Start an Activity from a Notification

When you start an activity from a notification, you must preserve the user's expected navigation experience. Tapping *Back* should take the user back through the app's normal work flow to the Home screen, and opening the *Recents* screen should show the activity as a separate task. To preserve this navigation experience, you should start the activity in a fresh task.

Although the basic approach to set the tap behavior for your notification is described in Create a Notification (/training/notify-user/build-notification#builder), this page describes how you set up a `PendingIntent` (/reference/android/app/PendingIntent) for your notification's action so it creates a fresh task and back stack (/guide/components/activities/tasks-and-back-stack). But exactly how you do this depends on which type of activity you're starting:

Regular activity

> This is an activity that exists as a part of your app's normal UX flow. So when the user arrives in the activity from the notification, the new task should include a complete back stack (/guide/components/activities/tasks-and-back-stack), allowing them to press *Back* and navigate up the app hierarchy.

Special activity

> The user only sees this activity if it's started from a notification. In a sense, this activity extends the notification UI by providing information that would be hard to display in the notification itself. So this activity does not need a back stack.

Set up a regular activity PendingIntent

To start a "regular activity" from your notification, set up the `PendingIntent` (/reference/android/app/PendingIntent) using `TaskStackBuilder` (/reference/androidx/core/app/TaskStackBuilder) so that it creates a new back stack as follows.

Define your app's Activity hierarchy

Define the natural hierarchy for your activities by adding the `android:parentActivityName` (/guide/topics/manifest/activity-element#parent) attribute to each `<activity>` (/guide/topics/manifest/activity-element) element in your app manifest file. For example:

```
<activity
    android:name=".MainActivity"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<!-- MainActivity is the parent for ResultActivity -->
<activity
    android:name=".ResultActivity"
    android:parentActivityName=".MainActivity" />
    ...
</activity>
```

Build a PendingIntent with a back stack

To start an activity that includes a back stack of activities, you need to create an instance of `TaskStackBuilder` (/reference/androidx/core/app/TaskStackBuilder) and call `addNextIntentWithParentStack()` (/reference/androidx/core/app/TaskStackBuilder#addNextIntentWithParentStack(android.content.Intent)), passing it the `Intent` (/reference/android/content/Intent) for the activity you want to start.

As long as you've defined the parent activity for each activity as described above, you can call `getPendingIntent()` (/reference/androidx/core/app/TaskStackBuilder#getPendingIntent(int, int)) to receive a `PendingIntent`

(/reference/android/app/PendingIntent) that includes the entire back stack.

Kotlin (#kotlin) Java (#java)

```
// Create an Intent for the activity you want to start
Intent resultIntent = new Intent(this, ResultActivity.class);
// Create the TaskStackBuilder and add the intent, which inflates the back stack
TaskStackBuilder stackBuilder = TaskStackBuilder.create(this);
stackBuilder.addNextIntentWithParentStack(resultIntent);
// Get the PendingIntent containing the entire back stack
PendingIntent resultPendingIntent =
        stackBuilder.getPendingIntent(0,
            PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE);
```

If necessary, you can add arguments to `Intent` (/reference/android/content/Intent) objects in the stack by calling `TaskStackBuilder.editIntentAt()` (/reference/androidx/core/app/TaskStackBuilder#editIntentAt(int)). This is sometimes necessary to ensure that an activity in the back stack displays meaningful data when the user navigates up to it.

Then you can pass the `PendingIntent` (/reference/android/app/PendingIntent) to the notification as usual:

Kotlin (#kotlin) Java (#java)

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID);
builder.setContentIntent(resultPendingIntent);
...
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, builder.build());
```

Set up a special activity PendingIntent

Because a "special activity" started from a notification doesn't need a back stack, you can create the `PendingIntent` (/reference/android/app/PendingIntent) by calling `getActivity()` (/reference/android/app/PendingIntent#getActivity(android.content.Context, int, android.content.Intent, int)), but you should also be sure you've defined the appropriate task options in the manifest.

1. In your manifest, add the following attributes to the `<activity>` (/guide/topics/manifest/activity-element) element.

   `android:taskAffinity` (/guide/topics/manifest/activity-element#aff)`=""`

   > Combined with the `FLAG_ACTIVITY_NEW_TASK` (/reference/android/content/Intent#FLAG_ACTIVITY_NEW_TASK) flag that you'll use in code, setting this attribute blank ensures that this activity doesn't go into the app's default task. Any existing tasks that have the app's default affinity are not affected.

   `android:excludeFromRecents` (/guide/topics/manifest/activity-element#exclude)`="true"`

   > Excludes the new task from *Recents*, so that the user can't accidentally navigate back to it.

   For example:

```
<activity
    android:name=".ResultActivity"
    android:launchMode="singleTask"
    android:taskAffinity=""
```

```
    android:excludeFromRecents="true">
</activity>
```

2. Build and issue the notification:

    a. Create an `Intent` (/reference/android/content/Intent) that starts the `Activity` (/reference/android/app/Activity).

    b. Set the `Activity` (/reference/android/app/Activity) to start in a new, empty task by calling `setFlags()` (/reference/android/content/Intent#setFlags(int)) with the flags `FLAG_ACTIVITY_NEW_TASK` (/reference/android/content/Intent#FLAG_ACTIVITY_NEW_TASK) and `FLAG_ACTIVITY_CLEAR_TASK` (/reference/android/content/Intent#FLAG_ACTIVITY_CLEAR_TASK).

    c. Create a `PendingIntent` (/reference/android/app/PendingIntent) by calling `getActivity()` (/reference/android/app/PendingIntent#getActivity(android.content.Context, int, android.content.Intent, int)).

    For example:

Kotlin (#kotlin) Java (#java)

```
Intent notifyIntent = new Intent(this, ResultActivity.class);
// Set the Activity to start in a new, empty task
notifyIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
                    | Intent.FLAG_ACTIVITY_CLEAR_TASK);
// Create the PendingIntent
PendingIntent notifyPendingIntent = PendingIntent.getActivity(
        this, 0, notifyIntent,
        PendingIntent.FLAG_UPDATE_CURRENT | PendingIntent.FLAG_IMMUTABLE
);
```

3. Then you can pass the `PendingIntent` (/reference/android/app/PendingIntent) to the notification as usual:

Kotlin (#kotlin) Java (#java)

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this, CHANNEL_ID);
builder.setContentIntent(notifyPendingIntent);
...
NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify(NOTIFICATION_ID, builder.build());
```

For more information about the various task options and how the back stack works, see Tasks and the back stack (/guide/components/activities/tasks-and-back-stack).