Localize your app

Android runs on many devices in many regions. To reach the most users, your app should handle text, audio files, numbers, currency, and graphics in ways appropriate to the locales where your app is used.

This document describes best practices for localizing Android apps.

You should already have a working knowledge of the Java programming language and be familiar with Android resource loading (https://developer.android.com/guide/topics/resources/providing-resources.html), the declaration of user interface elements in XML (https://developer.android.com/guide/topics/ui/declaring-layout), development considerations such as activity lifecycle (https://developer.android.com/guide/components/activities/activity-lifecycle), and general principles of internationalization and localization.

It is good practice to use the Android resource framework to separate the localized aspects of your app as much as possible from the core Java-based functionality:

- You can put most or all of the *contents* of your app's user interface into resource files, as described in this document and in Providing Resources (https://developer.android.com/guide/topics/resources/providing-resources.html).
- The *behavior* of the user interface, on the other hand, is driven by your Java-based code. For example, if users input data that needs to be formatted or sorted differently depending on locale, then you would use the Java programming language to handle the data programmatically. This document doesn't cover how to localize your Java-based code.

For a short guide to localizing strings in your app, see the training lesson, Supporting Different Languages (https://developer.android.com/training/basics/supporting-devices/languages.html).

Overview: Resource switching in Android

Resources are text strings, layouts, sounds, graphics, and any other static data that your Android app needs. An app can include multiple sets of resources, each customized for a different device configuration. When a user runs the app, Android automatically selects and loads the resources that best match the device.

(This document focuses on localization and locale. For a complete description of resource-switching and all the types of configurations that you can specify — screen orientation, touchscreen type, and so on — see Providing Alternative Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources).)

When you write your app, you create default and alternative resources for your app to use. When users run your app, the Android system selects which resources to load, based upon the device's locale. To create resources, you place files within specially named subdirectories of the project's `res/` directory.

Why default resources are important

Whenever the app runs in a locale for which you have not provided locale-specific text, Android loads the default strings from `res/values/strings.xml`. If this default file is absent, or if it's missing a string that your app needs, then your app doesn't run and shows an error. The example below illustrates what can happen when the default text file is incomplete.

Example:

An app's Java-based code refers to just two strings, `text_a` and `text_b`. This app includes a localized resource file (`res/values-en/strings.xml`) that defines `text_a` and `text_b` in English. This app also includes a default resource file (`res/values/strings.xml`) that includes a definition for `text_a`, but not for `text_b`:

- When this app is launched on a device with locale set to English, the app might run without a problem, because `res/values-en/strings.xml` contains both of the needed text strings.
- However, **the user sees an error message and a Force Close button** when this app is launched on a device set to a language other than English. The app doesn't load.

To prevent this situation, make sure that a `res/values/strings.xml` file exists and that it defines every needed string. The situation applies to all types of resources, not just strings: You need to create a set of default resource files containing all the resources that your app calls upon —layouts, drawables, animations, etc. For information about testing, see Testing for Default Resources (#test-for-default).

Using resources for localization

How to create default resources

Put the app's default text in **`res/values/strings.xml`**.

The text strings in **`res/values/strings.xml`** should use the default language, which is the language that you expect most of your app's users to speak.

The default resource set must also include any default drawables and layouts, and can include other types of resources such as animations:

- **`res/drawable/`**(required directory holding at least one graphic file, for the app's icon on Google Play)
- **`res/layout/`** (required directory holding an XML file that defines the default layout)
- **`res/anim/`** (required if you have any **`res/anim-`*`<qualifiers>`*** folders)
- **`res/xml/`** (required if you have any **`res/xml-`*`<qualifiers>`*** folders)
- **`res/raw/`** (required if you have any **`res/raw-`*`<qualifiers>`*** folders)

Tip: In your code, examine each reference to an Android resource. Make sure that a default resource is defined for each one. Also make sure that the default string file is complete: A *localized* string file can contain a subset of the strings, but the *default* string file must contain them all.

How to create alternative resources

A large part of localizing an app is providing alternative text for different languages. In some cases you also provide alternative graphics, sounds, layouts, and other locale-specific resources.

