App Widget Host

In this document

The Android Home screen available on most Android devices allows the user to embedapp widgets for quick access to content. If you're building a Home replacement or a similar app, you can also allow the user to embed app widgets by implementing an `AppWidgetHost`. This is not something that most apps will ever need to do, but if you are creating your own host, it's important to understand the contractual obligations a host implicitly agrees to.

This document focuses on the responsibilities involved in implementing a custom `AppWidgetHost`. For an example of how to implement an `AppWidgetHost`, see the source code for the Android Home screen Launcher.

Here is an overview of key classes and concepts involved in implementing a custom `AppWidgetHost`:

- **App Widget Host**— The `AppWidgetHost` provides the interaction with the AppWidget service for apps, like the home screen, that want to embed app widgets in their UI. An `AppWidgetHost` must have an ID that is unique within the host's own package. This ID remains persistent across all uses of the host. The ID is typically a hard-coded value that you assign in your application.
- **App Widget ID**— Each app widget instance is assigned a unique ID at the time of binding (see `bindAppWidgetIdIfAllowed()`, discussed in more detail in Binding app widgets). The unique ID is obtained by the host using `allocateAppWidgetId()`. This ID is persistent across the lifetime of the widget, that is, until it is deleted from the host. Any host-specific state (such as the size and location of the widget) should be persisted by the hosting package and associated with the app widget ID.
- **App Widget Host View**— `AppWidgetHostView` can be thought of as a frame that the widget is wrapped in whenever it needs to be displayed. An app widget is assigned to an `AppWidgetHostView` every time the widget is inflated by the host.
- **Options Bundle**— The `AppWidgetHost` uses the options bundle to communicate information to the `AppWidgetProvider` about how the widget is being displayed (for example, size range, and whether the widget is on a lockscreen or the home screen). This information allows the `AppWidgetProvider` to tailor the widget's contents and appearance based on how and where it is displayed. You use `updateAppWidgetOptions()` and `updateAppWidgetSize()` to modify an app widget's bundle. Both of these methods trigger a callback to the `AppWidgetProvider`.

Binding App Widgets

When a user adds an app widget to a host, a process called *binding* occurs. *Binding* refers to associating a particular app widget ID to a specific host and to a specific `AppWidgetProvider`. There are different ways of achieving this, depending on what version of Android your app is running on.

Binding app widgets on Android 4.0 and lower

On devices running Android version 4.0 and lower, users add app widgets via a system activity that allows users to select a widget. This implicitly does a permission check—that is, by adding the app widget, the user is implicitly granting permission to your app to add app widgets to the host. Here is an example that illustrates this approach, taken from the original Launcher. In this snippet, an event handler invokes `startActivityForResult()` with the request code `REQUEST_PICK_APPWIDGET` in response to a user action:

```
private static final int REQUEST_CREATE_APPWIDGET = 5;
private static final int REQUEST_PICK_APPWIDGET = 9;
...
public void onClick(DialogInterface dialog, int which) {
    switch (which) {
    ...
        case AddAdapter.ITEM_APPWIDGET: {
            ...
            int appWidgetId =
                    Launcher.this.mAppWidgetHost.allocateAppWidgetId();
            Intent pickIntent =
                    new Intent(AppWidgetManager.ACTION_APPWIDGET_PICK);
            pickIntent.putExtra
                    (AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
            ...
            startActivityForResult(pickIntent, REQUEST_PICK_APPWIDGET);
            break;
    }
    ...
}
```

When the system activity finishes, it returns a result with the user's chosen app widget to your activity. In the following example, the activity responds by calling `addAppWidget()` to add the app widget:

```
public final class Launcher extends Activity
        implements View.OnClickListener, OnLongClickListener {
    ...
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        mWaitingForResult = false;

        if (resultCode == RESULT_OK && mAddItemCellInfo != null) {
            switch (requestCode) {
                ...
                case REQUEST_PICK_APPWIDGET:
                    addAppWidget(data);
                    break;
                case REQUEST_CREATE_APPWIDGET:
                    completeAddAppWidget(data, mAddItemCellInfo, !mDesktopLocked);
                    break;
                }
        }
        ...
    }
}
```

