|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. **CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS.** 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. VERIFY SUCCESSFUL API INTEGRATION 6. VERIFY DEVELOPED MOBILE APP. 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(2/15) | Page: 1 of74 |

**TITLE**:

**MOBILE APP USER INTERFACE ELEMENT**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of mobile app programming and user interface.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts and foundation for the understanding of the mobile application, how to develop it and designing a good user interface.

# **LAYOUT**

A layout defines the structure for a user interface in your app, such as in an activity. All elements in the layout are built using a hierarchy of View and ViewGroup objects. A View usually draws something the user can see and interact with. Whereas a ViewGroup is an invisible container that defines the layout structure for View and other ViewGroup objects, as shown in figure 1.0.

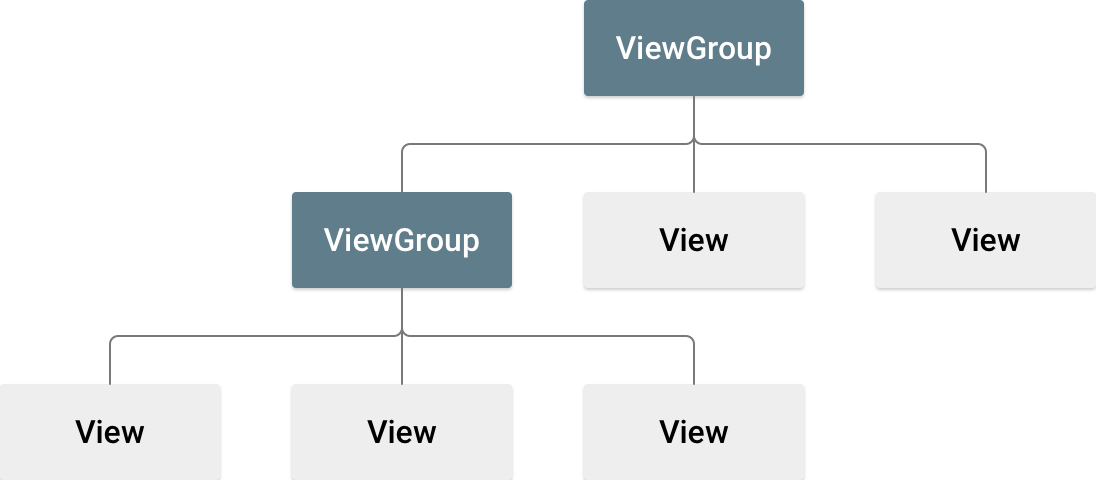


Figure 1.0 Illustration of a view hierarchy, which defines a UI layout

The View objects are usually called "widgets" and can be one of many subclasses, such as Button or TextView. The ViewGroup objects are usually called "layouts" can be one of many types that provide a different layout structure, such as LinearLayout or ConstraintLayout .

You can declare a layout in two ways:

* **Declare UI elements in XML**. Android provides a straightforward XML vocabulary that corresponds to the View classes and subclasses, such as those for widgets and layouts.

You can also use Android Studio's Layout Editor to build your XML

layout using a drag-and-drop interface.

* **Instantiate layout elements at runtime**. Your app can create View and ViewGroup objects (and manipulate their properties) programmatically.

Declaring your UI in XML allows you to separate the presentation of your app from the code that controls its behavior. Using XML files also makes it easy to provide different layouts for different screen sizes and orientations (discussed further in Supporting Different Screen Sizes).

The Android framework gives you the flexibility to use either or both of these methods to build your app's UI. For example, you can declare your app's default layouts in XML, and then modify the layout at runtime.

1. Write the XML

Using Android's XML vocabulary, you can quickly design UI layouts and the screen elements they contain, in the same way you create web pages in HTML — with a series of nested elements.

Each layout file must contain exactly one root element, which must be a View or ViewGroup object. Once you've defined the root element, you can add additional layout objects or widgets as child elements to gradually build a View hierarchy that defines your layout. For example, here's an XML layout that uses a vertical LinearLayout to hold a TextView and a Button:

<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical" >

<TextView android:id="@+id/text"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a TextView" />

<Button android:id="@+id/button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Hello, I am a Button" />

</LinearLayout>

After you've declared your layout in XML, save the file with the .xml extension, in your Android project's res/layout/ directory, so it will properly compile.

1. Load the XML Resource

When you compile your app, each XML layout file is compiled into a View resource. You should load the layout resource from your app code, in your Activity.onCreate() callback implementation. Do so by calling setContentView(), passing it the reference to your layout resource in the form of: R.layout.layout\_file\_name. For example, if your XML layout is saved as main\_layout.xml, you would load it for your Activity like so:

fun onCreate(savedInstanceState: Bundle) {

super.onCreate(savedInstanceState)

setContentView(R.layout.main\_layout)

}

The onCreate() callback method in your Activity is called by the Android framework when your Activity is launched (see the discussion about lifecycles, in the Activities document).

1. Attributes

Every View and ViewGroup object supports their own variety of XML attributes. Some attributes are specific to a View object (for example, TextView supports the textSize attribute), but these attributes are also inherited by any View objects that may extend this class. Some are common to all View objects, because they are inherited from the root View class (like the id attribute). And, other attributes are considered "layout parameters," which are attributes that describe certain layout orientations of the View object, as defined by that object's parent ViewGroup object.

1. ID

Any View object may have an integer ID associated with it, to uniquely identify the View within the tree. When the app is compiled, this ID is referenced as an integer, but the ID is typically assigned in the layout XML file as a string, in the id attribute. This is an XML attribute common to all View objects (defined by the View class) and you will use it very often. The syntax for an ID, inside an XML tag is.

android:id="@+id/my\_button"

The at-symbol (@) at the beginning of the string indicates that the XML parser should parse and expand the rest of the ID string and identify it as an ID resource. The plus-symbol (+) means that this is a new resource name that must be created and added to our resources (in the R.java file). There are a number of other ID resources that are offered by the Android framework. When referencing an Android resource ID, you do not need the plus-symbol, but must add the android package namespace, like so:

android:id="@android:id/empty"

With the android package namespace in place, we're now referencing an ID from the android.R resources class, rather than the local resources class.

In order to create views and reference them from the app, a common pattern is to:

i) Define a view/widget in the layout file and assign it a unique ID:

<Button android:id="@+id/my\_button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="@string/my\_button\_text"/>

ii) Then create an instance of the view object and capture it from the

layout (typically in the onCreate() method):

val myButton: Button = findViewById(R.id.my\_button)

Defining IDs for view objects is important when creating a RelativeLayout. In a relative layout, sibling views can define their layout relative to another sibling view, which is referenced by the unique ID.

An ID need not be unique throughout the entire tree, but it should be unique within the part of the tree you are searching (which may often be the entire tree, so it's best to be completely unique when possible).

1. Layout Parameters

XML layout attributes named layout\_something define layout parameters for the View that are appropriate for the ViewGroup in which it resides.

Every ViewGroup class implements a nested class that extends ViewGroup.LayoutParams. This subclass contains property types that define the size and position for each child view, as appropriate for the view group. As you can see in figure 1.2, the parent view group defines layout parameters for each child view (including the child view group).

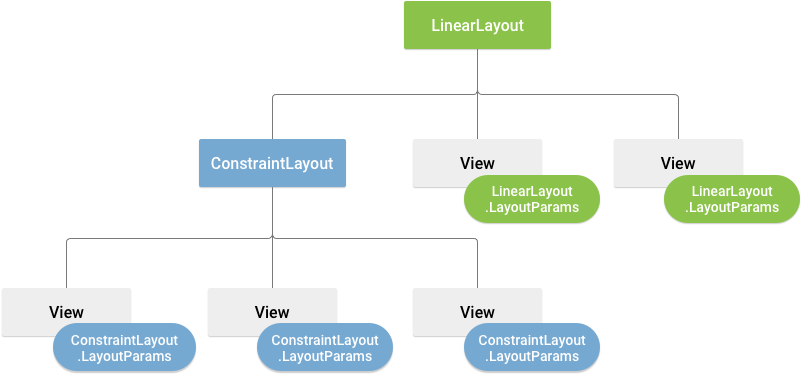


Figure 1.2 Visualization of a view hierarchy with layout parameters associated with each view

Note that every LayoutParams subclass has its own syntax for setting values. Each child element must define LayoutParams that are appropriate for its parent, though it may also define different LayoutParams for its own children.

All view groups include a width and height (layout\_width and layout\_height), and each view is required to define them. Many LayoutParams also include optional margins and borders.

You can specify width and height with exact measurements, though you probably won't want to do this often. More often, you will use one of these constants to set the width or height:

* wrap\_content tells your view to size itself to the dimensions required by its content.
* match\_parent tells your view to become as big as its parent view group will allow.

In general, specifying a layout width and height using absolute units such as pixels is not recommended. Instead, using relative measurements such as density-independent pixel units (dp), wrap\_content, or match\_parent, is a better approach, because it helps ensure that your app will display properly across a variety of device screen sizes. The accepted measurement types are defined in the Available Resources document.

1. Layout Position

The geometry of a view is that of a rectangle. A view has a location, expressed as a pair of left and top coordinates, and two dimensions, expressed as a width and a height. The unit for location and dimensions is the pixel.

It is possible to retrieve the location of a view by invoking the methods getLeft() and getTop(). The former returns the left, or X, coordinate of the rectangle representing the view. The latter returns the top, or Y, coordinate of the rectangle representing the view. These methods both return the location of the view relative to its parent. For instance, when getLeft() returns 20, that means the view is located 20 pixels to the right of the left edge of its direct parent.

In addition, several convenience methods are offered to avoid unnecessary computations, namely getRight() and getBottom(). These methods return the coordinates of the right and bottom edges of the rectangle representing the view. For instance, calling getRight() is similar to the following computation: getLeft() + getWidth().

1. Size, Padding and Margins

The size of a view is expressed with a width and a height. A view actually possesses two pairs of width and height values.

The first pair is known as measured width and measured height. These dimensions define how big a view wants to be within its parent. The measured dimensions can be obtained by calling getMeasuredWidth() and getMeasuredHeight().

The second pair is simply known as width and height, or sometimes drawing width and drawing height. These dimensions define the actual size of the view on screen, at drawing time and after layout. These values may, but do not have to, be different from the measured width and height. The width and height can be obtained by calling getWidth() and getHeight().

To measure its dimensions, a view takes into account its padding. The padding is expressed in pixels for the left, top, right and bottom parts of the view. Padding can be used to offset the content of the view by a specific number of pixels. For instance, a left padding of 2 will push the view's content by 2 pixels to the right of the left edge. Padding can be set using the setPadding(int, int, int, int) method and queried by calling getPaddingLeft(), getPaddingTop(), getPaddingRight() and getPaddingBottom().

1. Common Layouts

Each subclass of the ViewGroup class provides a unique way to display the views you nest within it. Below are some of the more common layout types that are built into the Android platform.

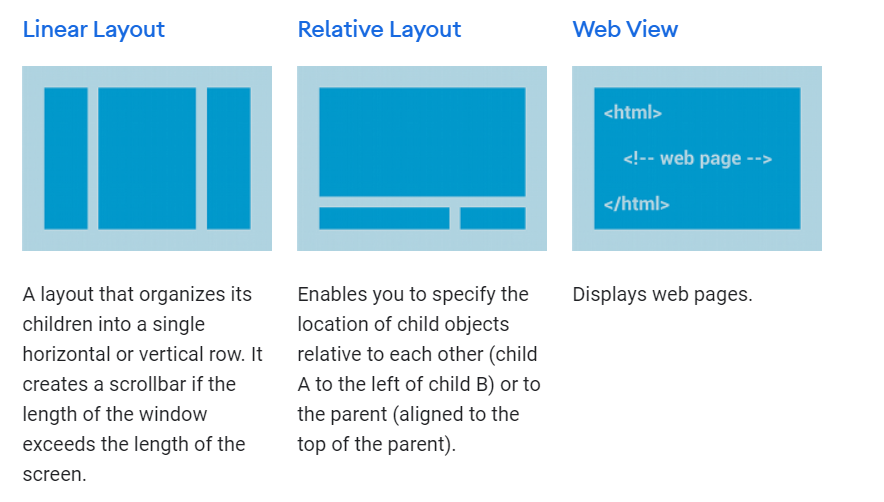


Figure 1.3

1. Building Layouts with an Adapter

When the content for your layout is dynamic or not pre-determined, you can use a layout that subclasses AdapterView to populate the layout with views at runtime. A subclass of the AdapterView class uses an Adapter to bind data to its layout. The Adapter behaves as a middleman between the data source and the AdapterView layout—the Adapter retrieves the data (from a source such as an array or a database query) and converts each entry into a view that can be added into the AdapterView layout.

