Internationalizing Flutter apps

Docs Development a11y & i18n i18n

Contents

What you'll learn

- How to track the device's locale (the user's preferred language).
- How to manage locale-specific app values.
- How to define the locales an app supports.

If your app might be deployed to users who speak another language then you'll need to internationalize it. That means you need to write the app in a way that makes it possible to localize values like text and layouts for each language or locale that the app supports. Flutter provides widgets and classes that help with internationalization and the Flutter libraries themselves are internationalized.

This page covers concepts and workflows necessary to localize a Flutter application using the `MaterialApp` and `CupertinoApp`classes, as most apps are written that way. However, applications written using the lower level `WidgetsApp` class can also be internationalized using the same classes and logic.

Sample internationalized apps

If you'd like to start out by reading the code for an internationalized Flutter app, here are two small examples. The first one is intended to be as simple as possible, and the second one uses the APIs and tools provided by the `intl` package. If Dart's intl package is new to you, see Using the Dart intl tools.

- Minimal internationalization
- Internationalization based on the `intl` package

Introduction to localizations in Flutter

This section provides a tutorial on how to internationalize a Flutter application, along with any additional setup that a target platform might require.

Setting up an internationalized app: the Flutter _localizations package

By default, Flutter only provides US English localizations. To add support for other languages, an application must specify additional `MaterialApp` (or `CupertinoApp`) properties, and include a package called `flutter_localizations`. As of November 2020, this package supports 78 languages.

To use flutter_localizations, add the package as a dependency to your `pubspec.yaml` file:

content_copy

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations: # Add this line
    sdk: flutter         # Add this line
```

Next, run `pub get packages`, then import the `flutter_localizations` library and specify `localizationsDelegates` and `supportedLocales` for `MaterialApp`:

content_copy

```
import 'package:flutter_localizations/flutter_localizations.dart';
```

```
return const MaterialApp(
  title: 'Localizations Sample App',
  localizationsDelegates: [
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
    GlobalCupertinoLocalizations.delegate,
  ],
  supportedLocales: [
    Locale('en', ''), // English, no country code
    Locale('es', ''), // Spanish, no country code
  ],
  home: MyHomePage(),
);
```

content_copy

After introducing the `flutter_localizations` package and adding the code above, the `Material` and `Cupertino` packages should now be correctly localized in one of the 78 supported locales. Widgets should be adapted to the localized messages, along with correct left-to-right and right-to-left layout.

Try switching the target platform's locale to Spanish (`es`) and notice that the messages should be localized.

Apps based on `WidgetsApp` are similar except that the `GlobalMaterialLocalizations.delegate` isn't needed.

The full `Locale.fromSubtags` constructor is preferred as it supports `scriptCode`, though the `Locale` default constructor is still fully valid.

The elements of the `localizationsDelegates` list are factories that produce collections of localized values. `GlobalMaterialLocalizations.delegate` provides localized strings and other values for the Material Components library. `GlobalWidgetsLocalizations.delegate` defines the default text direction, either left-to-right or right-to-left, for the widgets library.

More information about these app properties, the types they depend on, and how internationalized Flutter apps are typically structured, can be found below.

Adding your own localized messages

Once the `flutter_localizations` package is added, use the following instructions to add localized text to your application.

1. Add the `intl` package to the `pubspec.yaml` file:

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
  intl: ^0.17.0 # Add this line
```

content_copy

2. Also, in the `pubspec.yaml` file, enable the `generate` flag. This is added to the section of the pubspec that is specific to Flutter, and usually comes later in the pubspec file.

```
# The following section is specific to Flutter.
flutter:
  generate: true # Add this line
```

content_copy

3. Add a new yaml file to the root directory of the Flutter project called `l10n.yaml` with the following content:

```
arb-dir: lib/l10n
template-arb-file: app_en.arb
output-localization-file: app_localizations.dart
```

content_copy

   This file configures the localization tool; in this example, the input files are located in `${FLUTTER_PROJECT}/lib/l10n`, the `app_en.arb` file provides the template, and the generated localizations are placed in the `app_localizations.dart` file.

4. In `${FLUTTER_PROJECT}/lib/l10n`, add the `app_en.arb` template file. For example:

```
{
    "helloWorld": "Hello World!",
    "@helloWorld": {
      "description": "The conventional newborn programmer greeting"
    }
}
```

content_copy

5. Next, add an `app_es.arb` file in the same directory for Spanish translation of the same message:

```
{
    "helloWorld": "¡Hola Mundo!"
}
```

content_copy

6. Now, run your app so that codegen takes place. You should see generated files in`${FLUTTER_PROJECT}/.dart_tool/flutter_gen/gen_l10n`.