The method `addAppWidget()` checks to see if the app widget needs to be configured before it's added:

```
void addAppWidget(Intent data) {
    int appWidgetId = data.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, -1);

    String customWidget = data.getStringExtra(EXTRA_CUSTOM_WIDGET);
    AppWidgetProviderInfo appWidget =
            mAppWidgetManager.getAppWidgetInfo(appWidgetId);

    if (appWidget.configure != null) {
        // Launch over to configure widget, if needed.
        Intent intent = new Intent(AppWidgetManager.ACTION_APPWIDGET_CONFIGURE);
        intent.setComponent(appWidget.configure);
        intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
        startActivityForResult(intent, REQUEST_CREATE_APPWIDGET);
    } else {
        // Otherwise, finish adding the widget.
    }
}
```

For more discussion of configuration, see Creating an App Widget Configuration Activity.

Once the app widget is ready, the next step is to do the actual work of adding it to the workspace. The original Launcher uses a method called `completeAddAppWidget()` to do this.

Binding app widgets on Android 4.1 and higher

Android 4.1 adds APIs for a more streamlined binding process. These APIs also make it possible for a host to provide a custom UI for binding. To use this improved process, your app must declare the `BIND_APPWIDGET` permission in its manifest:

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

But this is just the first step. At runtime the user must explicitly grant permission to your app to allow it to add app widgets to the host. To test whether your app has permission to add the widget, you use the `bindAppWidgetIdIfAllowed()` method. If `bindAppWidgetIdIfAllowed()` returns `false`, your app must display a dialog prompting the user to grant permission ("allow" or "always allow," to cover all future app widget additions). This snippet gives an example of how to display the dialog:

```
Intent intent = new Intent(AppWidgetManager.ACTION_APPWIDGET_BIND);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID, appWidgetId);
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_PROVIDER, info.componentName);
// This is the options bundle discussed above
intent.putExtra(AppWidgetManager.EXTRA_APPWIDGET_OPTIONS, options);
startActivityForResult(intent, REQUEST_BIND_APPWIDGET);
```

The host also has to check whether the user added an app widget that needs configuration. For more discussion of this topic, see Creating an App Widget Configuration Activity.

Host Responsibilities

Widget developers can specify a number of configuration settings for widgets using the AppWidgetProviderInfo metadata. These configuration options, discussed in more detail below, can be retrieved by the host from the `AppWidgetProviderInfo` object associated with a widget provider.

Regardless of the version of Android you are targeting, all hosts have the following responsibilities:

- When adding a widget, you must allocate the widget ID as described above. You must also make sure that when a widget is removed from the host, you call `deleteAppWidgetId()` to deallocate the widget ID.
- When adding a widget, be sure to launch its configuration activity if it exists, as described in Updating the App Widget from the Configuration Activity. This is a necessary step for

> **What Version are You Targeting?**
>
> The approach you use in implementing your host should depend on what Android version you're targeting. Many of the features described in this section were introduced in 3.0 or later. For example:
>
> - Android 3.0 (API Level 11) introduces auto-advance behavior for widgets.
> - Android 3.1 (API Level 12) introduces the ability to resize widgets.
> - Android 4.0 (API Level 15) introduces a change in padding

many app widgets before they can be properly displayed.

- Every app widget specifies a minimum width and height in dps, as defined in the `AppWidgetProviderInfo` metadata (using `android:minWidth` and `android:minHeight`). Make sure that the widget is laid out with at least this many dps. For example, many hosts align icons and widgets in a grid. In this scenario, by default the host should add the app widget using the minimum number of cells that satisfy the `minWidth` and `minHeight` constraints.

In addition to the requirements listed above, specific platform versions introduce features that place new responsibilities on the host. These are described in the following sections.

