Android java notification

Continue

Android java notification sound. Android java notification priority. Android java notification service. Android java notification show. Android java notificationmanager. Android java notification not showing. Android java notification icon. Android java notification example.

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 People 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: val core_version = "1.6.0" dependencies { implementation "androidx.core:core:$core_version" } val core_version = "1.6.0" dependencies { implementation("androidx.core:core-ktx:$core_version") } Note: Other libraries in the androidx group also include core as a transitive dependency. So if you're already using other Jetpack APIs, you might have access to NotificationCompat without requiring the exact dependency shown above. 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 For more details about each part of a notification, read about the notification anatomy. 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) NotificationCompat.Builder builder = new 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) NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID) .setSmallIcon(R.drawable.notification_icon) .setContentTitle("My notification") .setContentText("Much longer text that cannot fit one line...") .setStyle(new NotificationCompat.BigTextStyle() .bigText("Much longer text that cannot fit one line...")) .setPriority(NotificationCompat.PRIORITY_DEFAULT); For more information about other large notification styles, including how to add an image and media playback controls, see Create a Notification with Expandable Detail. 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) } } private void 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) { CharSequence name = getString(R.string.channel_name); String description = getString(R.string.channel_description); int importance = NotificationManager.IMPORTANCE_DEFAULT; NotificationChannel channel = new NotificationChannel(CHANNEL_ID, name, importance); channel.setDescription(description); // Register the channel with the system; you can't change the importance // or other notification behaviors after this NotificationManager notificationManager = getSystemService(NotificationManager.class); 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, PendingIntent.FLAG_IMMUTABLE) 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) // Create an explicit intent for an Activity in your app Intent intent = new Intent(this, AlertDetails.class); intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK); PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, intent, PendingIntent.FLAG_IMMUTABLE); NotificationCompat.Builder builder = new 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 To make 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()) } NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this); // notificationId is a unique int for each notification that you must define notificationManager.notify(notificationId, builder.build()); Remember to save the notification ID that you pass to NotificationManagerCompat.notify() because you'll need it later if you want to update or remove the notification. Note: 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. 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) Intent snoozeIntent = new Intent(this, MyBroadcastReceiver.class); snoozeIntent.setAction(ACTION_SNOOZE); snoozeIntent.putExtra(EXTRA_NOTIFICATION_ID, 0); PendingIntent snoozePendingIntent = PendingIntent.getBroadcast(this, 0, snoozeIntent, 0); NotificationCompat.Builder builder = new 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); For more information about building a BroadcastReceiver to run background work, see the Broadcasts guide. If you're instead trying to build a notification with media playback buttons (such as to pause and skip tracks), see how to create a notification with media controls. Note: In Android 10 (API level 29) and higher, the platform automatically generates notification action buttons if an app does not provide its own. If you don't want your app's notifications to display any suggested replies or actions, you can opt-out of system-generated replies and actions by using setAllowGeneratedReplies() and setAllowSystemGeneratedContextualActions(). 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: 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() } // Key for the string that's delivered in the action's intent. private static final String KEY_TEXT_REPLY = "key_text_reply"; String replyLabel = getResources().getString(R.string.reply_label); RemoteInput remoteInput = new RemoteInput.Builder(KEY_TEXT_REPLY) .setLabel(replyLabel) .build(); 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) // Build a PendingIntent for the reply action to trigger. PendingIntent replyPendingIntent = PendingIntent.getBroadcast(getApplicationContext(), conversation.getConversationId(), getMessageReplyIntent(conversation.getConversationId()), PendingIntent.FLAG_UPDATE_CURRENT); Caution: If you re-use a PendingIntent, a user may reply to a different conversation than the one they thought they did. You must either provide a request code that is different for each conversation or provide an intent that doesn't return true when you call equals() on the reply intent of any other conversation. The conversation ID is frequently passed as part of the intent's extras bundle, but is ignored when you call equals(). 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(); 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) } // Build the notification and add the action. Notification newMessageNotification = new 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. NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this); notificationManager.notify(notificationId, newMessageNotification); The system prompts the user to input a response when they trigger the notification action, as shown in figure 3. 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) } private CharSequence getMessageText(Intent intent) { Bundle remoteInput = RemoteInput.getResultsFromIntent(intent); if (remoteInput != null) { return remoteInput.getCharSequence(KEY_TEXT_REPLY); } return null; } 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) } // Build a new notification, which informs the user that the system // handled their interaction with the previous notification. Notification repliedNotification = new Notification.Builder(context, CHANNEL_ID) .setSmallIcon(R.drawable.ic_message) .setContentText(getString(R.string.replied)) .build(); // Issue the new notification. NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this); 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()) } ... NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this); NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID); builder.setContentTitle("Picture Download") .setContentText("Download in progress") .setSmallIcon(R.drawable.ic_notification) .setPriority(NotificationCompat.PRIORITY_LOW); // Issue the initial notification with zero progress int PROGRESS_MAX = 100; int PROGRESS_CURRENT = 0; builder.setProgress(PROGRESS_MAX, PROGRESS_CURRENT, false); notificationManager.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); notificationManager.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). Note: Because the progress bar requires that your app continuously update the notification, this code should usually run in a background service. 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. Note: If you actually need to download a file, you should consider using DownloadManager, which provides its own notification to track your download progress. 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) NotificationCompat.Builder builder = new 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. Caution: Notifications containing full-screen intents are substantially intrusive, so it's important to use this type of notification only for the most urgent, time-sensitive messages.Note: If your app targets Android 10 (API level 29) or higher, you must request the USE_FULL_SCREEN_INTENT permission in your app's manifest file in order for the system to launch the full-screen activity associated with the time-sensitive 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) Intent fullScreenIntent = new Intent(this, ImportantActivity.class); PendingIntent fullScreenPendingIntent = PendingIntent.getActivity(this, 0, fullScreenIntent, PendingIntent.FLAG_UPDATE_CURRENT); NotificationCompat.Builder builder = new 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. Caution: Android applies a rate limit when updating a notification. If you post updates to a notification too frequently (many in less than one second), the system might drop some 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() Notification notification = new Notification.Builder(this, CHANNEL_ID) .setStyle(new 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. 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.