Common layouts backed by an adapter include:

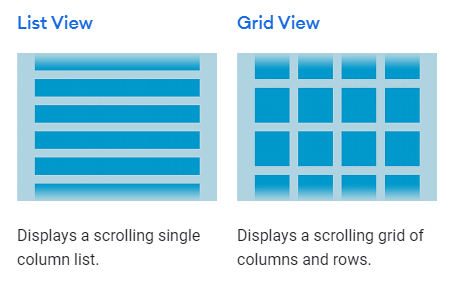


Figure 1.4

1. Filling an adapter view with data

You can populate an AdapterView such as ListView or GridView by binding the AdapterView instance to an Adapter, which retrieves data from an external source and creates a View that represents each data entry.

Android provides several subclasses of Adapter that are useful for retrieving different kinds of data and building views for an AdapterView. The two most common adapters are:

* ArrayAdapter

Use this adapter when your data source is an array. By default, ArrayAdapter creates a view for each array item by calling toString() on each item and placing the contents in a TextView.

For example, if you have an array of strings you want to display in a ListView, initialize a new ArrayAdapter using a constructor to specify the layout for each string and the string array:

val adapter = ArrayAdapter<String>(this, android.R.layout.simple\_list\_item\_1, myStringArray)

The arguments for this constructor are:

* Your app Context
* The layout that contains a TextView for each string in the array
* The string array

Then simply call setAdapter() on your ListView:

val listView: ListView = findViewById(R.id.listview)

listView.adapter = adapter

To customize the appearance of each item you can override the toString() method for the objects in your array. Or, to create a view for each item that's something other than a TextView (for example, if you want an ImageView for each array item), extend the ArrayAdapter class and override getView() to return the type of view you want for each item.

* SimpleCursorAdapter

Use this adapter when your data comes from a Cursor. When using SimpleCursorAdapter, you must specify a layout to use for each row in the Cursor and which columns in the Cursor should be inserted into which views of the layout. For example, if you want to create a list of people's names and phone numbers, you can perform a query that returns a Cursor containing a row for each person and columns for the names and numbers. You then create a string array specifying which columns from the Cursor you want in the layout for each result and an integer array specifying the corresponding views that each column should be placed:

val fromColumns = arrayOf(ContactsContract.Data.DISPLAY\_NAME,

ContactsContract.CommonDataKinds.Phone.NUMBER)

val toViews = intArrayOf(R.id.display\_name, R.id.phone\_number)

When you instantiate the SimpleCursorAdapter, pass the layout to use for each result, the Cursor containing the results, and these two arrays:

val adapter = SimpleCursorAdapter(this,R.layout.person\_name\_and\_number, cursor, fromColumns, toViews, 0)

val listView = getListView()

listView.adapter = adapter

The SimpleCursorAdapter then creates a view for each row in the Cursor using the provided layout by inserting each fromColumns item into the corresponding toViews view.

1. Handling click events

You can respond to click events on each item in an AdapterView by implementing the AdapterView.OnItemClickListener interface. For example:

listView.onItemClickListener = AdapterView.OnItemClickListener { parent, view, position, id ->

// Do something in response to the click

}

## Build a Responsive UI

Build a Responsive UI with ConstraintLayout ConstraintLayout allows you to create large and complex layouts with a flat view hierarchy (no nested view groups). It's similar to RelativeLayout in that all views are laid out according to relationships between sibling views and the parent layout, but it's more flexible than RelativeLayout and easier to use with Android Studio's Layout Editor.

All the power of ConstraintLayout is available directly from the Layout Editor's visual tools, because the layout API and the Layout Editor were specially built for each other. So you can build your layout with ConstraintLayout entirely by drag-and-dropping instead of editing the XML.

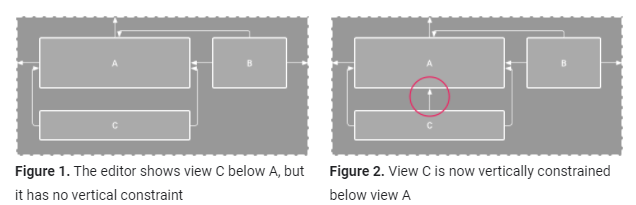
This page provides a guide to building a layout with ConstraintLayout in Android Studio 3.0 or higher. If you'd like more information about the Layout Editor itself, see the Android Studio guide to Build a UI with Layout Editor.

1. Constraints overview

To define a view's position in ConstraintLayout, you must add at least one horizontal and one vertical constraint for the view. Each constraint represents a connection or alignment to another view, the parent layout, or an invisible guideline. Each constraint defines the view's position along either the vertical or horizontal axis; so each view must have a minimum of one constraint for each axis, but often more are necessary.

When you drop a view into the Layout Editor, it stays where you leave it even if it has no constraints. However, this is only to make editing easier; if a view has no constraints when you run your layout on a device, it is drawn at position [0,0] (the top-left corner).

In figure 1, the layout looks good in the editor, but there's no vertical constraint on view C. When this layout draws on a device, view C horizontally aligns with the left and right edges of view A, but appears at the top of the screen because it has no vertical constraint.



Although a missing constraint won't cause a compilation error, the Layout Editor indicates missing constraints as an error in the toolbar. To view the errors and other warnings, click Show Warnings and Errors. To help you avoid missing constraints, the Layout Editor can automatically add constraints for you with the Autoconnect and infer constraints features.

1. Add ConstraintLayout to your project

To use ConstraintLayout in your project, proceed as follows:

1. Ensure you have the maven.google.com repository declared in your top-level build.gradle file:

repositories {

google()

}

1. Add the library as a dependency in the module-level build.gradle file, as shown in the following example. Note that the latest version might be different than what is shown in the example:

dependencies { implementation("androidx.constraintlayout:constraintlayout:2.0.4")

}

1. In the toolbar or sync notification, click Sync Project with Gradle Files.
2. Convert a layout

To convert an existing layout to a constraint layout, follow these steps:

1. Open your layout in Android Studio and click the Design tab at the bottom of the editor window.
2. In the Component Tree window, right-click the layout and click Convert layout to ConstraintLayout.

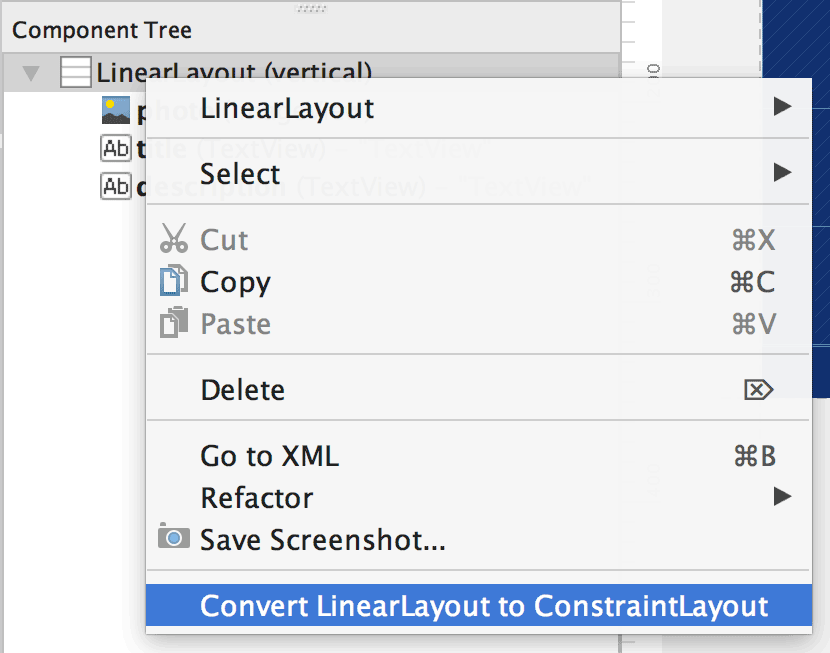


Figure 3 The menu to convert a layout to ConstraintLayout

1. Create a new layout

To start a new constraint layout file, follow these steps:

1. In the Project window, click the module folder and then select File > New > XML > Layout XML.
2. Enter a name for the layout file and enter "androidx.constraintlayout.widget.ConstraintLayout" for the Root Tag.
3. Click Finish.
4. Add or remove a constraint

To add a constraint, do the following:

1. Drag a view from the Palette window into the editor.

When you add a view in a ConstraintLayout, it displays a bounding box with square resizing handles on each corner and circular constraint handles on each side.

1. Click the view to select it.
2. Do one of the following:

* Click a constraint handle and drag it to an available anchor point. This point can be the edge of another view, the edge of the layout, or a guideline. Notice that as you drag the constraint handle, the Layout Editor shows potential connection anchors and blue overlays.
* Click one of the Create a connection buttons in the Layout section of the Attributes window, as shown in figure 4.

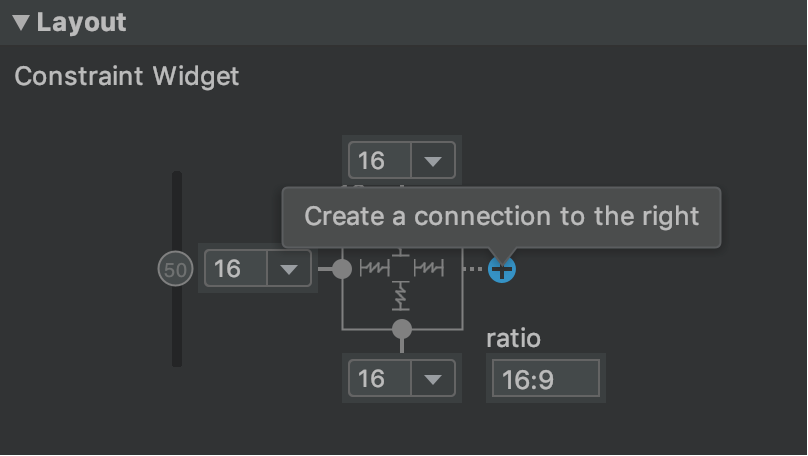


Figure 4. The Layout section of the Attributes window lets you create connections.

When the constraint is created, the editor gives it a default margin to separate the two views.

When creating constraints, remember the following rules:

* Every view must have at least two constraints: one horizontal and one vertical.
* You can create constraints only between a constraint handle and an anchor point that share the same plane. So a vertical plane (the left and right sides) of a view can be constrained only to another vertical plane; and baselines can constrain only to other baselines.
* Each constraint handle can be used for just one constraint, but you can create multiple constraints (from different views) to the same anchor point.

You can delete a constraint by doing any of the following:

* Click on a constraint to select it, and then press Delete.
* Press and hold Control (Command on macOS), and then click on a constraint anchor. Note that the constraint turns red to indicate that you can click to delete it, as shown in figure 5.

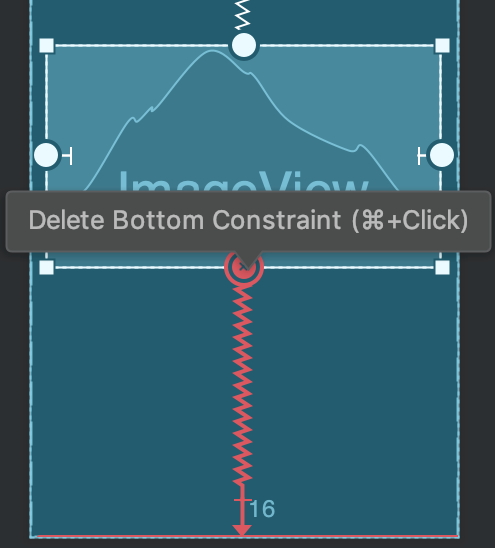
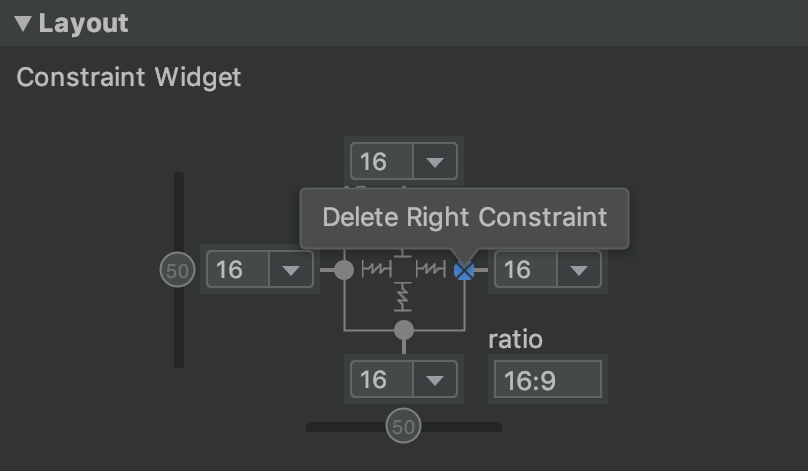


Figure 5 A red constraint indicates that you can click to delete it

In the Layout section of the Attributes window, click on a constraint anchor, as shown in figure 6.

Figure 6 Click on a constraint anchor to delete it

If you add opposing constraints on a view, the constraint lines become squiggly like a spring to indicate the opposing forces, as shown in video 2. The effect is most visible when the view size is set to "fixed" or "wrap content," in which case the view is centered between the constraints. If you instead want the view to stretch its size to meet the constraints, switch the size to "match constraints"; or if you want to keep the current size but move the view so that it is not centered, adjust the constraint bias.