> policy that puts the responsibility on the host to manage padding.
>
> - Android 4.1 (API Level 16) adds an API that allows the widget provider to get more detailed information about the environment in which its widget instances are being hosted.
> - Android 4.2 (API Level 17) introduces the options bundle and the `bindAppWidgetIdIfAllowed()` method. It also introduces lockscreen widgets.
>
> If you are targeting earlier devices, refer to the original Launcher as an example.

Android 3.0

Android 3.0 (API Level 11) introduces the ability for a widget to specify `autoAdvanceViewId()`. This view ID should point to an instance of an `Advanceable`, such as `StackView` or `AdapterViewFlipper`. This indicates that the host should call `advance()` on this view at an interval deemed appropriate by the host (taking into account whether it makes sense to advance the widget—for example, the host probably wouldn't want to advance a widget if it were on another page, or if the screen were turned off).

Android 3.1

Android 3.1 (API Level 12) introduces the ability to resize widgets. A widget can specify that it is resizable using the `android:resizeMode` attribute in the `AppWidgetProviderInfo` metadata, and indicate whether it supports horizontal and/or vertical resizing. Introduced in Android 4.0 (API Level 14), the widget can also specify a `android:minResizeWidth` and/or `android:minResizeHeight`.

It is the host's responsibility to make it possible for the widget to be resized horizontally and/or vertically, as specified by the widget. A widget that specifies that it is resizable can be resized arbitrarily large, but should not be resized smaller than the values specified by `android:minResizeWidth` and `android:minResizeHeight`. For a sample implementation, see `AppWidgetResizeFrame` in `Launcher2`.

Android 4.0

Android 4.0 (API Level 15) introduces a change in padding policy that puts the responsibility on the host to manage padding. As of 4.0, app widgets no longer include their own padding. Instead, the system adds padding for each widget, based the characteristics of the current screen. This leads to a more uniform, consistent presentation of widgets in a grid. To assist applications that host app widgets, the platform provides the method `getDefaultPaddingForWidget()`. Applications can call this method to get the system-defined padding and account for it when computing the number of cells to allocate to the widget.

Android 4.1

Android 4.1 (API Level 16) adds an API that allows the widget provider to get more detailed information about the environment in which its widget instances are being hosted. Specifically, the host hints to the widget provider about the size at which the widget is being displayed. It is the host's responsibility to provide this size information.

The host provides this information via `updateAppWidgetSize()`. The size is specified as a minimum and maximum width/height in dps. The reason that a range is specified (as opposed to a fixed size) is because the width and height of a widget may change with orientation. You don't want the host to have to update all of its widgets on rotation, as this could cause serious system slowdown. These values should be updated once upon the widget being placed, any time the widget is resized, and any time the launcher inflates the widget for the first time in a given boot (as the values aren't persisted across boot).

Android 4.2

Android 4.2 (API Level 17) adds the ability for the options bundle to be specified at bind time. This is the ideal way to specify app widget options, including size, as it gives the `AppWidgetProvider` immediate access to the options data on the first update. This can be achieved by using the method `bindAppWidgetIdIfAllowed()`. For more discussion of this topic, see Binding app widgets.

Android 4.2 also introduces lockscreen widgets. When hosting widgets on the lockscreen, the host must specify this information within the app widget options bundle (the `AppWidgetProvider` can use this information to style the widget appropriately). To designate a widget as a

lockscreen widget, use `updateAppWidgetOptions()` and include the field `OPTION_APPWIDGET_HOST_CATEGORY` with the value `WIDGET_CATEGORY_KEYGUARD`. This option defaults to `WIDGET_CATEGORY_HOME_SCREEN`, so it is not explicitly required to set this for a home screen host.

Make sure that your host adds only app widgets that are appropriate for your app—for example, if your host is a home screen, ensure that the `android:widgetCategory` attribute in the `AppWidgetProviderInfo` metadata includes the flag `WIDGET_CATEGORY_HOME_SCREEN`. Similarly, for the lockscreen, ensure that field includes the flag `WIDGET_CATEGORY_KEYGUARD`. For more discussion of this topic, see Enabling App Widgets on the Lockscreen.