7. Add the import statement on `app_localizations.dart` and `AppLocalizations.delegate` in your call to the constructor for `MaterialApp`.

```
import 'package:flutter_gen/gen_l10n/app_localizations.dart';
```

content_copy

```
return const MaterialApp(
  title: 'Localizations Sample App',
  localizationsDelegates: [
    AppLocalizations.delegate, // Add this line
    GlobalMaterialLocalizations.delegate,
    GlobalWidgetsLocalizations.delegate,
    GlobalCupertinoLocalizations.delegate,
  ],
  supportedLocales: [
    Locale('en', ''), // English, no country code
    Locale('es', ''), // Spanish, no country code
  ],
  home: MyHomePage(),
);
```
content_copy

8. Use AppLocalizations anywhere in your app. Here, the translated message is used in a Text widget.

```
Text(AppLocalizations.of(context)!.helloWorld);
```
content_copy

9. You can also use the generated `localizationsDelegates` and `supportedLocales` list instead of providing them manually.

```
const MaterialApp(
  title: 'Localizations Sample App',
  localizationsDelegates: AppLocalizations.localizationsDelegates,
  supportedLocales: AppLocalizations.supportedLocales,
);
```
content_copy

This code generates a Text widget that displays “Hello World!” if the target device’s locale is set to English, and “¡Hola Mundo!” if the target device’s locale is set to Spanish. In the `arb` files, the key of each entry is used as the method name of the getter, while the value of that entry contains the localized message.

To see a sample Flutter app using this tool, please see `gen_l10n_example`.

To localize your device app description, you can pass in the localized string into `MaterialApp.onGenerateTitle`:

```
return MaterialApp(
  onGenerateTitle: (BuildContext context) =>
      DemoLocalizations.of(context).title,
```
content_copy

For more information about the localization tool, such as dealing with DateTime and handling plurals, see the Internationalization User’s Guide.

Localizing for iOS: Updating the iOS app bundle

iOS applications define key application metadata, including supported locales, in an `Info.plist` file that is built into the application bundle. To configure the locales supported by your app, use the following instructions:

1. Open your project's `ios/Runner.xcworkspace` Xcode file.
2. In the **Project Navigator**, open the `Info.plist` file under the `Runner` project's `Runner` folder.
3. Select the **Information Property List** item. Then select **Add Item** from the **Editor** menu, and select **Localizations** from the pop-up menu.
4. Select and expand the newly-created `Localizations` item. For each locale your application supports, add a new item and select the locale you wish to add from the pop-up menu in the **Value** field. This list should be consistent with the languages listed in the supportedLocales parameter.
5. Once all supported locales have been added, save the file.

Advanced topics for further customization

This section covers additional ways to customize a localized Flutter application.

Advanced locale definition

Some languages with multiple variants require more than just a language code to properly differentiate.

For example, fully differentiating all variants of Chinese requires specifying the language code, script code, and country code. This is due to the existence of simplified and traditional script, as well as regional differences in the way characters are written within the same script type.

In order to fully express every variant of Chinese for the country codes `CN`, `TW`, and `HK`, the list of supported locales should include:

```
supportedLocales: [
  Locale.fromSubtags(languageCode: 'zh'), // generic Chinese 'zh'
  Locale.fromSubtags(
      languageCode: 'zh',
      scriptCode: 'Hans'), // generic simplified Chinese 'zh_Hans'
  Locale.fromSubtags(
      languageCode: 'zh',
      scriptCode: 'Hant'), // generic traditional Chinese 'zh_Hant'
  Locale.fromSubtags(
      languageCode: 'zh',
      scriptCode: 'Hans',
      countryCode: 'CN'), // 'zh_Hans_CN'
  Locale.fromSubtags(
      languageCode: 'zh',
      scriptCode: 'Hant',
      countryCode: 'TW'), // 'zh_Hant_TW'
  Locale.fromSubtags(
      languageCode: 'zh',
      scriptCode: 'Hant',
      countryCode: 'HK'), // 'zh_Hant_HK'
],
```
content_copy

This explicit full definition ensures that your app can distinguish between and provide the fully nuanced localized content to all combinations of these country codes. If a user's preferred locale is not specified, then the closest match is used instead, which likely contains differences to what the user expects. Flutter only resolves to locales defined in `supportedLocales`. Flutter provides scriptCode–differentiated localized content for commonly used languages. See `Localizations` for information on how the supported locales and the preferred locales are resolved.