You can use constraints to achieve different types of layout behavior, as described in the following sections.

1. Parent position

Constrain the side of a view to the corresponding edge of the layout.

In figure 7, the left side of the view is connected to the left edge of the parent layout. You can define the distance from the edge with margin.

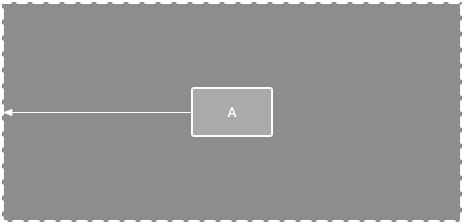


Figure 7 A horizontal constraint to the parent

1. Order position

Define the order of appearance for two views, either vertically or horizontally.

In figure 8, B is constrained to always be to the right of A, and C is constrained below A. However, these constraints do not imply alignment, so B can still move up and down.

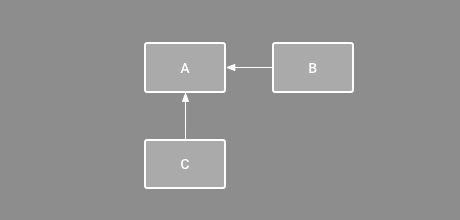


Figure 8 A horizontal and vertical constraint

1. Alignment

Align the edge of a view to the same edge of another view.

In figure 9, the left side of B is aligned to the left side of A. If you want to align the view centers, create a constraint on both sides.

You can offset the alignment by dragging the view inward from the constraint. For example, figure 10 shows B with a 24dp offset alignment. The offset is defined by the constrained view's margin.

You can also select all the views you want to align, and then click Align in the toolbar to select the alignment type.

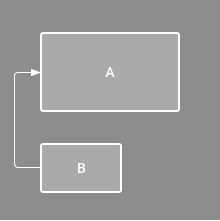


Figure 9. A horizontal alignment constraint

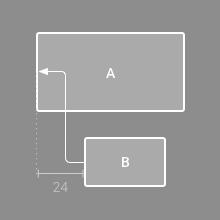


Figure 10. An offset horizontal alignment constraint

1. Baseline Alignment

Align the text baseline of a view to the text baseline of another view.

In figure 11, the first line of B is aligned with the text in A.

To create a baseline constraint, right-click the text view you want to constrain and then click Show Baseline. Then click on the text baseline and drag the line to another baseline.

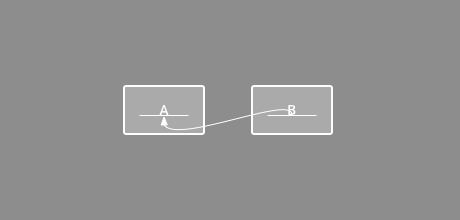


Figure 11. A baseline alignment constraint

1. Constrain to a guideline

You can add a vertical or horizontal guideline to which you can constrain views, and the guideline will be invisible to app users. You can position the guideline within the layout based on either dp units or percent, relative to the layout's edge.

To create a guideline, click Guidelines in the toolbar, and then click either Add Vertical Guideline or Add Horizontal Guideline.

Drag the dotted line to reposition it and click the circle at the edge of the guideline to toggle the measurement mode.

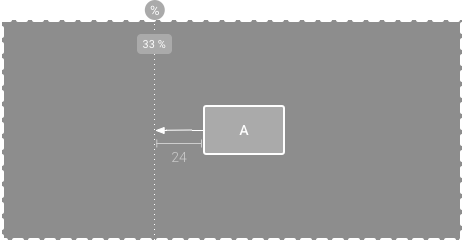


Figure 12. A view constrained to a guideline

1. Constrain to a barrier

Similar to a guideline, a barrier is an invisible line that you can constrain views to. Except a barrier does not define its own position; instead, the barrier position moves based on the position of views contained within it. This is useful when you want to constrain a view to the a set of views rather than to one specific view.

For example, figure 13 shows view C is constrained to the right side of a barrier. The barrier is set to the "end" (or the right side in a left-to-right layout) of both view A and view B. So the barrier moves depending on whether the right side of view A or view B is is farthest right.

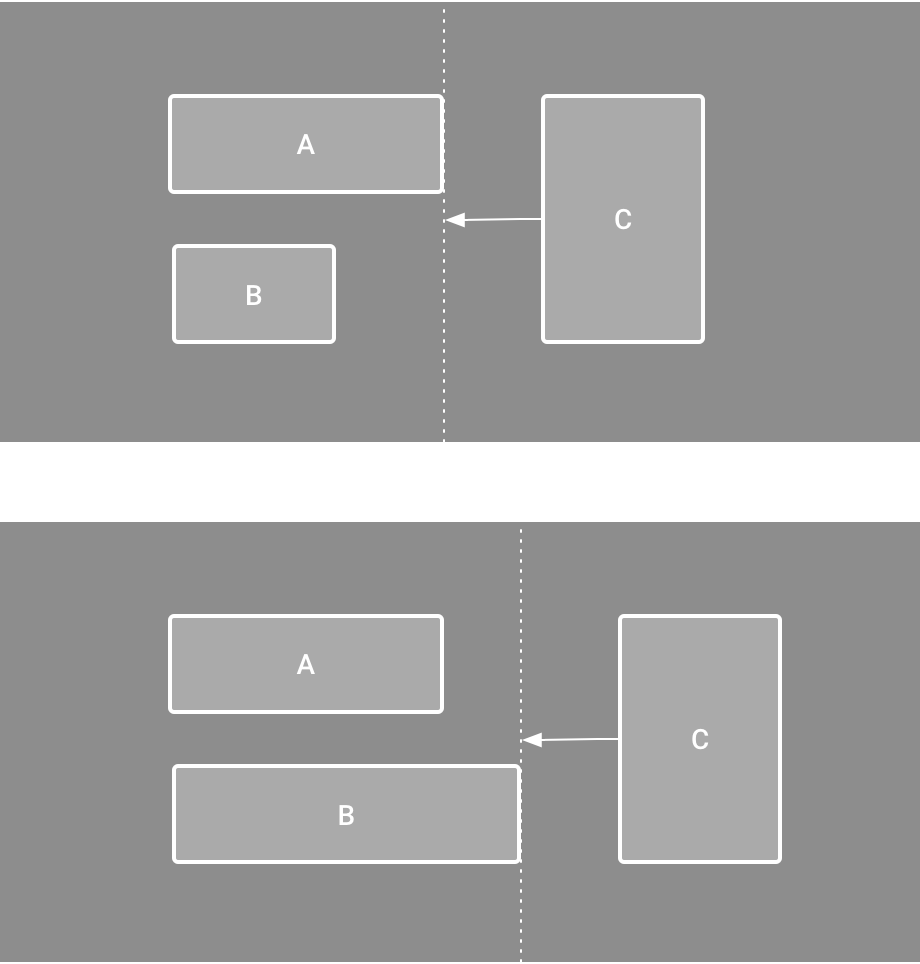


Figure 13. View C is constrained to a barrier, which moves based on the position/size of both view A and view B

To create a barrier, follow these steps:

1. Click Guidelines in the toolbar, and then click Add Vertical Barrier or Add Horizontal Barrier.
2. In the Component Tree window, select the views you want inside the barrier and drag them into the barrier component.
3. Select the barrier from the Component Tree, open the Attributes window, and then set the barrierDirection.

Now you can create a constraint from another view to the barrier.

You can also constrain views that are inside the barrier to the barrier. This way, you can ensure that all views in the barrier always align to each other, even if you don't know which view will be the longest or tallest.

You can also include a guideline inside a barrier to ensure a "minimum" position for the barrier.

1. Adjust the constraint bias

When you add a constraint to both sides of a view (and the view size for the same dimension is either "fixed" or "wrap content"), the view becomes centered between the two constraints with a bias of 50% by default. You can adjust the bias by dragging the bias slider in the Attributes window or by dragging the view.

1. Adjust the view size

You can use the corner handles to resize a view, but this hard codes the size so the view will not resize for different content or screen sizes. To select a different sizing mode, click a view and open the Attributes window on the right side of the editor.

Near the top of the Attributes window is the view inspector, which includes controls for several layout attributes, as shown in figure 14 (this is available only for views in a constraint layout).

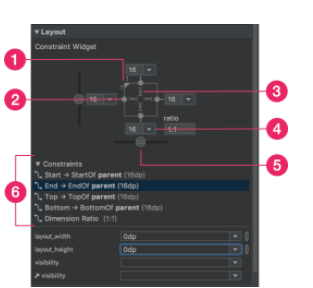


Figure 14. When selecting a view, the Attributes window includes controls for 1 size ratio, 2 deleting constraints, 3 height/width mode, 4 margins, and 5 constraint bias. You can also highlight individual constraints in the Layout Editor by clicking on them in the 6 constraint list.

You can change the way the height and width are calculated by clicking the symbols indicated with callout 3 in figure 14. These symbols represent the size mode as follows (click the symbol to toggle between these settings):

* Fixed: You specify a specific dimension in the text box below or by resizing the view in the editor.
* Wrap Content: The view expands only as much as needed to fit its contents.
* Match Constraints: The view expands as much as possible to meet the constraints on each side (after accounting for the view's margins). However, you can modify that behavior with the following attributes and values (these attributes take effect only when you set the view width to match constraints):
* layout\_constraintWidth\_default
  + spread: Expands the view as much as possible to meet the constraints on each side. This is the default behavior.
  + wrap: Expands the view only as much as needed to fit its

contents, but still allows the view to be smaller than that if

the constraints require it. So the difference between this

and using Wrap Content (above), is that setting the width

to Wrap Content forces the width to always exactly match

the content width; whereas using Match Constraints with

layout\_constraintWidth\_default set to wrap also allows

the view to be smaller than the content width.

* layout\_constraintWidth\_min

This takes a dp dimension for the view's minimum width.

* layout\_constraintWidth\_max

This takes a dp dimension for the view's maximum width.

However, if the given dimension has only one constraint, then the view expands to fit its contents. Using this mode on either the height or width also allows you to set a size ratio.

1. Set size as a ratio

You can set the view size to a ratio such as 16:9 if at least one of the view dimensions is set to "match constraints" (0dp). To enable the ratio, click Toggle Aspect Ratio Constraint (callout 1 in figure 14), and then enter the width:height ratio in the input that appears.

If both the width and height are set to match constraints, you can click Toggle Aspect Ratio Constraint to select which dimension is based on a ratio of the other. The view inspector indicates which is set as a ratio by connecting the corresponding edges with a solid line.

For example, if you set both sides to "match constraints", click Toggle Aspect Ratio Constraint twice to set the width be a ratio of the height. Now the entire size is dictated by the height of the view (which can be defined in any way) as shown in figure 15.

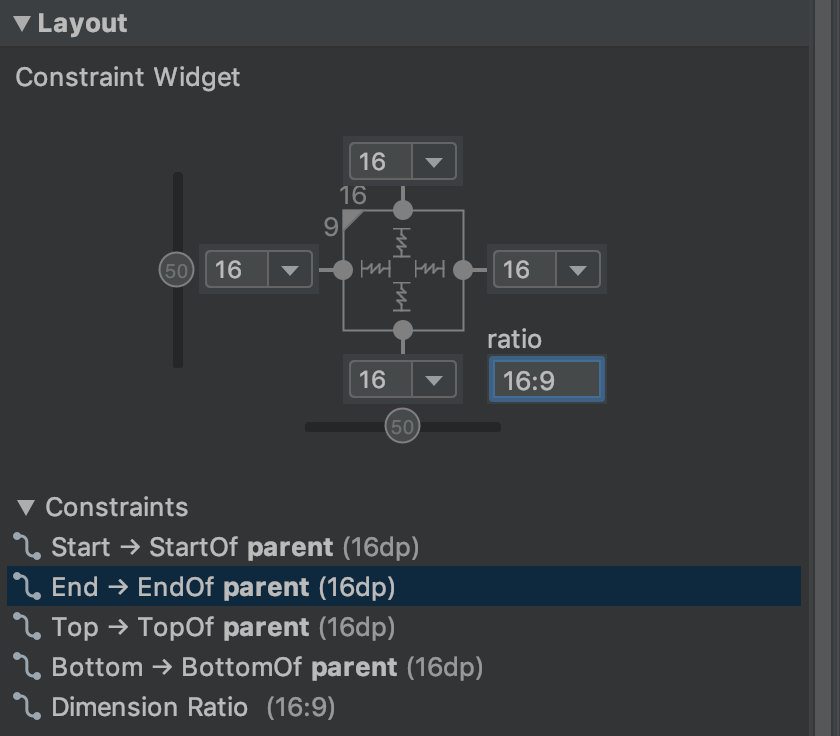


Figure 15. The view is set to a 16:9 aspect with the width based on a ratio of the height.

1. Adjust the view margins

To ensure that all your views are evenly spaced, click Margin in the toolbar to select the default margin for each view that you add to the layout. Any change you make to the default margin applies only to the views you add from then on.

