead of defining an action in isolation, define an entire protocol for the Intent objects your components can handle.

The action in an Intent object is set by the setAction() method and read by getAction().

Data

The URI of the data to be acted on and the MIME type of that data. Different actions are paired with different kinds of data specifications. For example, if the action field is ACTION_EDIT, the data field would contain the URI of the document to be displayed for editing. If the action is ACTION_CALL, the data field would be a tel: URI with the number to call. Similarly, if the action is ACTION_VIEW and the data field is an http: URI, the receiving activity would be called upon to download and display whatever data the URI refers to.

When matching an intent to a component that is capable of handling the data, it's often important to know the type of data (its MIME type) in addition to its URI. For example, a component able to display image data should not be called upon to play an audio file.

In many cases, the data type can be inferred from the URI — particularly content: URIs, which indicate that the data is located on the device and controlled by a content provider (see the separate discussion on content providers). But the type can also be explicitly set in the Intent object. The setData() method specifies data only as a URI, setType() specifies it only as a MIME type, and setDataAndType() specifies it as both a URI and a MIME type. The URI is read by getData() and the type by getType().

Category

A string containing additional information about the kind of component that should handle the intent. Any number of category descriptions can be placed in an Intent object. As it does for actions, the Intent class defines several category constants, including these:

Constant	Meaning
`CATEGORY_BROWSABLE`	The target activity can be safely invoked by the browser to display data referenced by a link — for example, an image or an e-mail message.
`CATEGORY_GADGET`	The activity can be embedded inside of another activity that hosts gadgets.
`CATEGORY_HOME`	The activity displays the home screen, the first screen the user sees when the device is turned on or when the HOME key is pressed.
`CATEGORY_LAUNCHER`	The activity can be the initial activity of a task and is listed in the top-level application launcher.
`CATEGORY_PREFERENCE`	The target activity is a preference panel.

See the `Intent` class description for the full list of categories.

The `addCategory()` method places a category in an Intent object, `removeCategory()` deletes a category previously added, and `getCategories()` gets the set of all categories currently in the object.

Extras

Key-value pairs for additional information that should be delivered to the component handling the intent. Just as some actions are paired with particular kinds of data URIs, some are paired with particular extras. For example, an `ACTION_TIMEZONE_CHANGED` intent has a "`time-zone`" extra that identifies the new time zone, and `ACTION_HEADSET_PLUG` has a "`state`" extra indicating whether the headset is now plugged in or unplugged, as well as a "`name`" extra for the type of headset. If you were to invent a `SHOW_COLOR` action, the color value would be set in an extra key-value pair.

The Intent object has a series of `put...()` methods for inserting various types of extra data and a similar set of `get...()` methods for reading the data. These methods parallel those for `Bundle` objects. In fact, the extras can be installed and read as a Bundle using the `putExtras()` and `getExtras()` methods.

Flags

Flags of various sorts. Many instruct the Android system how to launch an activity (for example, which task the activity should belong to) and how to treat it after it's launched (for example, whether it belongs in the list of recent activities). All these flags are defined in the Intent class.

The Android system and the applications that come with the platform employ Intent objects both to send out system-originated broadcasts and to activate system-defined components. To see how to structure an intent to activate a system component, consult the list of intents in the reference.

Intent Resolution

Intents can be divided into two groups:

- *Explicit intents* designate the target component by its name (the component name field, mentioned earlier, has a value set). Since component names would generally not be known to developers of other applications, explicit intents are typically used for application-internal messages — such as an activity starting a subordinate service or launching a sister activity.
- *Implicit intents* do not name a target (the field for the component name is blank). Implicit intents are often used to activate components in other applications.

Android delivers an explicit intent to an instance of the designated target class. Nothing in the Intent object other than the component name matters for determining which component should get the intent.

A different strategy is needed for implicit intents. In the absence of a designated target, the Android system must find the best component (or components) to handle the intent — a single activity or service to perform the requested action or the set of broadcast receivers to respond to the broadcast announcement. It does so by comparing the contents of the Intent object to *intent filters*, structures associated with components that can potentially receive intents. Filters advertise the capabilities of a component and delimit the intents it can handle. They open the component to the possibility of receiving implicit intents of the advertised type. If a component does not have any intent filters, it can receive only explicit intents. A component with filters can receive both explicit and implicit intents.

Only three aspects of an Intent object are consulted when the object is tested against an intent filter:

action
data (both URI and data type)
category

The extras and flags play no part in resolving which component receives an intent.