Although Chinese is a primary example, other languages like French (`fr_FR`, `fr_CA`) should also be fully differentiated for more nuanced localization.

Tracking the locale: The Locale class and the Localizations widget

The `Locale` class identifies the user's language. Mobile devices support setting the locale for all applications, usually using a system settings menu. Internationalized apps respond by displaying values that are locale–specific. For example, if the user switches the device's locale from English to French, then a `Text` widget that originally displayed "Hello World" would be rebuilt with "Bonjour le monde".

The `Localizations` widget defines the locale for its child and the localized resources that the child depends on. The `WidgetsApp`widget creates a `Localizations` widget and rebuilds it if the system's locale changes.

You can always lookup an app's current locale with `Localizations.localeOf()`:

```
Locale myLocale = Localizations.localeOf(context);
```
content_copy

Specifying the app's supportedLocales parameter

Although the `flutter_localizations` library currently supports 78 languages and language variants, only English language translations are available by default. It's up to the developer to decide exactly which languages to support.

The `MaterialApp` `supportedLocales` parameter limits locale changes. When the user changes the locale setting on their device, the app's `Localizations` widget only follows suit if the new locale is a member of this list. If an exact match for the device locale isn't found, then the first supported locale with a matching `languageCode` is used. If that fails, then the first element of the`supportedLocales` list is used.

An app that wants to use a different "locale resolution" method can provide a `localeResolutionCallback`. For example, to have your app unconditionally accept whatever locale the user selects:

content_copy

```
MaterialApp(
  localeResolutionCallback: (
    Locale? locale,
    Iterable<Locale> supportedLocales,
  ) {
    return locale;
  },
);
```

How internationalization in Flutter works

This section covers the technical details of how localizations work in Flutter. If you're planning on supporting your own set of localized messages, the following content would be helpful. Otherwise, you can skip this section.

Loading and retrieving localized values

The `Localizations` widget is used to load and lookup objects that contain collections of localized values. Apps refer to these objects with `Localizations.of(context,type)`. If the device's locale changes, the `Localizations` widget automatically loads values for the new locale and then rebuilds widgets that used it. This happens because `Localizations` works like an`InheritedWidget`. When a build function refers to an inherited widget, an implicit dependency on the inherited widget is created. When an inherited widget changes (when the `Localizations` widget's locale changes), its dependent contexts are rebuilt.

Localized values are loaded by the `Localizations` widget's list of `LocalizationsDelegate`s. Each delegate must define an asynchronous `load()` method that produces an object that encapsulates a collection of localized values. Typically these objects define one method per localized value.

In a large app, different modules or packages might be bundled with their own localizations. That's why the `Localizations` widget manages a table of objects, one per `LocalizationsDelegate`. To retrieve the object produced by one of the `LocalizationsDelegate`'s `load` methods, you specify a `BuildContext` and the object's type.

For example, the localized strings for the Material Components widgets are defined by the `MaterialLocalizations` class. Instances of this class are created by a `LocalizationDelegate` provided by the `MaterialApp` class. They can be retrieved with `Localizations.of()`:

```
Localizations.of<MaterialLocalizations>(context, MaterialLocalizations);
```
content_copy

This particular `Localizations.of()` expression is used frequently, so the `MaterialLocalizations` class provides a convenient shorthand:

```
static MaterialLocalizations of(BuildContext context) {
  return Localizations.of<MaterialLocalizations>(context,
MaterialLocalizations);
}

/// References to the localized values defined by MaterialLocalizations
/// are typically written like this:

tooltip: MaterialLocalizations.of(context).backButtonTooltip,
```
content_copy

Defining a class for the app's localized resources

Putting together an internationalized Flutter app usually starts with the class that encapsulates the app's localized values. The example that follows is typical of such classes.

Complete source code for the `intl_example` for this app.

This example is based on the APIs and tools provided by the `intl` package. An alternative class for the app's localized resourcesdescribes an example that doesn't depend on the `intl` package.