You can control the margin for each view in the Attributes window by clicking the number on the line that represents each constraint (in figure 14, callout 4 shows the bottom margin is set to 16dp).



Figure 16. The toolbar's Margin button.

All margins offered by the tool are factors of 8dp to help your views align to Material Design's 8dp square grid recommendations.

1. Control linear groups with a chain

A chain is a group of views that are linked to each other with bi-directional position constraints. The views within a chain can be distributed either vertically or horizontally.

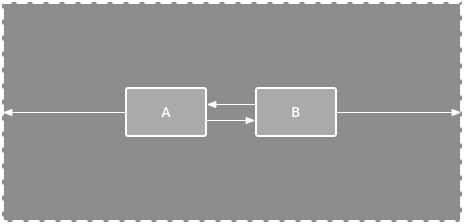


Figure 17. A horizontal chain with two views

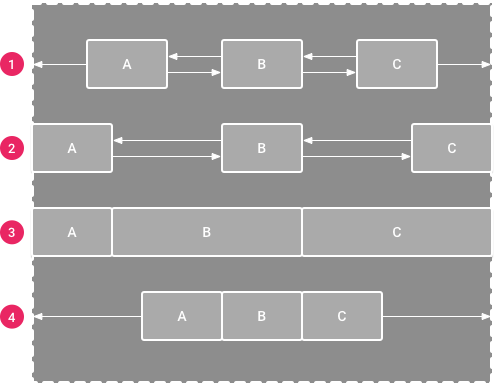


Figure 18. Examples of each chain style

1. Spread: The views are evenly distributed (after margins are accounted for). This is the default.
2. Spread inside: The first and last view are affixed to the constraints on each end of the chain and the rest are evenly distributed.
3. Weighted: When the chain is set to either spread or spread inside, you can fill the remaining space by setting one or more views to "match constraints" (0dp). By default, the space is evenly distributed between each view that's set to "match constraints," but you can assign a weight of importance to each view using the layout\_constraintHorizontal\_weight and layout\_constraintVertical\_weight attributes. If you're familiar with layout\_weight in a linear layout, this works the same way. So the view with the highest weight value gets the most amount of space; views that have the same weight get the same amount of space.
4. Packed: The views are packed together (after margins are accounted for). You can then adjust the whole chain's bias (left/right or up/down) by changing the chain's head view bias.

The chain's "head" view (the left-most view in a horizontal chain and the top-most view in a vertical chain) defines the chain's style in XML. However, you can toggle between spread, spread inside, and packed by selecting any view in the chain and then clicking the chain button that appears below the view.

To create a chain, select all of the views to be included in the chain, right-click one of the views, select Chains and then select either Center Horizontally or Center Vertically.

Here are a few other things to consider when using chains:

* A view can be a part of both a horizontal and a vertical chain, making it easy to build flexible grid layouts.
* A chain works properly only if each end of the chain is constrained to another object on the same axis, as shown in figure 14.
* Although the orientation of a chain is either vertical or horizontal, using one does not align the views in that direction. So be sure you include other constraints to achieve the proper position for each view in the chain, such as alignment constraints.

1. Automatically create constraints

nstead of adding constraints to every view as you place them in the layout, you can move each view into the positions you desire, and then click Infer Constraints to automatically create constraints.

Infer Constraints scans the layout to determine the most effective set of constraints for all views. It makes a best effort to constrain the views to their current positions while allowing flexibility. You might need to make some adjustments to be sure the layout responds as you intend for different screen sizes and orientations.

Autoconnect to parent is a separate feature that you can enable. If enabled, when you add child views to a parent, this feature automatically creates two or more constraints for each view as you add them to the layout, but only when it's appropriate to constrain the view to the parent layout. Autoconnect does not create constraints to other views in the layout.

Autoconnect is disabled by default. Enable it by clicking Enable Autoconnection to Parent in the Layout Editor toolbar.

1. Keyframe animations

Within a ConstraintLayout, you can animate changes to the size and position of elements by using ConstraintSet and TransitionManager.

A ConstraintSet is a lightweight object that represents the constraints, margins, and padding of all child elements within a ConstraintLayout. When you apply a ConstraintSet to a displayed ConstraintLayout, the layout updates the constraints of all of its children.

To build an animation using ConstraintSets, specify two layout files which act as a start and end keyframe for the animation. You can then load a ConstraintSet from the second keyframe file and apply it to the displayed ConstraintLayout.

The code example below shows how to animate moving a single button to the bottom of the screen.

// MainActivity.kt

fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.keyframe\_one)

constraintLayout = findViewById(R.id.constraint\_layout) // member variable

}

fun animateToKeyframeTwo() {

val constraintSet = ConstraintSet()

constraintSet.load(this, R.layout.keyframe\_two)

TransitionManager.beginDelayedTransition()

constraintSet.applyTo(constraintLayout)

}

// layout/keyframe1.xml

// Keyframe 1 contains the starting position for all elements in the animation as well as final colors and text sizes

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<Button

android:id="@+id/button2"

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:text="Button"

app:layout\_constraintEnd\_toEndOf="parent"

app:layout\_constraintStart\_toStartOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

// layout/keyframe2.xml

// Keyframe 2 contains another ConstraintLayout with the final positions

<?xml version="1.0" encoding="utf-8"?>

<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<Button

android:id="@+id/button2"

android:layout\_width="0dp"

android:layout\_height="wrap\_content"

android:text="Button"

app:layout\_constraintEnd\_toEndOf="parent"

app:layout\_constraintStart\_toStartOf="parent"

app:layout\_constraintBottom\_toBottomOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>

<https://developer.android.com/training/constraint-layout>

## Notifications Overview

A notification is a message that Android displays outside your app's UI to provide the user with reminders, communication from other people, or other timely information from your app. Users can tap the notification to open your app or take an action directly from the notification.

This page provides an overview of where notifications appear and the available features. If you want to start building your notifications, instead read Create a Notification.

For more information about the design and interaction patterns, see the Notifications design guide. Additionally, see the Android Notifications Sample for a demonstration of best practices in using the Notification.Style API in both mobile and wearable apps.

1. Appearances on a device

Notifications appear to users in different locations and formats, such as an icon in the status bar, a more detailed entry in the notification drawer, as a badge on the app's icon, and on paired wearables automatically.

**Status bar and notification drawer**

When you issue a notification, it first appears as an icon in the

status bar.

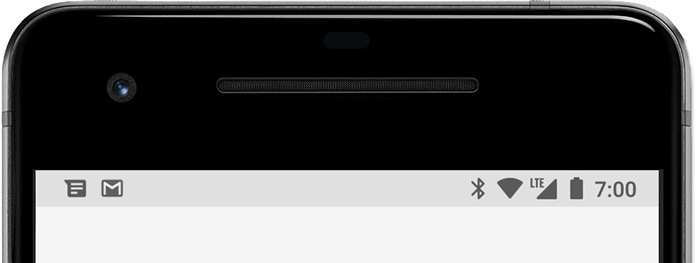


Figure 1. Notification icons appear on the left side of the status bar

Users can swipe down on the status bar to open the notification

drawer, where they can view more details and take actions with

the notification.

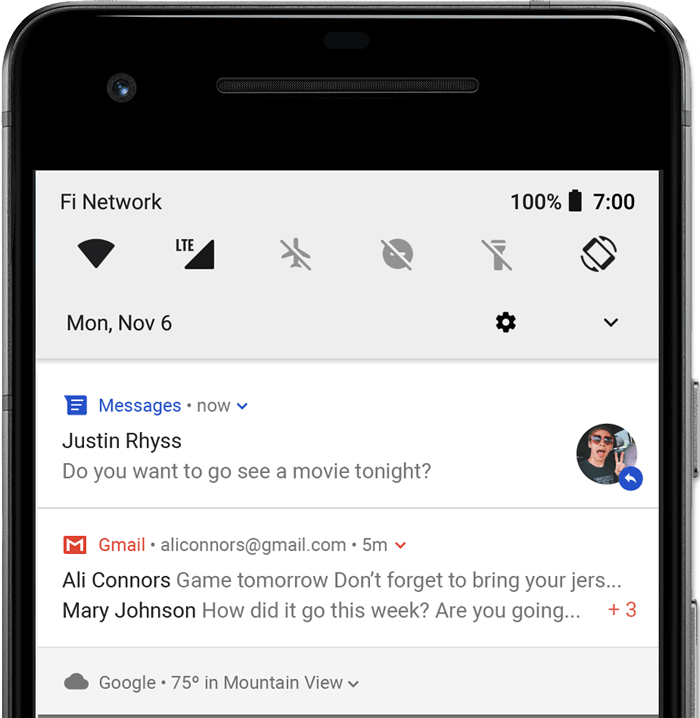


Figure 2. Notifications in the notification drawer

Users can drag down on a notification in the drawer to reveal the expanded view, which shows additional content and action buttons, if provided.

A notification remains visible in the notification drawer until dismissed by the app or the user.

**Heads-up notification**

Beginning with Android 5.0, notifications can briefly appear in a floating window called a *heads-up notification*. This behavior is normally for important notifications that the user should know about immediately, and it appears only if the device is unlocked.

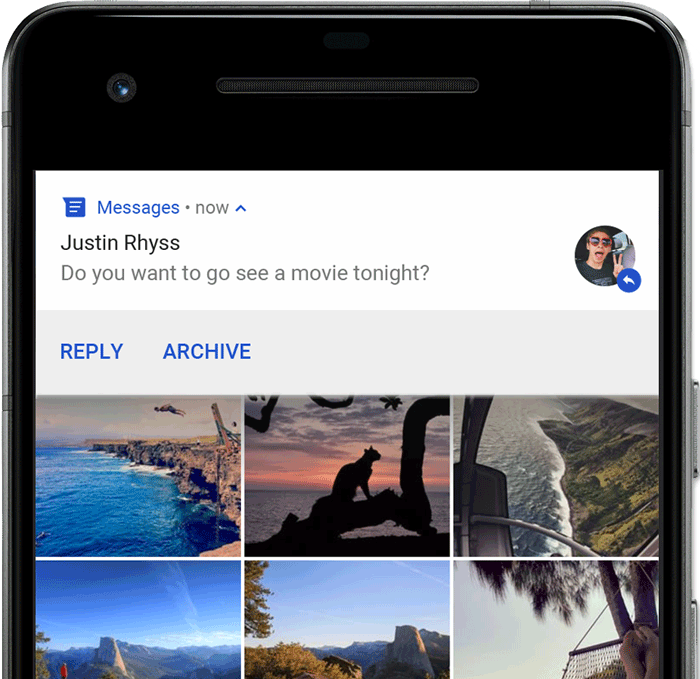


Figure 3. A heads-up notification appears in front of the foreground app

The heads-up notification appears the moment your app issues the notification and it disappears after a moment, but remains visible in the notification drawer as usual.

Example conditions that might trigger heads-up notifications include the following:

* The user's activity is in fullscreen mode (the app uses fullScreenIntent).
* The notification has high priority and uses ringtones or vibrations on devices running Android 7.1 (API level 25) and lower.
* The notification channel has high importance on devices running Android 8.0 (API level 26) and higher.

**Lock screen**

Beginning with Android 5.0, notifications can appear on the lock screen.

You can programmatically set the level of detail visible in notifications posted by your app on a secure lock screen, or even whether the notification will show on the lock screen at all.

Users can use the system settings to choose the level of detail visible in lock screen notifications, including the option to disable all lock screen notifications. Starting with Android 8.0, users can choose to disable or enable lock screen notifications for each notification channel.

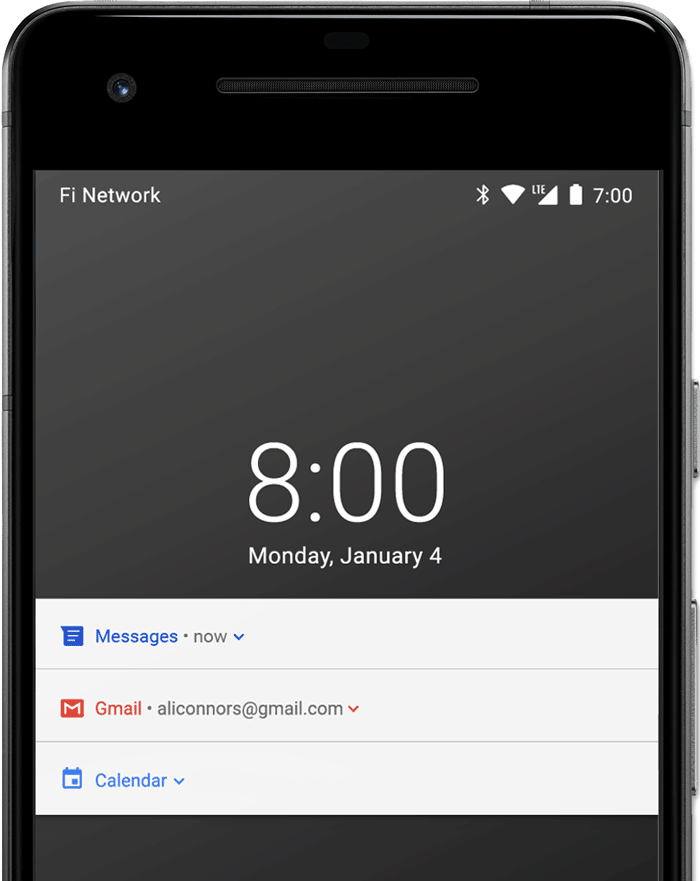


Figure 4. Notifications on the lock screen with sensitive content hidden

**App icon badge**

In supported launchers on devices running Android 8.0 (API level 26) and higher, app icons indicate new notifications with a colored "badge" (also known as a "notification dot") on the corresponding app launcher icon.

