Broadcasts overview

Archived: 2026-04-05 13:59:52 UTC

Android apps send and receive broadcast messages from the Android system and other Android apps, similar to the publish-subscribe design pattern. The system and apps typically send broadcasts when certain events occur. For example, the Android system sends broadcasts when various system events occur, such as system boot or device charging. Apps also send custom broadcasts, for example, to notify other apps of something that might interest them (for example, new data download).

Apps can register to receive specific broadcasts. When a broadcast is sent, the system automatically routes broadcasts to apps that have subscribed to receive that particular type of broadcast.

Generally speaking, broadcasts can be used as a messaging system across apps and outside of the normal user flow. However, you must be careful not to abuse the opportunity to respond to broadcasts and run jobs in the background that can contribute to a slow system performance.

About system broadcasts

The system automatically sends broadcasts when various system events occur, such as when the system switches in and out of Airplane Mode. All subscribed apps receive these broadcasts.

The `Intent` object wraps the broadcast message. The `action` string identifies the event that occurred, such as `android.intent.action.AIRPLANE_MODE`. The intent might also include additional information bundled into its extra field. For example, the Airplane Mode intent includes a boolean extra that indicates whether or not Airplane Mode is on.

For more information about how to read intents and get the action string from an intent, see Intents and Intent Filters.

System broadcast actions

For a complete list of system broadcast actions, see the `BROADCAST_ACTIONS.TXT` file in the Android SDK. Each broadcast action has a constant field associated with it. For example, the value of the constant `ACTION_AIRPLANE_MODE_CHANGED` is `android.intent.action.AIRPLANE_MODE`. Documentation for each broadcast action is available in its associated constant field.

Changes to system broadcasts

As the Android platform evolves, it periodically changes how system broadcasts behave. Keep the following changes in mind to support all versions of Android.

Android 16

In Android 16, broadcast delivery order using the `android:priority` attribute or `IntentFilter.setPriority()` across different processes won't be guaranteed. Broadcast priorities are only respected within the same application process rather than across all processes.

Also, broadcast priorities are automatically confined to the range (`SYSTEM_LOW_PRIORITY` + 1, `SYSTEM_HIGH_PRIORITY` - 1). Only system components are allowed to set `SYSTEM_LOW_PRIORITY` , `SYSTEM_HIGH_PRIORITY` as broadcast priority.

Android 14

While apps are in a cached state, the system optimizes broadcast delivery for system health. For example, the system defers less important system broadcasts such as `ACTION_SCREEN_ON` while the app is in a cached state. Once the app goes from the cached state into an active process lifecycle, the system delivers any deferred broadcasts.

Important broadcasts that are declared in the manifest temporarily remove apps from the cached state for delivery.

Android 9

Beginning with Android 9 (API level 28), The `NETWORK_STATE_CHANGED_ACTION` broadcast doesn't receive information about the user's location or personally identifiable data.

If your app is installed on a device running Android 9.0 (API level 28) or higher, the system doesn't include SSIDs, BSSIDs, connection information, or scan results in Wi-Fi broadcasts. To get this information, call `getConnectionInfo()` instead.

Android 8.0

Beginning with Android 8.0 (API level 26), the system imposes additional restrictions on manifest-declared receivers.

If your app targets Android 8.0 or higher, you cannot use the manifest to declare a receiver for most implicit broadcasts (broadcasts that don't target your app specifically). You can still use a context-registered receiver when the user is actively using your app.

Android 7.0

Android 7.0 (API level 24) and higher don't send the following system broadcasts:

- `ACTION_NEW_PICTURE`
- `ACTION_NEW_VIDEO`

Also, apps targeting Android 7.0 and higher must register the `CONNECTIVITY_ACTION` broadcast using `registerReceiver(BroadcastReceiver, IntentFilter)` . Declaring a receiver in the manifest doesn't work.

Receive broadcasts

Apps can receive broadcasts in two ways: through context-registered receivers and manifest-declared receivers.

Context-registered receivers

Context-registered receivers receive broadcasts as long as their registering context is valid. This is typically between the calls to `registerReceiver` and `unregisterReceiver`. The registering context also becomes invalid when the system destroys the corresponding context. For example, if you register within an `Activity` context, you receive broadcasts as long as the activity remains active. If you register with the Application context, you receive broadcasts as long as the app runs.

