

# Android - Notifications

Notifications are messages that you can display to the user while he is outside of your app UI. after creating the notification we send it to the system and it immediately display it in the notification bar, the user can expand the drawer and push on the notification.

Creating and posting simple notification

1. Create an instance of **Notification.Builder** and pass the context.
2. Set the title using **setContentTitle(notifTitle)**
3. Set the notification text using **setContentText(notifText)**
4. Set the notification's small icon(icon shown in the taskbar) using **setSmallIcon(icon)** **Note:** all of the above methods returns the new builder after modifying it so you can chain the functions calls(Same as the alertDialog builder).
5. Use the builder's **build()** function and get the new generated notification.
6. get an instance of the NotificationManager using the **getSystemService(Context.NOTIFICATION\_SERVICE)**
7. Use the Notification Manager's **notify()** function and pass an int representing the notification's id and the notification itself.

## Notification actions

The actions are optional and triggered when the user press on the notification. The most common action is opening an Activity in your application. You can also add buttons to the notification that perform additional actions such as snoozing an alarm or responding immediately to a text message (see the **addAction()** method).

Inside a Notification, the action itself is defined by a **PendingIntent** containing an **Intent** that starts an Activity in your application(or any other component). You add the **PendingIntent** by calling the builder's **setContentIntent(pendingIntent)**.

# Understanding PendingIntent

PendingIntent is used to grant other application the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity). In our case, we grant the Notification manager the right to execute our intent to open an activity with our permissions. As such, we should be careful about who we give this permissions.

PendingIntent can be created with the class static function :

**PendingIntent.getActivity**(Context context, int requestCode, Intent intent, int flags)

The above line of code retrieves a PendingIntent that will start a new activity, like calling Context.**startActivity**(Intent) - but the notification manager can execute this.

The last argument is the PendingIntent flag. The flags can take any of the following values (these flags tells the factory method - PendingIntent.getActivity - what to do in case a pendingintent that already been created):

FLAG\_CANCEL\_CURRENT - if this pendingintent already exists, the current one is canceled before generating a new one. Meaning the intent in the pending intent will be deleted, and the new one will take his place.

FLAG\_NO\_CREATE - if this pendingintent already exist do not create a new one, meaning the intent inside will stay the same one.

FLAG\_UPDATE\_CURRENT- if the described PendingIntent already exists, then keep it but its replace its extra data with what is in this new Intent, meaning the intent within the pendingintent is not deleted but its extras are updated.

## Steps to trigger an Activity upon notification press:

1. Create the explicit intent with the activity you wish to open and enter an extra for that activity that holds the notification's info.

2. Create a PendingIntent using the method of PendingIntent described above.
3. set the pendingintent using the builder's `setContentIntent(pendingIntent)` function.

## Notification flags

After creating the notification we can add a flags by bitwise bit flags (the | operation)

Here are some of the important flags:

<code>FLAG_AUTO_CANCEL</code>	should be set if the notification should be canceled when it is clicked by the user.
<code>FLAG_INSISTENT</code>	the audio will be repeated until the notification is cancelled or the notification window is opened.
<code>FLAG_NO_CLEAR</code>	should be set if the notification should not be canceled when the user clicks the Clear all button.
<code>FLAG_ONGOING_EVENT</code>	should be set if this notification is in reference to something that is ongoing, like a phone call.
<code>FLAG_SHOW_LIGHTS</code>	should be set if you want the LED on for this notification.

## Notification defaults

Each notification has a defaults attribute that Specifies which values should be taken from the defaults. defaults values can be (will be bitwise-ored like before and set with the builder's `setDefaults()` function)

<code>DEFAULT_ALL</code>	Use all default values (where applicable).
<code>DEFAULT_LIGHTS</code>	Use the default notification lights.
<code>DEFAULT_SOUND</code>	Use the default notification sound.

DEFAULT_VIBRATE	Use the default notification vibrate.
-----------------	---------------------------------------

## Canceling notifications

Notifications remain visible until one of the following happens:

- The user dismisses the notification either individually or by using "Clear All" (if the notification can be cleared).
- The user clicks the notification, and you called the builder's `setAutoCancel()` when you created the notification.
- You call the NotificationManger's `cancel(int)` function for a specific notification ID. This method also deletes ongoing notifications.
- You call the NotificationManger's `cancelAll()` function, which removes all of the notifications you previously issued.

## Notification Channels

Android Oreo introduced Notification Channels, they provide us with the ability to group the notifications that our application sends into manageable groups. Once our notifications are in these channels, we no longer have input into their functionality — so it is up to the user to manage these channels. The user can block a single notification channel and allow another, the user can also set the importance of notifications from any channel.