Users can long-press on an app icon to see the notifications for that app. Users can then dismiss or act on notifications from that menu, similar to the notification drawer.

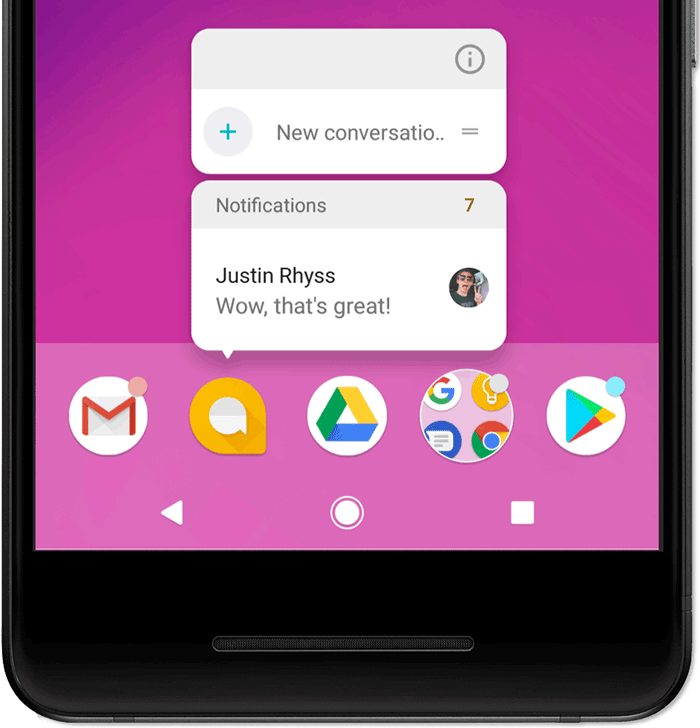


Figure 5. Notification badges and the long-press menu

# **WEAR OS DEVICES**

If the user has a paired Wear OS device, all your notifications appear there automatically, including expandable detail and action buttons.

You can also enhance the experience by customizing some appearances for the notification on wearables and provide different actions, including suggested replies and voice input replies.

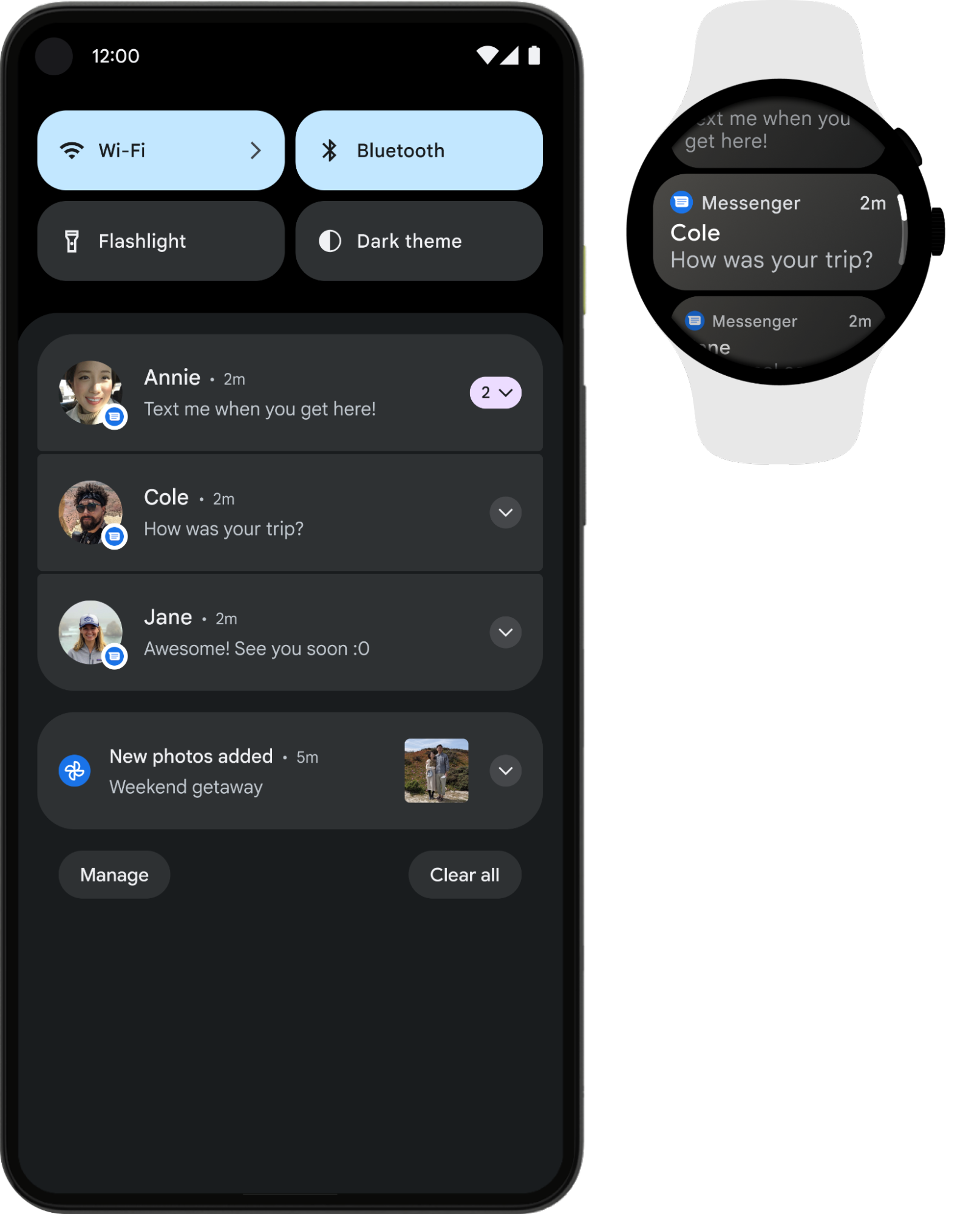
****

Figure 6. Notifications automatically appear on a paired Wear OS device

## Notification anatomy

The design of a notification is determined by system templates—your app simply defines the contents for each portion of the template. Some details of the notification appear only in the expanded view.

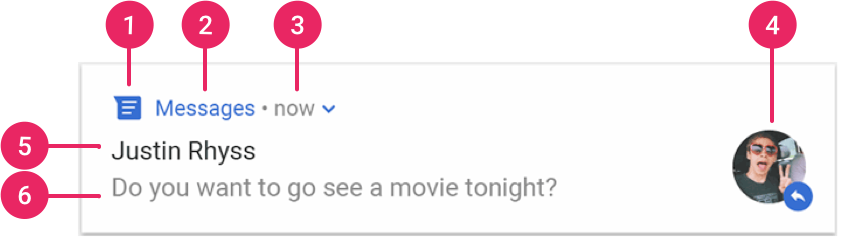


Figure 7. A notification with basic details

1. Small icon: This is required and set with setSmallIcon().
2. App name: This is provided by the system.
3. Time stamp: This is provided by the system but you can override with setWhen() or hide it with setShowWhen(false).
4. Large icon: This is optional (usually used only for contact photos; do not use it for your app icon) and set with setLargeIcon().
5. Title: This is optional and set with setContentTitle().
6. Text: This is optional and set with setContentText().

**Notification actions**

Although it's not required, every notification should open an appropriate app activity when tapped. In addition to this default notification action, you can add action buttons that complete an app-related task from the notification (often without opening an activity), as shown in figure 8.

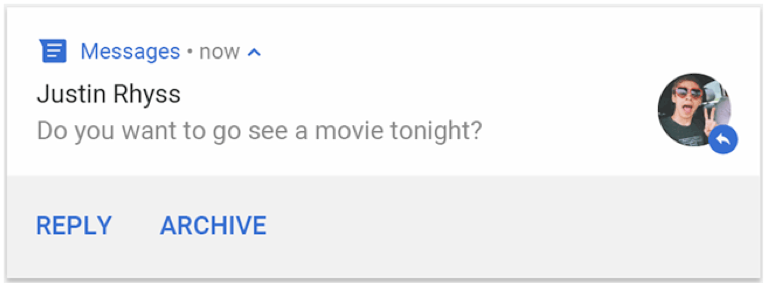


Figure 9. A notification with action buttons

Starting in Android 7.0 (API level 24), you can also add an action to reply to messages or enter other text directly from the notification.

Starting in Android 10 (API level 29), the platform can automatically generate action buttons with suggested intent-based actions.

Adding action buttons is explained further in Create a Notification.

**Expandable notification**

By default, the notification's text content is truncated to fit on one line. If you want your notification to be longer, you can enable a larger text area that's expandable by applying an additional template, as shown in figure 8.

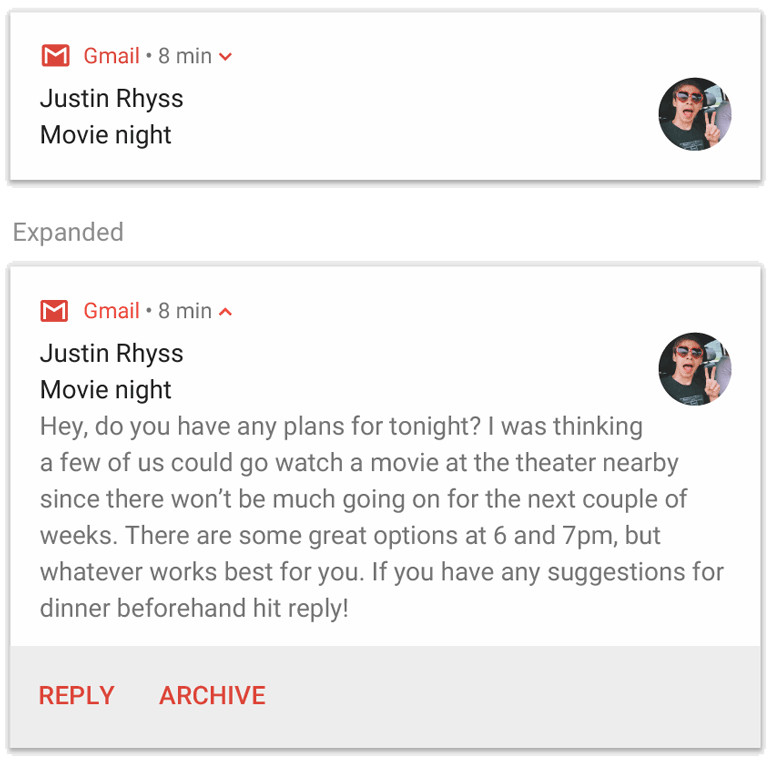


Figure 8. An expandable notification for large text

You can also create an expandable notification with an image, in inbox style, a chat conversation, or media playback controls.

And although we recommend you always use these templates to ensure proper design compatibility on all devices, if necessary, you can also create a custom notification layout.

## Notification updates and groups

To avoid bombarding your users with multiple or redundant notifications when you have additional updates, you should consider updating an existing notification rather than issuing a new one, or consider using the inbox-style notification to show conversation updates.

However, if it's necessary to deliver multiple notifications, you should consider grouping those separate notifications into a group (available on Android 7.0 and higher). A notification group allows you to collapse multiple notifications into just one post in the notification drawer, with a summary. The user can then expand the notification to reveal the details for each individual notification.

The user can progressively expand the notification group and each notification within it for more details.