The `DemoLocalizations` class contains the app's strings (just one for the example) translated into the locales that the app supports. It uses the `initializeMessages()` function generated by Dart's `intl` package, `Intl.message()`, to look them up.

```
class DemoLocalizations {
  DemoLocalizations(this.localeName);

  static Future<DemoLocalizations> load(Locale locale) {
    final String name =
        locale.countryCode == null || locale.countryCode!.isEmpty
            ? locale.languageCode
            : locale.toString();
    final String localeName = Intl.canonicalizedLocale(name);

    return initializeMessages(localeName).then((_) {
      return DemoLocalizations(localeName);
    });
  }

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context, DemoLocalizations)!;
  }

  final String localeName;

  String get title {
    return Intl.message(
      'Hello World',
      name: 'title',
      desc: 'Title for the Demo application',
      locale: localeName,
    );
  }
}
```

content_copy

A class based on the `intl` package imports a generated message catalog that provides the `initializeMessages()` function and the per-locale backing store for `Intl.message()`. The message catalog is produced by an `intl` tool that analyzes the source code for classes that contain `Intl.message()` calls. In this case that would just be the `DemoLocalizations` class.

Adding support for a new language

An app that needs to support a language that's not included in `GlobalMaterialLocalizations` has to do some extra work: it must provide about 70 translations ("localizations") for words or phrases and the date patterns and symbols for the locale.

See the following for an example of how to add support for the Norwegian Nynorsk language.

A new `GlobalMaterialLocalizations` subclass defines the localizations that the Material library depends on. A new `LocalizationsDelegate` subclass, which serves as factory for the `GlobalMaterialLocalizations` subclass, must also be defined.

Here's the source code for the complete `add_language` example, minus the actual Nynorsk translations.

The locale-specific `GlobalMaterialLocalizations` subclass is called `NnMaterialLocalizations`, and the `LocalizationsDelegate`subclass is `_NnMaterialLocalizationsDelegate`. The value of `NnMaterialLocalizations.delegate` is an instance of the delegate, and is all that's needed by an app that uses these localizations.

The delegate class includes basic date and number format localizations. All of the other localizations are defined by `String` valued property getters in `NnMaterialLocalizations`, like this:

```
@override
String get moreButtonTooltip => r'More';

@override
String get aboutListTileTitleRaw => r'About $applicationName';

@override
String get alertDialogLabel => r'Alert';
```

content_copy

These are the English translations, of course. To complete the job you need to change the return value of each getter to an appropriate Nynorsk string.

The getters return "raw" Dart strings that have an r prefix, like `r'About $applicationName'`, because sometimes the strings contain variables with a `$` prefix. The variables are expanded by parameterized localization methods:

```
@override
String get pageRowsInfoTitleRaw => r'$firstRow–$lastRow of $rowCount';

@override
String get pageRowsInfoTitleApproximateRaw =>
    r'$firstRow–$lastRow of about $rowCount';
```

content_copy

The date patterns and symbols of the locale will also need to be specified. In the source code, the date patterns and symbols are defined like this:

```
const nnLocaleDatePatterns = {
  'd': 'd.',
  'E': 'ccc',
  'EEEE': 'cccc',
  'LLL': 'LLL',
  // ...
}
```

content_copy

```
const nnDateSymbols = {
  'NAME': 'nn',
  'ERAS': <dynamic>[
    'f.Kr.',
    'e.Kr.',
  ],
  // ...
}
```

content_copy

These will need to be modified for the locale to use the correct date formatting. Unfortunately, since the `intl` library does not share the same flexibility for number formatting, the formatting for an existing locale will have to be used as a substitute in `_NnMaterialLocalizationsDelegate`:

content_copy

```
class _NnMaterialLocalizationsDelegate
    extends LocalizationsDelegate<MaterialLocalizations> {
  const _NnMaterialLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) => locale.languageCode == 'nn';

  @override
  Future<MaterialLocalizations> load(Locale locale) async {
    final String localeName = intl.Intl.canonicalizedLocale(locale.toString());

    // The locale (in this case `nn`) needs to be initialized into the custom
    // date symbols and patterns setup that Flutter uses.
    date_symbol_data_custom.initializeDateFormattingCustom(
      locale: localeName,
      patterns: nnLocaleDatePatterns,
      symbols: intl.DateSymbols.deserializeFromMap(nnDateSymbols),
    );

    return SynchronousFuture<MaterialLocalizations>(
      NnMaterialLocalizations(
        localeName: localeName,
        // The `intl` library's NumberFormat class is generated from CLDR data
        // (see https://github.com/dart-
lang/intl/blob/master/lib/number_symbols_data.dart).
        // Unfortunately, there is no way to use a locale that isn't defined in
        // this map and the only way to work around this is to use a listed
        // locale's NumberFormat symbols. So, here we use the number formats
        // for 'en_US' instead.
        decimalFormat: intl.NumberFormat('#,##0.###', 'en_US'),
        twoDigitZeroPaddedFormat: intl.NumberFormat('00', 'en_US'),
        // DateFormat here will use the symbols and patterns provided in the
        // `date_symbol_data_custom.initializeDateFormattingCustom` call above.
        // However, an alternative is to simply use a supported locale's
        // DateFormat symbols, similar to NumberFormat above.
        fullYearFormat: intl.DateFormat('y', localeName),
        compactDateFormat: intl.DateFormat('yMd', localeName),
        shortDateFormat: intl.DateFormat('yMMMd', localeName),
        mediumDateFormat: intl.DateFormat('EEE, MMM d', localeName),
        longDateFormat: intl.DateFormat('EEEE, MMMM d, y', localeName),
        yearMonthFormat: intl.DateFormat('MMMM y', localeName),
        shortMonthDayFormat: intl.DateFormat('MMM d'),
      ),
    );
  }

  @override
  bool shouldReload(_NnMaterialLocalizationsDelegate old) => false;
}
```

