Language and Locale

Starting in Android 7.0 (API level 24), Android provides enhanced support for multilingual users, allowing them to select multiple locales in settings. Android provides this capability by greatly expanding the number of locales supported and changing the way the system resolves resources.

This document starts by explaining the resource resolution strategy in versions of Android lower than 7.0 (API level 24). Next, it describes the improved resource-resolution strategy in Android 7.0. Last, it explains how to take advantage of the expanded number of locales to support more multilingual users.

Challenges in Resolving Language Resources

Prior to Android 7.0, Android could not always successfully match app and system locales.

For example, assume that you have the following situation:

- Your app's default language is `en_US` (US English), and it also has Spanish strings localized in `es_ES` resource files.
- A device is set to `es_MX`

  When your Java code refers to strings, the system would load strings from the default (`en_US`) resource file, even if the app has Spanish resources localized under `es_ES`. This is because when the system cannot find an exact match, it continues to look for resources by stripping the country code off the locale. Finally, if no match is found, the system falls back to the default, which is `en_US`.

  The system would also default to `en_US` if the user chose a language that the app didn't support at all, like French. For example:

  Table 1. Resource resolution without an exact locale match.

User Settings	App Resources	Resource Resolution
fr_CH	default (en) de_DE es_ES fr_FR it_IT	Try fr_CH => Fail Try fr => Fail Use default (en)

  In this example, the system displays English strings without knowing whether the user can understand English. This behavior is pretty common today.

Improvements to Resource-Resolution Strategy

Android 7.0 (API level 24) brings more robust resource resolution, and finds better fallbacks automatically. However, to speed up resolution and improve maintainability, you should store resources in the most common parent dialect. For example, if you were storing Spanish resources in the `values-es-rUS` directory before, move them into the `values-b+es+419` directory, which contains Latin American Spanish. Similarly, if you have resource strings in a directory named `values-en-rGB`, rename the directory to `values-b+en+001` (International English), because the most common parent for `en-GB` strings is `en-001`. The following examples explain why these practices improve performance and reliability of resource resolution.

Resource resolution examples

With versions of Android greater than 7.0, the case described in **Table 1** is resolved differently:

Table 2. An improved resolution strategy for when there is no exact locale match.

User Settings	App Resources	Resource Resolution
1. fr_CH	default (en) de_DE es_ES fr_FR it_IT	Try fr_CH => Fail Try fr => Fail Try children of fr => fr_FR Use fr_FR

Now the user gets French resources instead of English. This example also shows why you should store French strings in `fr` rather than `fr_FR` for Android 7.0 or higher. Here the course of action is to match the closest parent dialect, making resolution faster and more predictable.

In addition to this improved resolution logic, Android now offers more user languages to choose from. Let's try the above example again with Italian specified as an additional user language, but without app support for French.

Table 3. Resource resolution when the app only matches the user's second-preferred locale setting.

User Settings	App Resources	Resource Resolution
1. fr_CH 2. it_CH	default (en) de_DE es_ES it_IT	Try fr_CH => Fail Try fr => Fail Try children of fr => Fail Try it_CH => Fail Try it => Fail Try children of it => it_IT Use it_IT

The user still gets a language they understand, even though the app doesn’t support French.

Designing your App to Support Additional Locales

LocaleList API

Starting with Android 7.0 (API level 24), Android exposes the `LocaleList.getDefault()` API that lets apps directly query the list of languages a user has specified. This API allows you to create more sophisticated app behavior and better-optimized display of content. For example, Search can show results in multiple languages based on user’s settings. Browser apps can avoid offering to translate pages in a language the user already knows, and keyboard apps can auto-enable all appropriate layouts.

Formatters

Up through Android 6.0 (API level 23), Android supported only one or two locales for many common languages (en, es, ar, fr, ru). Because there were only a few variants of each language, apps could get away with storing some numbers and dates as hard coded strings in resource files. However, with Android's broadened set of supported locales, there can be significant differences in formats for dates, times, currencies, and similar information even within a single locale. Hard-coding your formats can produce a confusing experience for end users. Therefore, when developing for Android 7.0 or higher versions, make sure to use formatters instead of hard coding numbers and date strings.

For example, Android 7.0 and higher includes support for 27 Arabic locales. These locales can share most resources, but some prefer ASCII digits, while others prefer native digits. For example, when you want to create a sentence with a digit variable, such as "Choose a 4 digit pin", use formatters as shown below:

```
format(locale, "Choose a %d-digit PIN", 4)
```