To register a receiver with a context, perform the following steps:

1. In your app's module-level build file, include version 1.9.0 or higher of the AndroidX Core library:

```
dependencies {
    def core_version = "1.18.0"

    // Java language implementation
    implementation "androidx.core:core:$core_version"
    // Kotlin
    implementation "androidx.core:core-ktx:$core_version"

    // To use RoleManagerCompat
    implementation "androidx.core:core-role:1.1.0"

    // To use the Animator APIs
    implementation "androidx.core:core-animation:1.0.0"
    // To test the Animator APIs
    androidTestImplementation "androidx.core:core-animation-testing:1.0.0"

    // Optional - To enable APIs that query the performance characteristics of GMS devices.
    implementation "androidx.core:core-performance:1.0.0"

    // Optional - to use ShortcutManagerCompat to donate shortcuts to be used by Google
    implementation "androidx.core:core-google-shortcuts:1.1.0"

    // Optional - to support backwards compatibility of RemoteViews
    implementation "androidx.core:core-remoteviews:1.1.0"

    // Optional - APIs for SplashScreen, including compatibility helpers on devices prior Andr
    implementation "androidx.core:core-splashscreen:1.2.0"
}
```

```
dependencies {
    val core_version = "1.18.0"
```

```
    // Java language implementation
    implementation("androidx.core:core:$core_version")
    // Kotlin
    implementation("androidx.core:core-ktx:$core_version")

    // To use RoleManagerCompat
    implementation("androidx.core:core-role:1.1.0")

    // To use the Animator APIs
    implementation("androidx.core:core-animation:1.0.0")
    // To test the Animator APIs
    androidTestImplementation("androidx.core:core-animation-testing:1.0.0")

    // Optional - To enable APIs that query the performance characteristics of GMS devices.
    implementation("androidx.core:core-performance:1.0.0")

    // Optional - to use ShortcutManagerCompat to donate shortcuts to be used by Google
    implementation("androidx.core:core-google-shortcuts:1.1.0")

    // Optional - to support backwards compatibility of RemoteViews
    implementation("androidx.core:core-remoteviews:1.1.0")

    // Optional - APIs for SplashScreen, including compatibility helpers on devices prior Andr
    implementation("androidx.core:core-splashscreen:1.2.0")
}
```

2. Create an instance of `BroadcastReceiver`:

```
val myBroadcastReceiver = MyBroadcastReceiver()
```

```
MyBroadcastReceiver myBroadcastReceiver = new MyBroadcastReceiver();
```

3. Create an instance of `IntentFilter`:

```
val filter = IntentFilter("com.example.snippets.ACTION_UPDATE_DATA")
```

```
IntentFilter filter = new IntentFilter("com.example.snippets.ACTION_UPDATE_DATA");
```

4. Choose whether the broadcast receiver should be exported and visible to other apps on the device. If this receiver is listening for broadcasts sent from the system or from other apps—even other apps that you own—use the `RECEIVER_EXPORTED` flag. If instead this receiver is listening only for broadcasts sent by your app, use the `RECEIVER_NOT_EXPORTED` flag.

```
 val listenToBroadcastsFromOtherApps = false
val receiverFlags = if (listenToBroadcastsFromOtherApps) {
    ContextCompat.RECEIVER_EXPORTED
} else {
    ContextCompat.RECEIVER_NOT_EXPORTED
}
```

BroadcastReceiverSnippets.kt

```
 boolean listenToBroadcastsFromOtherApps = false;
int receiverFlags = listenToBroadcastsFromOtherApps
        ? ContextCompat.RECEIVER_EXPORTED
        : ContextCompat.RECEIVER_NOT_EXPORTED;
```

BroadcastReceiverJavaSnippets.java

5. Register the receiver by calling `registerReceiver()`:

```
ContextCompat.registerReceiver(context, myBroadcastReceiver, filter, receiverFlags)
```

```
ContextCompat.registerReceiver(context, myBroadcastReceiver, filter, receiverFlags);
```

6. To stop receiving broadcasts, call `unregisterReceiver(android.content.BroadcastReceiver)`. Be sure to unregister the receiver when you no longer need it or the context is no longer valid.