Intent filters

To inform the system which implicit intents they can handle, activities, services, and broadcast receivers can have one or more intent filters. Each filter describes a capability of the component, a set of intents that the component is willing to receive. It, in effect, filters in intents of a desired type, while filtering out unwanted intents — but only unwanted implicit intents (those that don't name a target class). An explicit intent is always delivered to its target, no matter what it contains; the filter is not consulted. But an implicit intent is delivered to a component only if it can pass through one of the component's filters.

A component has separate filters for each job it can do, each face it can present to the user. For example, the NoteEditor activity of the sample Note Pad application has two filters — one for starting up with a specific note that the user can view or edit, and another for starting with a new, blank note that the user can fill in and save. (All of Note Pad's filters are described in the Note Pad Example section, later.)

An intent filter is an instance of the `IntentFilter` class. However, since the Android system must know about the capabilities of a component before it can launch that component, intent filters are generally not set up in Java code, but in the application's manifest file (AndroidManifest.xml) as `<intent-filter>` elements. (The one exception would be filters for broadcast receivers that are registered dynamically by calling `Context.registerReceiver()`; they are directly created as IntentFilter objects.)

Filters and security

An intent filter cannot be relied on for security. While it opens a component to receiving only certain kinds of implicit intents, it does nothing to prevent explicit intents from targeting the component. Even though a filter restricts the intents a component will be asked to handle to certain actions and data sources, someone could always put together an explicit intent with a different action and data source, and name the component as the target.

A filter has fields that parallel the action, data, and category fields of an Intent object. An implicit intent is tested against the filter in all three areas. To be delivered to the component that owns the filter, it must pass all three tests. If it fails even one of them, the Android system won't deliver it to the component — at least not on the basis of that filter. However, since a component can have multiple intent filters, an intent that does not pass through one of a component's filters might make it through on another.

Each of the three tests is described in detail below:

Action test

An `<intent-filter>` element in the manifest file lists actions as `<action>` subelements. For example:

```
<intent-filter . . . >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    . . .
</intent-filter>
```

As the example shows, while an Intent object names just a single action, a filter may list more than one. The list cannot be empty; a filter must contain at least one `<action>` element, or it will block all intents.

To pass this test, the action specified in the Intent object must match one of the actions listed in the filter. If the object or the filter does not specify an action, the results are as follows:

- If the filter fails to list any actions, there is nothing for an intent to match, so all intents fail the test. No intents can get through the filter.
- On the other hand, an Intent object that doesn't specify an action automatically passes the test — as long as the filter contains at least one action.

Category test

An `<intent-filter>` element also lists categories as subelements. For example:

```
<intent-filter . . . >
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    . . .
</intent-filter>
```

Note that the constants described earlier for actions and categories are not used in the manifest file. The full string values are used instead. For instance, the "`android.intent.category.BROWSABLE`" string in the example above corresponds to the `CATEGORY_BROWSABLE` constant mentioned earlier in this document. Similarly, the string "`android.intent.action.EDIT`" corresponds to the `ACTION_EDIT` constant.

For an intent to pass the category test, every category in the Intent object must match a category in the filter. The filter can list additional categories, but it cannot omit any that are in the intent.

In principle, therefore, an Intent object with no categories should always pass this test, regardless of what's in the filter. That's mostly true. However, with one exception, Android treats all implicit intents passed to `startActivity()` as if they contained at least one category: "`android.intent.category.DEFAULT`" (the `CATEGORY_DEFAULT` constant). Therefore, activities that are willing to receive implicit intents must include "`android.intent.category.DEFAULT`" in their intent filters. (Filters with "`android.intent.action.MAIN`" and "`android.intent.category.LAUNCHER`" settings are the exception. They mark activities that begin new tasks and that are represented on the launcher screen. They can include "`android.intent.category.DEFAULT`" in the list of categories, but don't need to.) See Using intent matching, later, for more on these filters.)

Data test

Like the action and categories, the data specification for an intent filter is contained in a subelement. And, as in those cases, the subelement can appear multiple times, or not at all. For example:

```
<intent-filter . . . >
    <data android:mimeType="video/mpeg" android:scheme="http" . . . />
    <data android:mimeType="audio/mpeg" android:scheme="http" . . . />
    . . .
</intent-filter>
```

Each `<data>` element can specify a URI and a data type (MIME media type). There are separate attributes — `scheme`, `host`, `port`, and `path` — for each part of the URI:

```
scheme://host:port/path
```

For example, in the following URI,

```
content://com.example.project:200/folder/subfolder/etc
```

the scheme is "`content`", the host is "`com.example.project`", the port is "`200`", and the path is "`folder/subfolder/etc`". The host and port together constitute the URI *authority*; if a host is not specified, the port is ignored.