An app can specify many **`res/`*`<qualifiers>`*`/`** directories, each with different qualifiers. To create an alternative resource for a different locale, you use a qualifier that specifies a language or a language-region combination. (The name of a resource directory must conform to the naming scheme described in Providing Alternative Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources), or else your app cannot compile.)

Example:

Suppose that your app's default language is English. Suppose also that you want to localize all the text in your app to French, and most of the text in your app (everything except the

app's title) to Japanese. In this case, you could create three alternative `strings.xml` files, each stored in a locale-specific resource directory:

1. `res/values/strings.xml`
   Contains English text for all the strings that the app uses, including text for a string named `title`.
2. `res/values-fr/strings.xml`
   Contain French text for all the strings, including `title`.
3. `res/values-ja/strings.xml`
   Contain Japanese text for all the strings *except* `title`.

If your Java-based code refers to `R.string.title`, here is what happens at runtime:

- If the device is set to any language other than French, Android loads `title` from the `res/values/strings.xml` file.
- If the device is set to French, Android loads `title` from the `res/values-fr/strings.xml` file.

Notice that if the device is set to Japanese, Android looks for `title` in the `res/values-ja/strings.xml` file. But because no such string is included in that file, Android falls back to the default, and loads `title` in English from the `res/values/strings.xml` file.

Which resources take precedence?

If multiple resource files match a device's configuration, Android follows a set of rules in deciding which file to use. Among the qualifiers that can be specified in a resource directory name, **locale almost always takes precedence**.

Example:

Assume that an app includes a default set of graphics and two other sets of graphics, each optimized for a different device setup:

- `res/drawable/`
  Contains default graphics.
- `res/drawable-small-land-stylus/`
  Contains graphics optimized for use with a device that expects input from a stylus and has a QVGA low-density screen in landscape orientation.
- `res/drawable-ja/`
  Contains graphics optimized for use with Japanese.

If the app runs on a device that is configured to use Japanese, Android loads graphics from `res/drawable-ja/`, even if the device happens to be one that expects input from a stylus and has a QVGA low-density screen in landscape orientation.

Exception: The only qualifiers that take precedence over locale in the selection process are MCC and MNC (mobile country code and mobile network code).

Example:

Assume that you have the following situation:

- The app code calls for `R.string.text_a`
- Two relevant resource files are available:
  - `res/values-mcc404/strings.xml`, which includes `text_a` in the app's default language, in this case English.
  - `res/values-hi/strings.xml`, which includes `text_a` in Hindi.
- The app is running on a device that has the following configuration:
  - The SIM card is connected to a mobile network in India (MCC 404).
  - The language is set to Hindi (`hi`).

Android loads `text_a` from `res/values-mcc404/strings.xml` (in English), even if the device is configured for Hindi. That is because in the resource-selection process, Android prefers an MCC match over a language match.

The selection process isn't always as straightforward as these examples suggest. Please read How Android Finds the Best-matching Resource (https://developer.android.com/guide/topics/resources/providing-resources.html#BestMatch) for a more nuanced description of the process. All the qualifiers are described and listed in order of precedence in Table 2 of Providing Alternative Resources (https://developer.android.com/guide/topics/resources/providing-resources.html#table2).

Referring to resources in code

In your app's Java-based code, you refer to resources using the syntax `R.`*`resource_type`*`.`*`resource_name`* or `android.R.`*`resource_type`*`.`*`resource_name`*. For more about this, see Accessing Resources (https://developer.android.com/guide/topics/resources/accessing-resources.html).

Managing strings for localization

Move all strings into strings.xml

As you build your apps, don't hard code any string. Instead, declare *all* of your strings as resources in a default `strings.xml` file, which makes it easy to update and localize. Strings in the `strings.xml` file can then be easily extracted, translated, and integrated back into your app (with appropriate qualifiers) without any changes to the compiled code.

If you generate images with text, put those strings in `strings.xml` as well, and regenerate the images after translation.

Follow Android guidelines for UI strings

As you design and develop your UIs, make sure that you pay close attention to *how* you talk to your user. In general, use a succinct and compressed style (https://material.io/guidelines/style/writing.html) that is friendly but brief, and use a consistent style throughout your UIs.