Unregister your broadcast receiver

While the broadcast receiver is registered, it holds a reference to the Context that you registered it with. This can potentially cause leaks if the receiver's registered scope exceeds the Context lifecycle scope. For example, this can occur when you register a receiver on an Activity scope, but you forget to unregister it when the system destroys the Activity. Therefore, always unregister your broadcast receiver.

```
class MyActivity : ComponentActivity() {
    private val myBroadcastReceiver = MyBroadcastReceiver()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // ...
        ContextCompat.registerReceiver(this, myBroadcastReceiver, filter, receiverFlags)
        setContent { MyApp() }
    }

    override fun onDestroy() {
        super.onDestroy()
        // When you forget to unregister your receiver here, you're causing a leak!
        this.unregisterReceiver(myBroadcastReceiver)
    }
}
```

BroadcastReceiverSnippets.kt

```
class MyActivity extends ComponentActivity {
    MyBroadcastReceiver myBroadcastReceiver;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // ...
        ContextCompat.registerReceiver(this, myBroadcastReceiver, filter, receiverFlags);
        // Set content
    }
}
```

BroadcastReceiverJavaSnippets.java

Register receivers in the smallest scope

Your broadcast receiver should only be registered when you're actually interested in the result. Choose the smallest possible receiver scope:

- `LifecycleResumeEffect` or activity `onResume` / `onPause` lifecycle methods: The broadcast receiver only receives updates while the app is in its resumed state.

- `LifecycleStartEffect` or activity `onStart` / `onStop` lifecycle methods: The broadcast receiver only receives updates while the app is in its resumed state.
- `DisposableEffect`: The broadcast receiver only receives updates while the composable is in the composition tree. This scope is not attached to the activity lifecycle scope. Consider registering the receiver on the application context. This is because the composable could theoretically outlive the activity lifecycle scope and leak the activity.
- Activity `onCreate` / `onDestroy`: The broadcast receiver receives updates while the activity is in its created state. Make sure to unregister in `onDestroy()` and not `onSaveInstanceState(Bundle)` because this might not be called.
- A custom scope: For example, you can register a receiver in your `ViewModel` scope, so it survives activity recreation. Make sure to use the application context to register the receiver on, as the receiver can outlive the activity lifecycle scope and leak the activity.

Create stateful and stateless composable

Compose has stateful and stateless composables. Registering or unregistering a broadcast receiver inside a composable makes it stateful. The composable is not a deterministic function that renders the same content when passed the same parameters. Internal state can change based on calls to the registered broadcast receiver.

As a best practice in Compose, we recommend that you split your composables into stateful and stateless versions. Therefore, we recommend that you hoist the creation of the broadcast receiver out of a Composable to make it stateless:

```
 @Composable
fun MyStatefulScreen() {
    val myBroadcastReceiver = remember { MyBroadcastReceiver() }
    val context = LocalContext.current
    LifecycleStartEffect(true) {
        // ...
        ContextCompat.registerReceiver(context, myBroadcastReceiver, filter, flags)
        onStopOrDispose { context.unregisterReceiver(myBroadcastReceiver) }
    }
    MyStatelessScreen()
}

@Composable
fun MyStatelessScreen() {
    // Implement your screen
}
```

BroadcastReceiverSnippets.kt

Manifest-declared receivers

If you declare a broadcast receiver in your manifest, the system launches your app when the broadcast is sent. If the app is not already running, the system launches the app.

To declare a broadcast receiver in the manifest, perform the following steps:

1. Specify the `<receiver>` element in your app's manifest.

```
 <!-- If this receiver listens for broadcasts sent from the system or from
     other apps, even other apps that you own, set android:exported to "true". -->
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
    <intent-filter>
        <action android:name="com.example.snippets.ACTION_UPDATE_DATA" />
    </intent-filter>
</receiver>
```

AndroidManifest.xml

   The intent filters specify the broadcast actions your receiver subscribes to.