When targeting android O each notification has a channel, we create the channel giving it a name and an id, also we can customize each channel with the following:

```
String channelId = "some_channel_id";
CharSequence channelName = "Some Channel";
int importance = NotificationManager.IMPORTANCE_HIGH;
NotificationChannel notificationChannel = new NotificationChannel(channelId, channelName, importance);
notificationChannel.enableLights(true);
notificationChannel.setLightColor(Color.RED);
notificationChannel.enableVibration(true);
notificationChannel.setVibrationPattern(new long[]{100, 200, 300, 400, 500, 400, 300, 200, 400});
```

After configuring the channel we create it like this:

```
notificationManager.createNotificationChannel(notificationChannel);
```

And then assigning the channel to the notification:

```
builder.setChannelId(channelId);
```

This we do only for android o, but don't worry about backward compatibility, these channel will be completely ignored on previous versions of android.

## Updating notifications

To set up a notification so it can be updated, issue it with a notification ID by calling `NotificationManager.notify()`. To update this notification once you've issued it, update or create a `NotificationCompat.Builder` object, build a `Notification` object from it, and issue the `Notification` with the same ID you used previously. If the previous notification is still visible, the system updates it from the contents of the `Notification` object. If the previous notification has been dismissed, a new notification is created instead.

## Displaying Progress in a Notification

Notifications can include an animated progress indicator that shows users the status of an ongoing operation. If you can estimate how long the operation takes and how much of it is complete at any time, use the "determinate" form of the indicator (a progress bar). If you can't estimate the length of the operation, use the "indeterminate" form of the indicator (an activity indicator).

To use a progress indicator on platforms starting with Android 4.0, call **`setProgress()`**. For previous versions, you must create your own custom notification layout that includes a `ProgressBar` view.

To display a determinate progress bar, add the bar to your notification by calling **`setProgress(max, progress, false)`** and then issue the notification. As your operation proceeds, increment progress, and update the notification. At the end of the operation, progress should equal max. To remove the progress bar, call **`setProgress(0, 0, false)`**.

## Notification priority

If you wish, you can set the priority of a notification. The priority acts as a hint to the device about how to order the notifications in the notifications drawer. To set a notification's priority, call the builder's `setPriority()` and pass in one of the priority constants. There are five priority levels, ranging from `PRIORITY_MIN` (-2) to `PRIORITY_MAX` (2); if not set, the priority defaults to `PRIORITY_DEFAULT` (0).

## Lock Screen Notifications

With the release of Android 5.0 (API level 21), notifications may now appear on the lock screen. Users can choose via Settings whether to display notifications on the lock screen, and you can designate whether a notification from your app is visible on the lock screen.

You call the builder's `setVisibility()` and specify one of the following values:

`VISIBILITY_PUBLIC` shows the notification's full content.

`VISIBILITY_SECRET` doesn't show any part of this notification on the lock screen.

`VISIBILITY_PRIVATE` shows basic information, such as the notification's icon and the content title, but hides the notification's full content.

## Adding custom actions

We can add additional actions to the notification by using the builder's **`addAction()`** method.

You can add up to three actions to the notification. the call to **`addAction()`** accept 3 args:

*icon*                      Resource ID of a drawable that represents the action.

*title*                      Text describing the action.

*intent*                      PendingIntent to be fired when the action is invoked.

# Adding custom view

We can add our own view to the notification. For doing this we need to create our own layout file for the notification, after doing so we need to create a RemoteViews instance from it by passing the package name (**getPackageName()**) and the layout file name to the RemoteViews constructor. After doing that we need to create out PendingIntents for each action and button we want and use the RemoteViews **setOnClickPendingIntent** function and pass it the button or any other widget id and the desired pending intent we want to be executed. After finishing all the configuration we need to set the remote view instance as the notification contentview (note, by doing so all the text, title and all the other content will be overridden).

## Additional builder's functions

**setAutoCancel(boolean)** - if true the notification will be canceled when the user pressed on it.

**setDeleteIntent (PendingIntent intent)** - Supply a PendingIntent to send when the notification is cleared explicitly by the user.

**setLargeIcon** - Add a large icon to the notification content view. when the user has expanded the notification bar he will see the large icon. If not specified the small icon is shown.

**setSound (Uri sound)** - Set the sound to play.

**setWhen (long when)** - Add a timestamp pertaining to the notification (usually the time the event occurred). It will be shown in the notification content view by default

### References

Android developer guide: [Notifications](#) and [Notification.Builder](#) by Google©