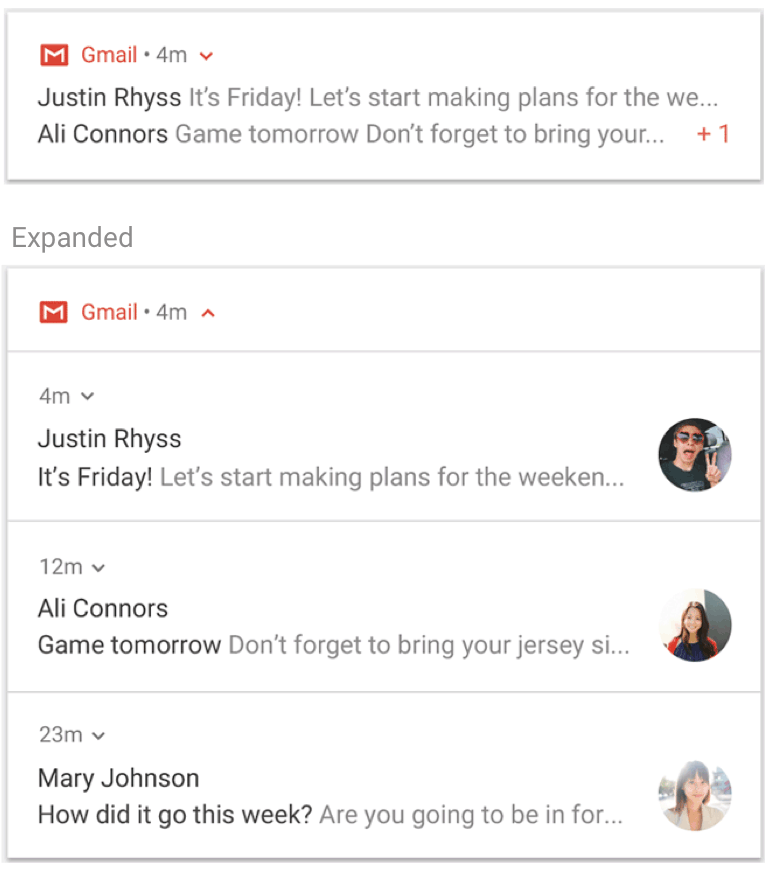


Figure 10. A collapsed and expanded notification group

## Notification channels

Starting in Android 8.0 (API level 26), all notifications must be assigned to a channel or it will not appear. By categorizing notifications into channels, users can disable specific notification channels for your app (instead of disabling all your notifications), and users can control the visual and auditory options for each channel—all from the Android system settings (figure 11). Users can also long-press a notification to change behaviors for the associated channel.

On devices running Android 7.1 (API level 25) and lower, users can manage notifications on a per-app basis only (effectively each app only has one channel on Android 7.1 and lower).

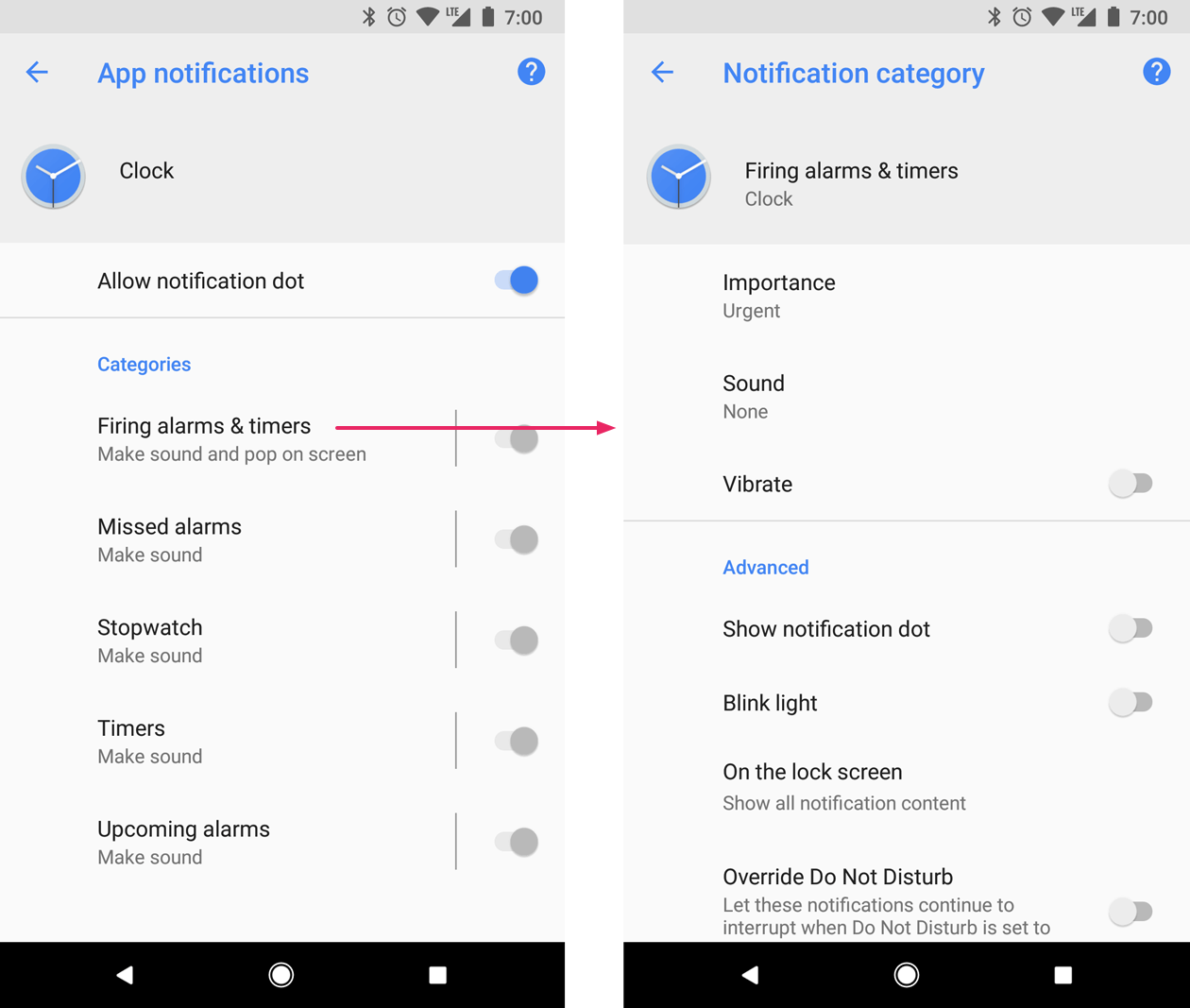


Figure 11. Notification settings for Clock app and one of its channels

One app can have multiple notification channels—a separate channel for each type of notification the app issues. An app can also create notification channels in response to choices made by users of your app. For example, you may set up separate notification channels for each conversation group created by a user in a messaging app.

The channel is also where you specify the importance level for your notifications on Android 8.0 and higher. So all notifications posted to the same notification channel have the same behavior.

## Notification importance

Android uses the importance of a notification to determine how much the notification should interrupt the user (visually and audibly). The higher the importance of a notification, the more interruptive the notification will be.

On Android 8.0 (API level 26) and above, importance of a notification is determined by the importance of the channel the notification was posted to. Users can change the importance of a notification channel in the system settings (figure 12). On Android 7.1 (API level 25) and below, importance of each notification is determined by the notification's priority.

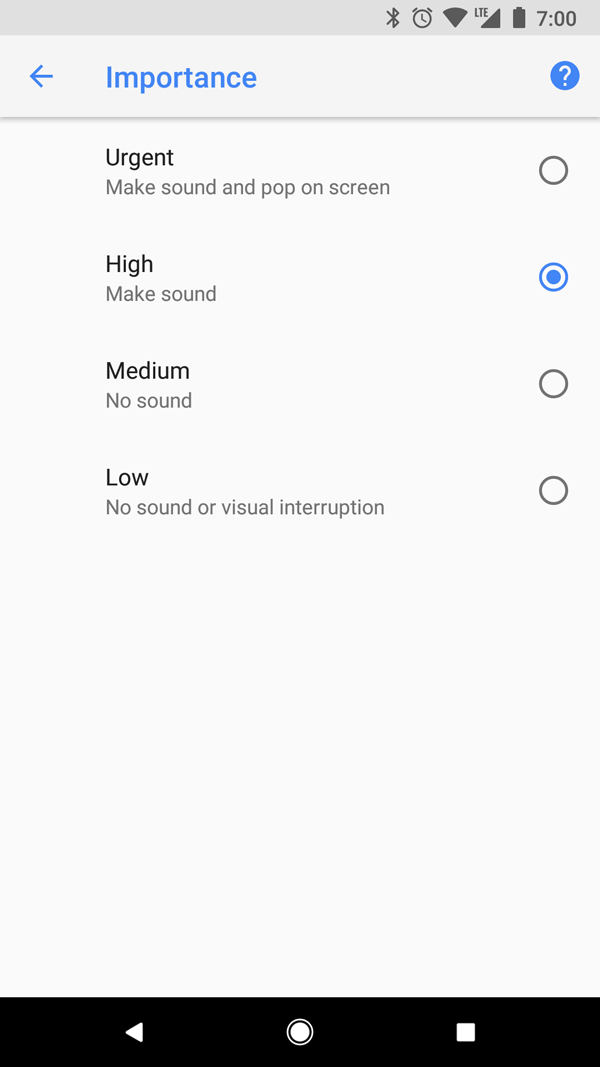


Figure 12. Users can change the importance of each channel on Android 8.0 and higher

The possible importance levels are the following:

* Urgent: Makes a sound and appears as a heads-up notification.
* High: Makes a sound.
* Medium: No sound.
* Low: No sound and does not appear in the status bar.

All notifications, regardless of importance, appear in non-interruptive system UI locations, such as in the notification drawer and as a badge on the launcher icon (though you can modify the appearance of the notification badge).

## Do Not Disturb mode

Starting in Android 5.0 (API level 21), users can enable Do Not Disturb mode, which silences sounds and vibration for all notifications. Notifications still appear in the system UI as normal, unless the user specifies otherwise.

There are three different levels available in Do Not Disturb mode:

* Total silence: blocks all sounds and vibrations, including from alarms, music, videos, and games.
* Alarms only: blocks all sounds and vibrations, except from alarms.
* Priority only: users can configure which system-wide categories can interrupt them (such as only alarms, reminders, events, calls, or messages). For messages and calls, users can also choose to filter based on who the sender or caller is (figure 13).

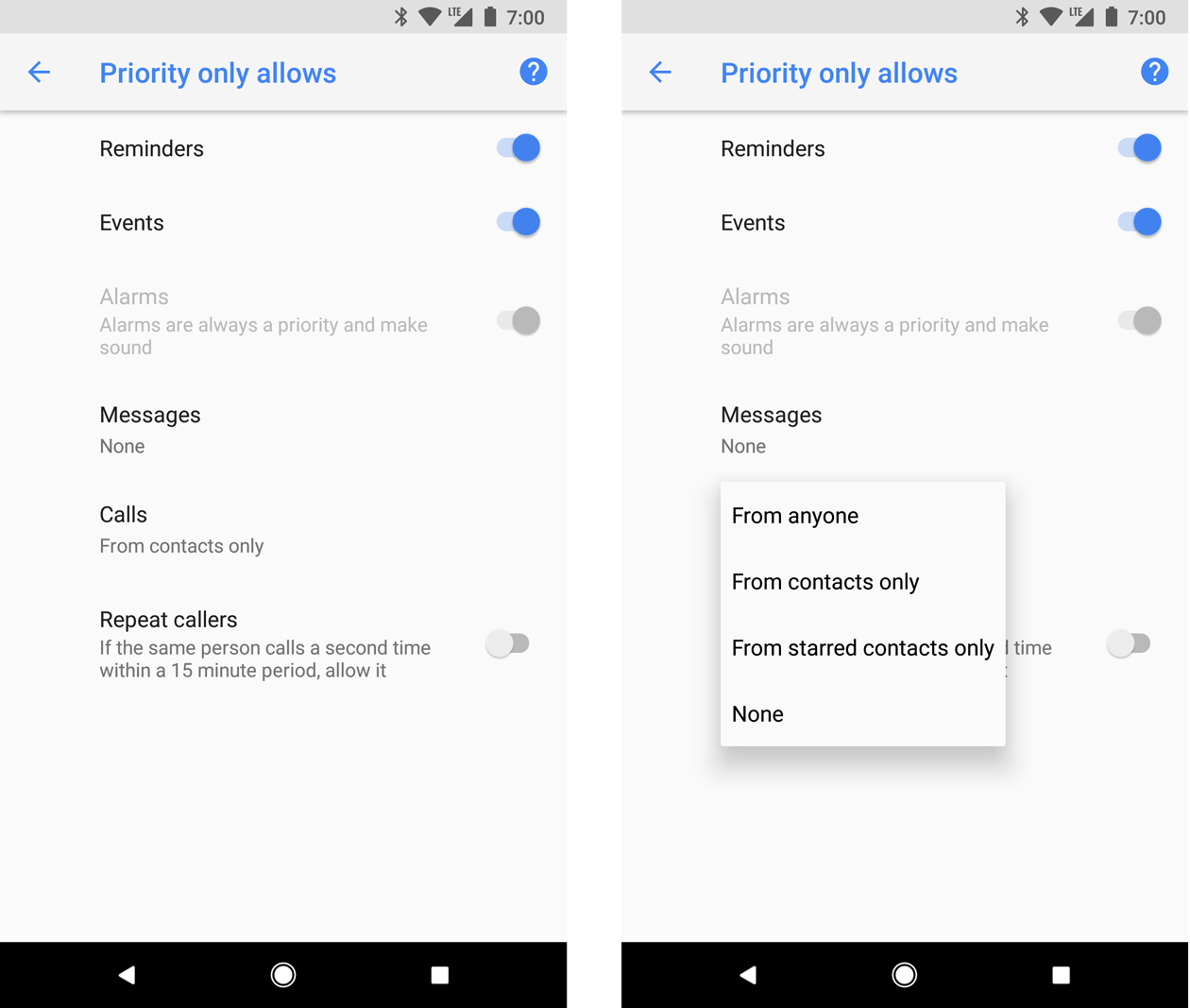


Figure 13. Users can can allow notifications through based on system-wide categories (left) and based on who sent a message or who is calling (right).