2. Subclass `BroadcastReceiver` and implement `onReceive(Context, Intent)`. The broadcast receiver in the following example logs and displays the contents of the broadcast:

```
 class MyBroadcastReceiver : BroadcastReceiver() {

    @Inject
    lateinit var dataRepository: DataRepository

    override fun onReceive(context: Context, intent: Intent) {
        if (intent.action == "com.example.snippets.ACTION_UPDATE_DATA") {
            val data = intent.getStringExtra("com.example.snippets.DATA") ?: "No data"
            // Do something with the data, for example send it to a data repository:
            dataRepository.updateData(data)
        }
    }
}
```

BroadcastReceiverSnippets.kt

```
public static class MyBroadcastReceiver extends BroadcastReceiver {

    @Inject
    DataRepository dataRepository;

    @Override
    public void onReceive(Context context, Intent intent) {
        if (Objects.equals(intent.getAction(), "com.example.snippets.ACTION_UPDATE_DATA")) {
            String data = intent.getStringExtra("com.example.snippets.DATA");
            // Do something with the data, for example send it to a data repository:
            if (data != null) { dataRepository.updateData(data); }
        }
    }
}
```

BroadcastReceiverJavaSnippets.java

The system package manager registers the receiver when the app is installed. The receiver then becomes a separate entry point into your app which means that the system can start the app and deliver the broadcast if the app is not running.

The system creates a new `BroadcastReceiver` component object to handle each broadcast that it receives. This object is valid only for the duration of the call to `onReceive(Context, Intent)`. Once your code returns from this method, the system considers the component no longer active.

Effects on process state

Whether your `BroadcastReceiver` is operating or not affects its contained process, which can alter its system-killing likelihood. A foreground process executes a receiver's `onReceive()` method. The system runs the process except under extreme memory pressure.

The system deactivates the `BroadcastReceiver` after `onReceive()`. The receiver's host process's significance depends on its app components. If that process hosts only a manifest-declared receiver, the system might kill it after `onReceive()` to free resources for other more critical processes. This is common for apps the user has never or not recently interacted with.

Thus, broadcast receivers shouldn't initiate long-running background threads. The system can stop the process at any moment after `onReceive()` to reclaim memory, terminating the created thread. To keep the process alive, schedule a `JobService` from the receiver using the `JobScheduler` so the system knows the process is still working. Background Work Overview provides more details.

Send broadcasts

Android provides two ways for apps to send broadcasts:

- The `sendOrderedBroadcast(Intent, String)` method sends broadcasts to one receiver at a time. As each receiver executes in turn, it can propagate a result to the next receiver. It can also completely abort the broadcast so that it doesn't reach other receivers. You can control the order in which receivers run within the same app process. To do so, use the `android:priority` attribute of the matching intent-filter. Receivers with the same priority are run in an arbitrary order.
- The `sendBroadcast(Intent)` method sends broadcasts to all receivers in an undefined order. This is called a Normal Broadcast. This is more efficient, but means that receivers cannot read results from other receivers, propagate data received from the broadcast, or abort the broadcast.

The following code snippet demonstrates how to send a broadcast by creating an Intent and calling `sendBroadcast(Intent)`.

```
 val intent = Intent("com.example.snippets.ACTION_UPDATE_DATA").apply {
    putExtra("com.example.snippets.DATA", newData)
    setPackage("com.example.snippets")
}
context.sendBroadcast(intent)
```

BroadcastReceiverSnippets.kt

```
 Intent intent = new Intent("com.example.snippets.ACTION_UPDATE_DATA");
intent.putExtra("com.example.snippets.DATA", newData);
intent.setPackage("com.example.snippets");
context.sendBroadcast(intent);
```

BroadcastReceiverJavaSnippets.java

The broadcast message is wrapped in an `Intent` object. The intent's `action` string must provide the app's Java package name syntax and uniquely identify the broadcast event. You can attach additional information to the intent with `putExtra(String, Bundle)`. You can also limit a broadcast to a set of apps in the same organization by calling `setPackage(String)` on the intent.

Restrict broadcasts with permissions

Permissions allow you to restrict broadcasts to the set of apps that hold certain permissions. You can enforce restrictions on either the sender or receiver of a broadcast.

Send broadcasts with permissions

When you call `sendBroadcast(Intent, String)` or `sendOrderedBroadcast(Intent, String, BroadcastReceiver, Handler, int, String, Bundle)`, you can specify a permission parameter. Only receivers who have requested that permission with the `<uses-permission>` tag in their manifest can receive the broadcast. If the permission is dangerous, you must grant the permission before the receiver can receive the broadcast. For example, the following code sends a broadcast with a permission:

```
context.sendBroadcast(intent, android.Manifest.permission.ACCESS_COARSE_LOCATION)
```

```
context.sendBroadcast(intent, android.Manifest.permission.ACCESS_COARSE_LOCATION);
```

To receive the broadcast, the receiving app must request the permission as follows:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

You can specify either an existing system permission like `BLUETOOTH_CONNECT` or define a custom permission with the `<permission>` element. For information on permissions and security in general, see the System Permissions.