Each of these attributes is optional, but they are not independent of each other: For an authority to be meaningful, a scheme must also be specified. For a path to be meaningful, both a scheme and an authority must be specified.

When the URI in an Intent object is compared to a URI specification in a filter, it's compared only to the parts of the URI actually mentioned in the filter. For example, if a filter specifies only a scheme, all URIs with that scheme match the filter. If a filter specifies a scheme and an authority but no path, all URIs with the same scheme and authority match, regardless of their paths. If a filter specifies a scheme, an authority, and a path, only URIs with the same scheme, authority, and path match. However, a path specification in the filter can contain wildcards to require only a partial match of the path.

The `type` attribute of a `<data>` element specifies the MIME type of the data. It's more common in filters than a URI. Both the Intent object and the filter can use a "*" wildcard for the subtype field — for example, "`text/*`" or "`audio/*`" — indicating any subtype matches.

The data test compares both the URI and the data type in the Intent object to a URI and data type specified in the filter. The rules are as follows:

a. An Intent object that contains neither a URI nor a data type passes the test only if the filter likewise does not specify any URIs or data types.

b. An Intent object that contains a URI but no data type (and a type cannot be inferred from the URI) passes the test only if its URI matches a URI in the filter and the filter likewise does not specify a type. This will be the case only for URIs like `mailto:` and `tel:` that do not refer to actual data.

c. An Intent object that contains a data type but not a URI passes the test only if the filter lists the same data type and similarly does not specify a URI.

d. An Intent object that contains both a URI and a data type (or a data type can be inferred from the URI) passes the data type part of the test only if its type matches a type listed in the filter. It passes the URI part of the test either if its URI matches a URI in the filter or if it has a `content:` or `file:` URI and the filter does not specify a URI. In other words, a component is presumed to support `content:` and `file:` data if its filter lists only a data type.

If an intent can pass through the filters of more than one activity or service, the user may be asked which component to activate. An exception is raised if no target can be found.

Common cases

The last rule shown above for the data test, rule (d), reflects the expectation that components are able to get local data from a file or content provider. Therefore, their filters can list just a data type and do not need to explicitly name the `content:` and `file:` schemes. This is a typical case. A `<data>` element like the following, for example, tells Android that the component can get image data from a content provider and display it:

```
<data android:mimeType="image/*" />
```

Since most available data is dispensed by content providers, filters that specify a data type but not a URI are perhaps the most common.

Another common configuration is filters with a scheme and a data type. For example, a `<data>` element like the following tells Android that the component can get video data from the network and display it:

```
<data android:scheme="http" android:type="video/*" />
```

Consider, for example, what the browser application does when the user follows a link on a web page. It first tries to display the data (as it could if the link was to an HTML page). If it can't display the data, it puts together an implicit intent with the scheme and data type and tries to start an activity that can do the job. If there are no takers, it asks the download manager to download the data. That puts it under the control of a content provider, so a potentially larger pool of activities (those with filters that just name a data type) can respond.

Most applications also have a way to start fresh, without a reference to any particular data. Activities that can initiate applications have filters with "`android.intent.action.MAIN`" specified as the action. If they are to be represented in the application launcher, they also specify the "`android.intent.category.LAUNCHER`" category:

```
<intent-filter . . . >
    <action android:name="code android.intent.action.MAIN" />
    <category android:name="code android.intent.category.LAUNCHER" />
</intent-filter>
```

Using intent matching

Intents are matched against intent filters not only to discover a target component to activate, but also to discover something about the set of components on the device. For example, the Android system populates the application launcher, the top-level screen that shows the applications that are available for the user to launch, by finding all the activities with intent filters that specify the "`android.intent.action.MAIN`" action and "`android.intent.category.LAUNCHER`" category (as illustrated in the previous section). It then displays the icons and labels of those activities in the launcher. Similarly, it discovers the home screen by looking for the activity with "`android.intent.category.HOME`" in its filter.

Your application can use intent matching is a similar way. The `PackageManager` has a set of `query...()` methods that return all components that can accept a particular intent, and a similar series of `resolve...()` methods that determine the best component to respond to an intent. For example, `queryIntentActivities()` returns a list of all activities that can perform the intent passed as an argument, and `queryIntentServices()` returns a similar list of services. Neither method activates the components; they just list the ones that can respond. There's a similar method, `queryBroadcastReceivers()`, for broadcast receivers.