On Android 8.0 (API level 26) and above, users can additionally allow notifications through for app-specific categories (also known as channels) by overriding Do Not Disturb on a channel-by-channel basis. For example, a payment app might have channels for notifications related to withdrawals and deposits. The user can then choose to allow either withdrawal notifications, deposit notifications, or both when in priority mode. On devices running Android 7.1 (API level 25) and below, users can allow notifications through on an app by app basis, rather than on a channel by channel basis.

To properly configure your notifications for these user settings, you must set a system-wide category and channel.

## Notifications for foreground services.

A notification is required when your app is running a "foreground service"—a Service running in the background that's long living and noticeable to the user, such as a media player. This notification cannot be dismissed like other notifications. To remove the notification, the service must be either stopped or removed from the "foreground" state.

For more information read Running a service in the foreground. And if you are building a media player, also read Using MediaStyle notifications with a foreground service.

1. Posting limits

Beginning with Android 8.1 (API level 27), apps cannot make a notification sound more than once per second. If your app posts multiple notifications in one second, they all appear as expected, but only the first notification per second makes a sound.

However, Android also applies a rate limit when updating a notification. If you post updates to a single notification too frequently (many in less than one second), the system might drop some updates.

1. Notification compatibility

Since Android 1.0, the notification system UI and the notification-related APIs have continually evolved. To use the latest notification API features while still supporting older devices, use the support library notification API: NotificationCompat and its subclasses, as well as NotificationManagerCompat. This will allow you to avoid writing conditional code to check API levels because these APIs handle that for you.

NotificationCompat is updated as the platform evolves to include the latest methods. It is important to note that the availability of a method in NotificationCompat does not guarantee that the corresponding feature will be provided on older devices. In some cases calling a newly-introduced API results in a no-op on older devices. For example, NotificationCompat.addAction() displays the action button on a device running Android 4.1 (API level 16) and higher only.

The following is a summary of the most notable behavior changes for Android notifications.

**Android 4.1, API level 16**

* Introduced expandable notification templates (called notification styles), allowing for larger notification content area to display information. Users can use a one finger swipe up/down gesture to expand a notification.
* Also introduced the ability to add additional actions, in the form of buttons, to a notification.
* Added ability for users to turn notifications off on a per-app basis in settings.

**Android 4.4, API level 19 and 20**

* Notification listener services were added to the API.
* Android Wear (now called Wear OS) support was added in API level 20.

**Android 5.0, API level 21**

* Introduced lock screen and heads-up notifications.
* The user can now put the phone into Do Not Disturb mode and configure which notifications are allowed to interrupt them when the device is in priority only mode.
* Methods added to API set whether or not a notification is displayed on the lock screen (setVisibility()) and for specifying “public” version of the notification text.
* setPriority() method added which tells the system how “interruptive” this notification should be (e.g. setting it to high makes the notification appear as a heads-up notification).
* Notification stacks support added to Android Wear (now called Wear OS) devices. Put notifications into a stack using setGroup(). Note that notification stacks were not supported on tablets nor phones yet. Notification stacks would later become known as a group or bundle.

**Android 7.0, API level 24**

* Notification templates were restyled to put emphasis on the hero image and avatar.
* Three notification templates were added: one for messaging apps and the other two for decorating custom content views with the expandable affordance and other system decorations.
* Support added to handheld devices (phones and tablets) for notification groups. Uses the same API as Android Wear (now called Wear OS) notification stacks introduced in Android 5.0 (API level 21).
* Users can reply directly inside of a notification (they can enter text which will then be routed to the notification’s parent app) using inline reply.

**Android 8.0, API level 26**

* Individual notifications must now be put in a specific channel.
* Users can now turn off notifications per channel, instead of turning off all notifications from an app.
* Apps with active notifications display a notification "badge" on top of their app icon on the home/launcher screen.
* Users can now snooze a notification from the drawer. You can set an automatic timeout for a notification.
* You can also set the notification's background color.
* Some APIs regarding notification behaviors were moved from Notification to NotificationChannel. For example, use NotificationChannel.setImportance() instead of NotificationCompat.Builder.setPriority() for Android 8.0 and higher.

<https://developer.android.com/guide/topics/ui/notifiers/notifications>

* 1. Create a Notification

Notifications provide short, timely information about events in your app while it's not in use. This page teaches you how to create a notification with various features for Android 4.0 (API level 14) and higher. For an introduction to how notifications appear on Android, see the Notifications Overview. For sample code that uses notifications, see the Android Notifications Sample.

Notice that the code on this page uses the NotificationCompat APIs from the Android support library. These APIs allow you to add features available only on newer versions of Android while still providing compatibility back to Android 4.0 (API level 14). However, some new features such as the inline reply action result in a no-op on older versions.

**Add the support library**

Although most projects created with Android Studio include the necessary dependencies to use NotificationCompat, you should verify that your module-level build.gradle file includes the following dependency:

dependencies {

implementation("com.android.support:support-compat:28.0.0")

}

**Create a basic notification**

A notification in its most basic and compact form (also known as collapsed form) displays an icon, a title, and a small amount of content text. In this section, you'll learn how to create a notification that the user can click on to launch an activity in your app.

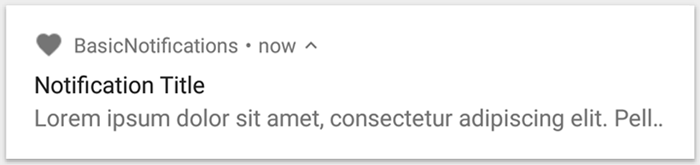


Figure 1. A notification with a title and text

**Set the notification content**

To get started, you need to set the notification's content and channel using a NotificationCompat.Builder object. The following example shows how to create a notification with the following:

* A small icon, set by setSmallIcon(). This is the only user-visible content that's required.
* A title, set by setContentTitle().
* The body text, set by setContentText().
* The notification priority, set by setPriority(). The priority determines how intrusive the notification should be on Android 7.1 and lower. (For Android 8.0 and higher, you must instead set the channel importance—shown in the next section.)

var builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle(textTitle)

.setContentText(textContent)

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

Notice that the NotificationCompat.Builder constructor requires that you provide a channel ID. This is required for compatibility with Android 8.0 (API level 26) and higher, but is ignored by older versions.

By default, the notification's text content is truncated to fit one line. If you want your notification to be longer, you can enable an expandable notification by adding a style template with setStyle(). For example, the following code creates a larger text area:

var builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle("My notification")

.setContentText("Much longer text that cannot fit one line...")

.setStyle(NotificationCompat.BigTextStyle()

.bigText("Much longer text that cannot fit one line..."))

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

**Create a channel and set the importance**

Before you can deliver the notification on Android 8.0 and higher, you must register your app's notification channel with the system by passing an instance of NotificationChannel to createNotificationChannel(). So the following code is blocked by a condition on the SDK\_INT version:

private fun createNotificationChannel() {

// Create the NotificationChannel, but only on API 26+ because

// the NotificationChannel class is new and not in the support library

if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.O) {

val name = getString(R.string.channel\_name)

val descriptionText = getString(R.string.channel\_description)

val importance = NotificationManager.IMPORTANCE\_DEFAULT

val channel = NotificationChannel(CHANNEL\_ID, name, importance).apply {

description = descriptionText

}

// Register the channel with the system

val notificationManager: NotificationManager =

getSystemService(Context.NOTIFICATION\_SERVICE) as NotificationManager

notificationManager.createNotificationChannel(channel)

}

}

Because you must create the notification channel before posting any notifications on Android 8.0 and higher, you should execute this code as soon as your app starts. It's safe to call this repeatedly because creating an existing notification channel performs no operation.

Notice that the NotificationChannel constructor requires an importance, using one of the constants from the NotificationManager class. This parameter determines how to interrupt the user for any notification that belongs to this channel—though you must also set the priority with setPriority() to support Android 7.1 and lower (as shown above).

Although you must set the notification importance/priority as shown here, the system does not guarantee the alert behavior you'll get. In some cases the system might change the importance level based other factors, and the user can always redefine what the importance level is for a given channel.

For more information about what the different levels mean, read about notification importance levels.

**Set the notification's tap action**

Every notification should respond to a tap, usually to open an activity in your app that corresponds to the notification. To do so, you must specify a content intent defined with a PendingIntent object and pass it to setContentIntent().

The following snippet shows how to create a basic intent to open an activity when the user taps the notification:

// Create an explicit intent for an Activity in your app

val intent = Intent(this, AlertDetails::class.java).apply {

flags = Intent.FLAG\_ACTIVITY\_NEW\_TASK or Intent.FLAG\_ACTIVITY\_CLEAR\_TASK

}

val pendingIntent: PendingIntent = PendingIntent.getActivity(this, 0, intent, 0)

val builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle("My notification")

.setContentText("Hello World!")

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

// Set the intent that will fire when the user taps the notification

.setContentIntent(pendingIntent)

.setAutoCancel(true)

Notice this code calls setAutoCancel(), which automatically removes the notification when the user taps it.

The setFlags() method shown above helps preserve the user's expected navigation experience after they open your app via the notification. But whether you want to use that depends on what type of activity you're starting, which may be one of the following:

An activity that exists exclusively for responses to the notification. There's no reason the user would navigate to this activity during normal app use, so the activity starts a new task instead of being added to your app's existing task and back stack. This is the type of intent created in the sample above.

An activity that exists in your app's regular app flow. In this case, starting the activity should create a back stack so that the user's expectations for the Back and Up buttons is preserved.

For more about the different ways to configure your notification's intent, read Start an Activity from a Notification.

**Show the notification**

Tomake the notification appear, call NotificationManagerCompat.notify(), passing it a unique ID for the notification and the result of NotificationCompat.Builder.build(). For example:

with(NotificationManagerCompat.from(this)) {

// notificationId is a unique int for each notification that you must define

notify(notificationId, builder.build())

}

**Add action buttons**

A notification can offer up to three action buttons that allow the user to respond quickly, such as snooze a reminder or even reply to a text message. But these action buttons should not duplicate the action performed when the user taps the notification.

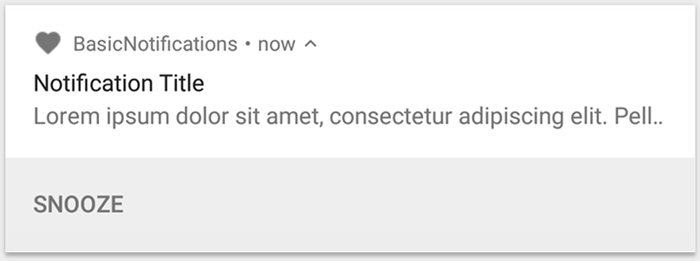


Figure 2. A notification with one action button

To add an action button, pass a PendingIntent to the addAction() method. This is just like setting up the notification's default tap action, except instead of launching an activity, you can do a variety of other things such as start a BroadcastReceiver that performs a job in the background so the action does not interrupt the app that's already open.

For example, the following code shows how to send a broadcast to a specific receiver:

val snoozeIntent = Intent(this, MyBroadcastReceiver::class.java).apply {

action = ACTION\_SNOOZE

putExtra(EXTRA\_NOTIFICATION\_ID, 0)

}

val snoozePendingIntent: PendingIntent =

PendingIntent.getBroadcast(this, 0, snoozeIntent, 0)

val builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle("My notification")

.setContentText("Hello World!")

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

.setContentIntent(pendingIntent)

.addAction(R.drawable.ic\_snooze, getString(R.string.snooze),

snoozePendingIntent)

**Add a direct reply action**

The direct reply action, introduced in Android 7.0 (API level 24), allows users to enter text directly into the notification, which is delivered to your app without opening an activity. For example, you can use a direct reply action to let users reply to text messages or update task lists from within the notification.

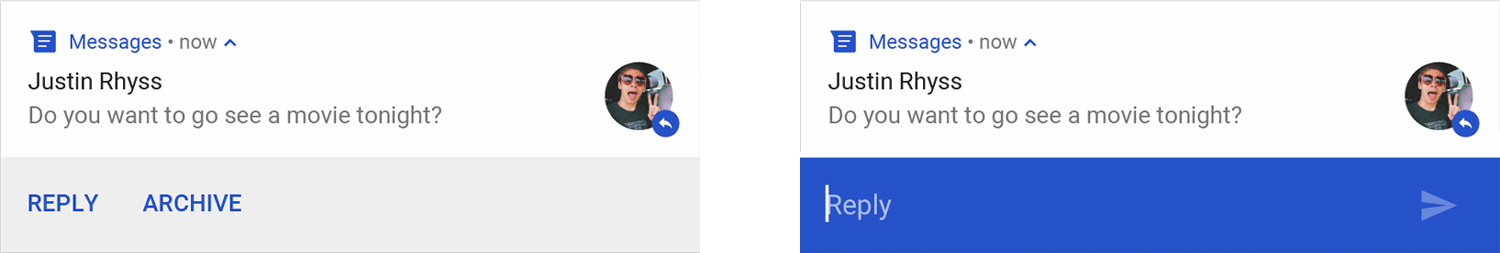


Figure 3. Tapping the "Reply" button opens the text input

The direct reply action appears as an additional button in the notification that opens a text input. When the user finishes typing, the system attaches the text response to the intent you had specified for the notification action and sends the intent to your app.