Receive broadcasts with permissions

If you specify a permission parameter when registering a broadcast receiver (either with `registerReceiver(BroadcastReceiver, IntentFilter, String, Handler)` or in `<receiver>` tag in your manifest), then only broadcasters who have requested the permission with the `<uses-permission>` tag in their manifest can send an Intent to the receiver. If the permission is dangerous, the broadcaster must also be granted the permission.

For example, assume your receiving app has a manifest-declared receiver as follows:

```
 <!-- If this receiver listens for broadcasts sent from the system or from
     other apps, even other apps that you own, set android:exported to "true". -->
<receiver
    android:name=".MyBroadcastReceiverWithPermission"
    android:permission="android.permission.ACCESS_COARSE_LOCATION"
    android:exported="true">
    <intent-filter>
        <action android:name="com.example.snippets.ACTION_UPDATE_DATA" />
    </intent-filter>
</receiver>
```

AndroidManifest.xml

```
```

Or your receiving app has a context-registered receiver as follows:

```
 ContextCompat.registerReceiver(
    context, myBroadcastReceiver, filter,
    android.Manifest.permission.ACCESS_COARSE_LOCATION,
    null, // scheduler that defines thread, null means run on main thread
    receiverFlags
)
```

BroadcastReceiverSnippets.kt

```
 ContextCompat.registerReceiver(
        context, myBroadcastReceiver, filter,
        android.Manifest.permission.ACCESS_COARSE_LOCATION,
        null, // scheduler that defines thread, null means run on main thread
        receiverFlags
);
```

BroadcastReceiverJavaSnippets.java

Then, to be able to send broadcasts to those receivers, the sending app must request the permission as follows:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Security considerations

Here are some security considerations for sending and receiving broadcasts:

- If many apps have registered to receive the same broadcast in their manifest, it can cause the system to launch a lot of apps, causing a substantial impact on both device performance and user experience. To avoid this, prefer using context registration over manifest declaration. Sometimes, the Android system itself enforces the use of context-registered receivers. For example, the `CONNECTIVITY_ACTION` broadcast is delivered only to context-registered receivers.
- Don't broadcast sensitive information using an implicit intent. Any app can read the information if it registers to receive the broadcast. There are three ways to control who can receive your broadcasts:

    - You can specify a permission when sending a broadcast.
    - In Android 4.0 (API level 14) and higher, you can specify a package with `setPackage(String)` when sending a broadcast. The system restricts the broadcast to the set of apps that match the package.

- When you register a receiver, any app can send potentially malicious broadcasts to your app's receiver. There are several ways to limit the broadcasts that your app receives:

    - You can specify a permission when registering a broadcast receiver.
    - For manifest-declared receivers, you can set the android:exported attribute to "false" in the manifest. The receiver does not receive broadcasts from sources outside of the app.

- The namespace for broadcast actions is global. Make sure that action names and other strings are written in a namespace you own. Otherwise, you may inadvertently conflict with other apps.

- Because a receiver's `onReceive(Context, Intent)` method runs on the main thread, it should execute and return quickly. If you need to perform long-running work, be careful about spawning threads or starting background services because the system can kill the entire process after `onReceive()` returns. For more information, see Effect on process state To perform long running work, we recommend:

    - Calling `goAsync()` in your receiver's `onReceive()` method and passing the `BroadcastReceiver.PendingResult` to a background thread. This keeps the broadcast active after returning from `onReceive()`. However, even with this approach the system expects you to finish with the broadcast very quickly (under 10 seconds). It does allow you to move work to another thread to avoid glitching the main thread.
    - Scheduling a job with the `JobScheduler`. For more information, see Intelligent Job Scheduling.

- Don't start activities from broadcast receivers because the user experience is jarring; especially if there is more than one receiver. Instead, consider displaying a notification.

Source: https://developer.android.com/guide/components/broadcasts#changes-system-broadcasts