Note Pad Example

The Note Pad sample application enables users to browse through a list of notes, view details about individual items in the list, edit the items, and add a new item to the list. This section looks at the intent filters declared in its manifest file. (If you're working offline in the SDK, you can find all the source files for this sample application, including its manifest file, at `<sdk>/samples/NotePad/index.html`. If you're viewing the documentation online, the source files are in the Tutorials and Sample Code section here.)

In its manifest file, the Note Pad application declares three activities, each with at least one intent filter. It also declares a content provider that manages the note data. Here is the manifest file in its entirety:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.example.android.notepad">
    <application android:icon="@drawable/app_notes"
                 android:label="@string/app_name" >

        <provider android:name="NotePadProvider"
                  android:authorities="com.google.provider.NotePad" />

        <activity android:name="NotesList"
android:label="@string/title_notes_list">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.GET_CONTENT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.google.note"
/>
            </intent-filter>
        </activity>

        <activity android:name="NoteEditor"
                  android:theme="@android:style/Theme.Light"
                  android:label="@string/title_note" >
            <intent-filter android:label="@string/resolve_edit">
                <action android:name="android.intent.action.VIEW" />
                <action android:name="android.intent.action.EDIT" />
                <action android:name="com.android.notepad.action.EDIT_NOTE" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.item/vnd.google.note"
/>
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.INSERT" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
            </intent-filter>
        </activity>

        <activity android:name="TitleEditor"
                  android:label="@string/title_edit_title"
                  android:theme="@android:style/Theme.Dialog">
            <intent-filter android:label="@string/resolve_title">
                <action android:name="com.android.notepad.action.EDIT_TITLE" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.ALTERNATIVE" />
                <category
android:name="android.intent.category.SELECTED_ALTERNATIVE" />
                <data android:mimeType="vnd.android.cursor.item/vnd.google.note"
/>
            </intent-filter>
        </activity>

    </application>
</manifest>
```

The first activity, NotesList, is distinguished from the other activities by the fact that it operates on a directory of notes (the note list) rather than on a single note. It would generally serve as the initial user interface into the application. It can do three things as described by its three intent filters:

1.
```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

   This filter declares the main entry point into the Note Pad application. The standard `MAIN` action is an entry point that does not require any other information in the Intent (no data specification, for example), and the `LAUNCHER` category says that this entry point should be listed in the application launcher.

2.
```
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="android.intent.action.PICK" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

   This filter declares the things that the activity can do on a directory of notes. It can allow the user to view or edit the directory (via the `VIEW` and `EDIT` actions), or to pick a particular note from the directory (via the `PICK` action).

   The `mimeType` attribute of the `<data>` element specifies the kind of data that these actions operate on. It indicates that the activity can get a Cursor over zero or more items (`vnd.android.cursor.dir`) from a content provider that holds Note Pad data (`vnd.google.note`). The Intent object that launches the activity would include a `content:` URI specifying the exact data of this type that the activity should open.

Note also the DEFAULT category supplied in this filter. It's there because the Context.startActivity() and Activity.startActivityForResult() methods treat all intents as if they contained the DEFAULT category — with just two exceptions:

- Intents that explicitly name the target activity
- Intents consisting of the MAIN action and LAUNCHER category

Therefore, the DEFAULT category is *required* for all filters — except for those with the MAIN action and LAUNCHER category. (Intent filters are not consulted for explicit intents.)

3.
```
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

This filter describes the activity's ability to return a note selected by the user without requiring any specification of the directory the user should choose from. The GET_CONTENT action is similar to the PICK action. In both cases, the activity returns the URI for a note selected by the user. (In each case, it's returned to the activity that called startActivityForResult() to start the NoteList activity.) Here, however, the caller specifies the type of data desired instead of the directory of data the user will be picking from.

The data type, vnd.android.cursor.item/vnd.google.note, indicates the type of data the activity can return — a URI for a single note. From the returned URI, the caller can get a Cursor for exactly one item (vnd.android.cursor.item) from the content provider that holds Note Pad data (vnd.google.note).

In other words, for the PICK action in the previous filter, the data type indicates the type of data the activity could display to the user. For the GET_CONTENT filter, it indicates the type of data the activity can return to the caller.