**Add the reply button**

To create a notification action that supports direct reply:

1. Create an instance of RemoteInput.Builder that you can add to your notification action. This class's constructor accepts a string that the system uses as the key for the text input. Later, your handheld app uses that key to retrieve the text of the input.

// Key for the string that's delivered in the action's intent.

private val KEY\_TEXT\_REPLY = "key\_text\_reply"

var replyLabel: String = resources.getString(R.string.reply\_label)

var remoteInput: RemoteInput = RemoteInput.Builder(KEY\_TEXT\_REPLY).run {

setLabel(replyLabel)

build()

}

1. Create a PendingIntent for the reply action.

// Build a PendingIntent for the reply action to trigger.

var replyPendingIntent: PendingIntent =

PendingIntent.getBroadcast(applicationContext,

conversation.getConversationId(),

getMessageReplyIntent(conversation.getConversationId()),

PendingIntent.FLAG\_UPDATE\_CURRENT)

1. Attach the RemoteInput object to an action using addRemoteInput().

// Create the reply action and add the remote input.

var action: NotificationCompat.Action =

NotificationCompat.Action.Builder(R.drawable.ic\_reply\_icon,

getString(R.string.label), replyPendingIntent)

.addRemoteInput(remoteInput)

.build()

1. Apply the action to a notification and issue the notification.

// Build the notification and add the action.

val newMessageNotification = Notification.Builder(context, CHANNEL\_ID)

.setSmallIcon(R.drawable.ic\_message)

.setContentTitle(getString(R.string.title))

.setContentText(getString(R.string.content))

.addAction(action)

.build()

// Issue the notification.

with(NotificationManagerCompat.from(this)) {

notificationManager.notify(notificationId, newMessageNotification)

}

Retrieve user input from the reply

To receive user input from the notification's reply UI, call RemoteInput.getResultsFromIntent(), passing it the Intent received by your BroadcastReceiver:

private fun getMessageText(intent: Intent): CharSequence? {

return RemoteInput.getResultsFromIntent(intent)?.getCharSequence(KEY\_TEXT\_REPLY)

}

After you’ve processed the text, you must update the notification by calling NotificationManagerCompat.notify() with the same ID and tag (if used). This is necessary to hide direct reply UI and confirm to the user that their reply was received and processed correctly.

// Build a new notification, which informs the user that the system

// handled their interaction with the previous notification.

val repliedNotification = Notification.Builder(context, CHANNEL\_ID)

.setSmallIcon(R.drawable.ic\_message)

.setContentText(getString(R.string.replied))

.build()

// Issue the new notification.

NotificationManagerCompat.from(this).apply {

notificationManager.notify(notificationId, repliedNotification)

}

When working with this new notification, use the context that's passed to the receiver's onReceive() method.

You should also append the reply to the bottom of the notification by calling setRemoteInputHistory(). However, if you’re building a messaging app, you should create a messaging-style notification and append the new message to the conversation.

For more advice for notifications from a messaging apps, see best practices for messaging apps.

**Add a progress bar**

Notifications can include an animated progress indicator that shows users the status of an ongoing operation.

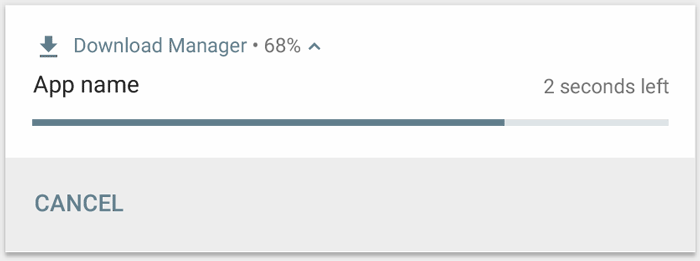


Figure 4. The progress bar during and after the operation.

If you can estimate how much of the operation is complete at any time, use the "determinate" form of the indicator (as shown in figure 4) by calling setProgress(max, progress, false). The first parameter is what the "complete" value is (such as 100); the second is how much is currently complete, and the last indicates this is a determinate progress bar.

As your operation proceeds, continuously call setProgress(max, progress, false) with an updated value for progress and re-issue the notification.

val builder = NotificationCompat.Builder(this, CHANNEL\_ID).apply {

setContentTitle("Picture Download")

setContentText("Download in progress")

setSmallIcon(R.drawable.ic\_notification)

setPriority(NotificationCompat.PRIORITY\_LOW)

}

val PROGRESS\_MAX = 100

val PROGRESS\_CURRENT = 0

NotificationManagerCompat.from(this).apply {

// Issue the initial notification with zero progress

builder.setProgress(PROGRESS\_MAX, PROGRESS\_CURRENT, false)

notify(notificationId, builder.build())

// Do the job here that tracks the progress.

// Usually, this should be in a

// worker thread

// To show progress, update PROGRESS\_CURRENT and update the notification with:

// builder.setProgress(PROGRESS\_MAX, PROGRESS\_CURRENT, false);

// notificationManager.notify(notificationId, builder.build());

// When done, update the notification one more time to remove the progress bar

builder.setContentText("Download complete")

.setProgress(0, 0, false)

notify(notificationId, builder.build())

}

At the end of the operation, progress should equal max. You can either leave the progress bar showing when the operation is done, or remove it. In either case, remember to update the notification text to show that the operation is complete. To remove the progress bar, call setProgress(0, 0, false).

To display an indeterminate progress bar (a bar that does not indicate percentage complete), call setProgress(0, 0, true). The result is an indicator that has the same style as the progress bar above, except the progress bar is a continuous animation that does not indicate completion. The progress animation runs until you call setProgress(0, 0, false) and then update the notification to remove the activity indicator.

Remember to change the notification text to indicate that the operation is complete.

**Set a system-wide category**

Android uses some pre-defined system-wide categories to determine whether to disturb the user with a given notification when the user has enabled Do Not Disturb mode.

If your notification falls into one of the pre-defined notification categories defined in NotificationCompat—such as CATEGORY\_ALARM,CATEGORY\_REMINDER, CATEGORY\_EVENT, or CATEGORY\_CALL—you should declare it as such by passing the appropriate category to setCategory().

var builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle("My notification")

.setContentText("Hello World!")

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

.setCategory(NotificationCompat.CATEGORY\_MESSAGE)

This information about your notification category is used by the system to make decisions about displaying your notification when the device is in Do Not Disturb mode.

However, you are not required to set a system-wide category and should only do so if your notifications match one of the categories defined by in NotificationCompat.

**Show an urgent message**

Your app might need to display an urgent, time-sensitive message, such as an incoming phone call or a ringing alarm. In these situations, you can associate a full-screen intent with your notification. When the notification is invoked, users see one of the following, depending on the device's lock status:

If the user's device is locked, a full-screen activity appears, covering the lockscreen.

If the user's device is unlocked, the notification appears in an expanded form that includes options for handling or dismissing the notification.

The following code snippet demonstrates how to associate your notification with a full-screen intent:

val fullScreenIntent = Intent(this, ImportantActivity::class.java)

val fullScreenPendingIntent = PendingIntent.getActivity(this, 0,

fullScreenIntent, PendingIntent.FLAG\_UPDATE\_CURRENT)

var builder = NotificationCompat.Builder(this, CHANNEL\_ID)

.setSmallIcon(R.drawable.notification\_icon)

.setContentTitle("My notification")

.setContentText("Hello World!")

.setPriority(NotificationCompat.PRIORITY\_DEFAULT)

.setFullScreenIntent(fullScreenPendingIntent, true)

**Set lock screen visibility**

To control the level of detail visible in the notification from the lock screen, call 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.

When VISIBILITY\_PRIVATE is set, you can also provide an alternate version of the notification content which hides certain details. For example, an SMS app might display a notification that shows You have 3 new text messages, but hides the message contents and senders. To provide this alternative notification, first create the alternative notification with NotificationCompat.Builder as usual. Then attach the alternative notification to the normal notification with setPublicVersion().

However, the user always has final control over whether their notifications are visible on the lock screen and can even control that based on your app's notification channels.

**Update a notification**

To update this notification after you've issued it, call NotificationManagerCompat.notify() again, passing it a notification with the same ID you used previously. If the previous notification has been dismissed, a new notification is created instead.

You can optionally call setOnlyAlertOnce() so your notification interupts the user (with sound, vibration, or visual clues) only the first time the notification appears and not for later updates.

**Remove a notification**

Notifications remain visible until one of the following happens:

* The user dismisses the notification.
* The user clicks the notification, and you called setAutoCancel() when you created the notification.
* You call cancel() for a specific notification ID. This method also deletes ongoing notifications.
* You call cancelAll(), which removes all of the notifications you previously issued.
* If you set a timeout when creating a notification using setTimeoutAfter(), the system cancels the notification after the specified duration elapses. If required, you can cancel a notification before the specified timeout duration elapses.

**Best practices for messaging apps**

Use the best practices listed here as a quick reference of what to keep in mind when creating notifications for your messaging and chat apps.

**Use MessagingStyle**

Starting in Android 7.0 (API level 24), Android provides a notification style template specifically for messaging content. Using the NotificationCompat.MessagingStyle class, you can change several of the labels displayed on the notification, including the conversation title, additional messages, and the content view for the notification.

The following code snippet demonstrates how to customize a notification's style using the MessagingStyle class.

var notification = NotificationCompat.Builder(this, CHANNEL\_ID)

.setStyle(NotificationCompat.MessagingStyle("Me")

.setConversationTitle("Team lunch")

.addMessage("Hi", timestamp1, null) // Pass in null for user.

.addMessage("What's up?", timestamp2, "Coworker")

.addMessage("Not much", timestamp3, null)

.addMessage("How about lunch?", timestamp4, "Coworker"))

.build()

Starting in Android 8.0 (API level 26), notifications that use the NotificationCompat.MessagingStyle class display more content in their collapsed form. You can also use the addHistoricMessage() method to provide context to a conversation by adding historic messages to messaging-related notifications.

When using NotificationCompat.MessagingStyle:

* Call MessagingStyle.setConversationTitle() to set a title for group chats with more than two people. A good conversation title might be the name of the group chat or, if it doesn't have a specific name, a list of the participants in the conversation. Without this, the message may be mistaken as belonging to a one-to-one conversation with the sender of the most recent message in the conversation.
* Use the MessagingStyle.setData() method to include media messages such as images. MIME types, of the pattern image/\* are currently supported.

**Use direct reply**

Direct Reply allows a user to reply inline to a message.

* After a user replies with the inline reply action, use MessagingStyle.addMessage() to update the MessagingStyle notification and do not retract or cancel the notification. Not cancelling the notification allows a user to send multiple replies from the notification.
* To make the inline reply action compatible with Wear OS, call Action.WearableExtender.setHintDisplayInlineAction(true).
* Use the addHistoricMessage() method to provide context to a direct reply conversation by adding historic messages to the notification.

**Enable smart reply**

* To enable Smart Reply, call setAllowGeneratedResponses(true) on the reply action. This causes Smart Reply responses to be available to users when the notification is bridged to a Wear OS device. Smart Reply responses are generated by an entirely on-watch machine learning model using the context provided by the NotificationCompat.MessagingStyle notification, and no data is uploaded to the Internet to generate the responses.

**Add notification metadata**

* Assign notification metadata to tell the system how to handle your app notifications when the device is in Do Not Disturb mode. For example, use the addPerson() or setCategory(Notification.CATEGORY\_MESSAGE) method to override the Do Not Disturb mode.

<https://developer.android.com/training/notify-user/build-notification>

**QUESTIONS:**

1. Explain the mobile application.

**Answer:**

* 1. **A type of software application that is intended to run specifically on mobile devices.**

**REFERENCE:**

1. [https://www.techopedia.com/definition/2953/mobile-application-mobile-app\](https://www.techopedia.com/definition/2953/mobile-application-mobile-app%5C)
2. <https://clevertap.com/blog/types-of-mobile-apps/>
3. <https://www.lifewire.com/what-is-ios-1994355>
4. <https://recombu.com/mobile/article/what-is-ios-and-what-does-ios-stand-for>
5. <https://www.androidauthority.com/android-app-development-1128595/>
6. <https://semaphoreci.com/blog/what-is-react-native>
7. <https://www.educative.io/blog/flutter-tutorial>
8. <https://dev.to/sm0ke/firebase-short-introduction-for-beginners-24d8>
9. <https://xd.adobe.com/ideas/principles/>
10. <https://www.getcloudapp.com/blog/ui-design>