Make sure that you read and follow the Material Design recommendations for writing style and word choice (https://material.io/guidelines/style/writing.html). Doing so makes your apps appear more polished to the user and helps users understand your UI more quickly.

Also, always use Android standard terminology wherever possible—such as for UI elements such as Action Bar, Options Menu, System Bar, Notifications, and so on. Using Android terms correctly and consistently makes translation easier and results in a better end-product for users.

Provide sufficient context for declared strings

As you declare strings in your `strings.xml` file, make sure to describe the context in which the string is used. This information is invaluable to the translator and result in better quality translation. It also helps you manage your strings more effectively.

Here is an example:

```
<!-- The action for submitting a form. This text is on a button that car
<string name="login_submit_button">Sign in</string>
```

Consider providing context information that may include:

- What is this string for? When and where is it presented to the user?

- Where is this in the layout? For example, translations are less flexible in buttons than in text boxes.

Mark message parts that should not be translated

Often strings contain contain text that should not be translated into other languages. Common examples might be a piece of code, a placeholder for a value, a special symbol, or a name. As you prepare your strings for translation, look for and mark text that should remain as-is, without translation, so that the translator doesn't change it.

To mark text that should not be translated, use an **<xliff:g>** placeholder tag. Here is an example tag that ensures the text "%1$s" isn't changed during translation (otherwise it could break the message):

```
<string name="countdown">
  <xliff:g id="time" example="5 days">%1$s</xliff:g>until holiday
</string>
```

When you declare a placeholder tag, always add an id attribute that explains what the placeholder is for. If your apps later replace the placeholder value, be sure to provide an example attribute to clarify the expected use.

Here are some more examples of placeholder tags:

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
<!-- Example placeholder for a special unicode symbol -->
<string name="star_rating">Check out our 5
    <xliff:g id="star">\u2605</xliff:g>
</string>
<!-- Example placeholder for a for a URL -->
<string name="app_homeurl">
    Visit us at <xliff:g
    id="application_homepage">http://my/app/home.html</xliff:g>
</string>
<!-- Example placeholder for a name -->
<string name="prod_name">
    Learn more at <xliff:g id="prod_gamegroup">Game Group</xliff:g>
</string>
<!-- Example placeholder for a literal -->
<string name="promo_message">
    Please use the "<xliff:g id="promotion_code">ABCDEFG</xliff:g>" to get a
</string>
...
</resources>
```

Localization checklist

For a complete overview of the process of localizing and distributing an Android app, see the Localization Checklist (https://developer.android.com/distribute/tools/localization-checklist.html) document.

Localization tips

Design your app to work in any locale

You cannot assume anything about the device on which a user runs your app. The device might have hardware that you were not anticipating, or it might be set to a locale that you did not plan for or that you cannot test. Design your app so that it functions normally or fails gracefully no matter what device it runs on.

Important: Make sure that your app includes a full set of default resources.

Make sure to include `res/drawable/` and a `res/values/` folders (without any additional modifiers in the folder names) that contain all the images and text that your app needs.

If an app is missing even one default resource, it doesn't run on a device that is set to an unsupported locale. For example, the `res/values/strings.xml` default file might lack one string that the app needs: When the app runs in an unsupported locale and attempts to load `res/values/strings.xml`, the user sees an error message and a Force Close button.

For more information, see Testing for Default Resources (#test-for-default).

Design a flexible layout

If you need to rearrange your layout to fit a certain language (for example German with its long words), you can create an alternative layout for that language (for example `res/layout-de/main.xml`). However, doing this can make your app harder to maintain. It is better to create a single layout that is more flexible.

Another typical situation is a language that requires something different in its layout. For example, you might have a contact form that should include two name fields when the app runs in Japanese, but three name fields when the app runs in some other language. You could handle this in either of two ways:

- Create one layout with a field that you can programmatically enable or disable, based on the language, or
- Have the main layout include another layout that includes the changeable field. The second layout can have different configurations for different languages.

Avoid creating more resource files and text strings than you need

You probably don't need to create a locale-specific alternative for every resource in your app. For example, the layout defined in the `res/layout/main.xml` file might work in any locale, in which case there would be no need to create any alternative layout files.

Also, you might not need to create alternative text for every string. For example, assume the following:

- Your app's default language is American English. Every string that the app uses is defined, using American English spellings, in `res/values/strings.xml`.
- For a few important phrases, you want to provide British English spelling. You want these alternative strings to be used when your app runs on a device in the United Kingdom.

To do this, you could create a small file called `res/values-en-rGB/strings.xml` that includes only the strings that should be different when the app runs in the U.K. For all the rest of the strings, the app falls back to the defaults and use what is defined in `res/values/strings.xml`.

Use the Android Context object for manual locale lookup

You can look up the locale using the `Context` (https://developer.android.com/reference/android/content/Context.html) object that Android makes available:

```
String locale = context.getResources().getConfiguration().locale.getDisp
```

Use the App Translation Service

The App Translation Service (https://support.google.com/l10n/answer/6359997) is integrated into the Play Console (https://support.google.com/l10n/answer/6341304), and it is also accessible from Android Studio (https://support.google.com/l10n/answer/6341928). It is a quick and simple way to get an instant quote and place an order with a translation company. You

can order translations into one or more languages for app UI strings, Play Store Listing text, IAP names, and ad campaign text.

Testing localized apps

Testing on a device

Keep in mind that the device you are testing may be significantly different from the devices available to consumers in other geographies. The locales available on your device may differ from those available on other devices. Also, the resolution and density of the device screen may differ, which could affect the display of strings and drawables in your UI.

To change the locale or language on a device, use the Settings app.

Testing on an emulator

For details about using the emulator, see Android Emulator (https://developer.android.com/tools/help/emulator.html).

Creating and using a custom locale

A "custom" locale is a language/region combination that the Android system image does not explicitly support. You can test how your app runs in a custom locale by creating a custom locale in the emulator. There are two ways to do this:

- Use the Custom Locale app, which is accessible from the app tab. (After you create a custom locale, switch to it by pressing and holding the locale name.)
- Change to a custom locale from the adb shell, as described below.

When you set the emulator to a locale that isn't available in the Android system image, the system itself displays in its default language. Your app, however, should localize properly.

Changing the emulator locale from the adb shell

To change the locale in the emulator by using the adb shell.

1. Pick the locale you want to test and determine its BCP-47 language tag, for example, Canadian French would be `fr-CA`.
2. Launch an emulator.

3. From a command-line shell on the host computer, run the following command:
   `adb shell`
   or if you have a device attached, specify that you want the emulator by adding the `-e` option:
   `adb -e shell`
4. At the adb shell prompt (#), run this command:
   `setprop persist.sys.locale [`*`BCP-47 language tag`*`];stop;sleep 5;start`
   Replace bracketed sections with the appropriate codes from Step 1.

For instance, to test in Canadian French:

```
setprop persist.sys.locale fr-CA;stop;sleep 5;start
```

This causes the emulator to restart. (It looks like a full reboot, but it isn't.) Once the Home screen appears again, re-launch your app, and the app launches with the new locale.

Testing for default resources

Here's how to test whether an app includes every string resource that it needs:

1. Set the emulator or device to a language that your app does not support. For example, if the app has French strings in `res/values-fr/` but does not have any Spanish strings in `res/values-es/`, then set the emulator's locale to Spanish. (You can use the Custom Locale app to set the emulator to an unsupported locale.)
2. Run the app.
3. If the app shows an error message and a Force Close button, it might be looking for a string that isn't available. Make sure that your `res/values/strings.xml` file includes a definition for every string that the app uses.

If the test is successful, repeat it for other types of configurations. For example, if the app has a layout file called `res/layout-land/main.xml` but does not contain a file called `res/layout-port/main.xml`, then set the emulator or device to portrait orientation and see if the app runs.

Next
Test your app with pseudolocales →
(https://developer.android.com/guide/topics/resources/pseudolocales)

Last updated April 17, 2018.

Twitter
Follow @AndroidDev on Twitter

Google+
Follow Android Developers on Google+

YouTube
Check out Android Developers on YouTube