Given these capabilities, the following intents will resolve to the NotesList activity:

action: android.intent.action.MAIN
: Launches the activity with no data specified.

action: android.intent.action.MAIN
category: android.intent.category.LAUNCHER
: Launches the activity with no data selected specified. This is the actual intent used by the Launcher to populate its top-level list. All activities with filters that match this action and category are added to the list.

action: android.intent.action.VIEW
data: content://com.google.provider.NotePad/notes
: Asks the activity to display a list of all the notes under content://com.google.provider.NotePad/notes. The user can then browse through the list and get information about the items in it.

action: android.intent.action.PICK
data: content://com.google.provider.NotePad/notes
: Asks the activity to display a list of the notes under content://com.google.provider.NotePad/notes. The user can then pick a note from the list, and the activity will return the URI for that item back to the activity that started the NoteList activity.

action: android.intent.action.GET_CONTENT
data type: vnd.android.cursor.item/vnd.google.note
: Asks the activity to supply a single item of Note Pad data.

The second activity, NoteEditor, shows users a single note entry and allows them to edit it. It can do two things as described by its two intent filters:

1.
```
<intent-filter android:label="@string/resolve_edit">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <action android:name="com.android.notepad.action.EDIT_NOTE" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The first, primary, purpose of this activity is to enable the user to interact with a single note &mdash to either `VIEW` the note or `EDIT` it. (The `EDIT_NOTE` category is a synonym for `EDIT`.) The intent would contain the URI for data matching the MIME type `vnd.android.cursor.item/vnd.google.note` — that is, the URI for a single, specific note. It would typically be a URI that was returned by the `PICK` or `GET_CONTENT` actions of the NoteList activity.

As before, this filter lists the `DEFAULT` category so that the activity can be launched by intents that don't explicitly specify the NoteEditor class.

2.
```
<intent-filter>
    <action android:name="android.intent.action.INSERT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.dir/vnd.google.note" />
</intent-filter>
```

The secondary purpose of this activity is to enable the user to create a new note, which it will `INSERT` into an existing directory of notes. The intent would contain the URI for data matching the MIME type `vnd.android.cursor.dir/vnd.google.note` — that is, the URI for the directory where the note should be placed.

Given these capabilities, the following intents will resolve to the NoteEditor activity:

action: `android.intent.action.VIEW`
data: `content://com.google.provider.NotePad/notes/`*`ID`*
Asks the activity to display the content of the note identified by *`ID`*. (For details on how `content:` URIs specify individual members of a group, see Content Providers.)

action: `android.intent.action.EDIT`
data: `content://com.google.provider.NotePad/notes/`*`ID`*
Asks the activity to display the content of the note identified by *`ID`*, and to let the user edit it. If the user saves the changes, the activity updates the data for the note in the content provider.

action: `android.intent.action.INSERT`
data: `content://com.google.provider.NotePad/notes`
Asks the activity to create a new, empty note in the notes list at `content://com.google.provider.NotePad/notes` and allow the user to edit it. If the user saves the note, its URI is returned to the caller.

The last activity, TitleEditor, enables the user to edit the title of a note. This could be implemented by directly invoking the activity (by explicitly setting its component name in the Intent), without using an intent filter. But here we take the opportunity to show how to publish alternative operations on existing data:

```
<intent-filter android:label="@string/resolve_title">
    <action android:name="com.android.notepad.action.EDIT_TITLE" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.ALTERNATIVE" />
    <category android:name="android.intent.category.SELECTED_ALTERNATIVE" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
</intent-filter>
```

The single intent filter for this activity uses a custom action called "`com.android.notepad.action.EDIT_TITLE`". It must be invoked on a specific note (data type `vnd.android.cursor.item/vnd.google.note`), like the previous `VIEW` and `EDIT` actions. However, here the activity displays the title contained in the note data, not the content of the note itself.

In addition to supporting the usual `DEFAULT` category, the title editor also supports two other standard categories: `ALTERNATIVE` and `SELECTED_ALTERNATIVE`. These categories identify activities that can be presented to users in a menu of options (much as the `LAUNCHER` category identifies activities that should be presented to user in the application launcher). Note that the filter also supplies an explicit label (via `android:label="@string/resolve_title"`) to better control what users see when presented with this activity as an alternative action to the data they are currently viewing. (For more information on these categories and building options menus, see the `PackageManager.queryIntentActivityOptions()` and

`Menu.addIntentOptions()` methods.)

Given these capabilities, the following intent will resolve to the TitleEditor activity:

action: `com.android.notepad.action.EDIT_TITLE`
data: `content://com.google.provider.NotePad/notes/`*`ID`*
Asks the activity to display the title associated with note *ID*, and allow the user to edit the title.

↑ Go to top