For more information about localization strings, see the flutter_localizations README.

Once you've implemented your language-specific subclasses of `GlobalMaterialLocalizations` and `LocalizationsDelegate`, you just need to add the language and a delegate instance to your app. Here's some code that sets the app's

language to Nynorsk and adds the `NnMaterialLocalizations` delegate instance to the app's `localizationsDelegates` list:

content_copy

```
const MaterialApp(
  localizationsDelegates: [
    GlobalWidgetsLocalizations.delegate,
    GlobalMaterialLocalizations.delegate,
    NnMaterialLocalizations.delegate, // Add the newly created delegate
  ],
  supportedLocales: [
    Locale('en', 'US'),
    Locale('nn'),
  ],
  home: Home(),
),
```

Alternative internationalization workflows

This section describes different approaches to internationalize your Flutter application.

An alternative class for the app's localized resources

The previous DemoApp example was defined in terms of the Dart `intl` package. Developers can choose their own approach for managing localized values for the sake of simplicity or perhaps to integrate with a different i18n framework.

Complete source code for the `minimal` app.

In this version of DemoApp the class that contains the app's localizations, DemoLocalizations, includes all of its translations directly in per language Maps.

```
class DemoLocalizations {
  DemoLocalizations(this.locale);

  final Locale locale;

  static DemoLocalizations of(BuildContext context) {
    return Localizations.of<DemoLocalizations>(context, DemoLocalizations)!;
  }

  static const _localizedValues = <String, Map<String, String>>{
    'en': {
      'title': 'Hello World',
    },
    'es': {
      'title': 'Hola Mundo',
    },
  };

  static List<String> languages ()=> _localizedValues.keys.toList();

  String get title {
    return _localizedValues[locale.languageCode]!['title']!;
  }
}
```

content_copy

In the minimal app the `DemoLocalizationsDelegate` is slightly different. Its `load` method returns a `SynchronousFuture` because no asynchronous loading needs to take place.

```
class DemoLocalizationsDelegate
    extends LocalizationsDelegate<DemoLocalizations> {
  const DemoLocalizationsDelegate();

  @override
  bool isSupported(Locale locale) =>
DemoLocalizations.languages().contains(locale.languageCode);


  @override
  Future<DemoLocalizations> load(Locale locale) {
    // Returning a SynchronousFuture here because an async "load" operation
    // isn't needed to produce an instance of DemoLocalizations.
    return SynchronousFuture<DemoLocalizations>(DemoLocalizations(locale));
  }

  @override
  bool shouldReload(DemoLocalizationsDelegate old) => false;
}
```

content_copy

Using the Dart intl tools

Before building an API using the Dart `intl` package you'll want to review the `intl` package's documentation. Here's a summary of the process for localizing an app that depends on the `intl` package.

The demo app depends on a generated source file called `l10n/messages_all.dart`, which defines all of the localizable strings used by the app.

Rebuilding `l10n/messages_all.dart` requires two steps.

1. With the app's root directory as the current directory, generate `l10n/intl_messages.arb` from `lib/main.dart`:

```
$ flutter pub run intl_translation:extract_to_arb --output-dir=lib/l10n
lib/main.dart
```
content_copy

   The `intl_messages.arb` file is a JSON format map with one entry for each `Intl.message()` function defined in `main.dart`. This file serves as a template for the English and Spanish translations, `intl_en.arb` and `intl_es.arb`. These translations are created by you, the developer.

2. With the app's root directory as the current directory, generate `intl_messages_<locale>.dart` for each `intl_<locale>.arb` file and `intl_messages_all.dart`, which imports all of the messages files:

```
$ flutter pub run intl_translation:generate_from_arb \
    --output-dir=lib/l10n --no-use-deferred-loading \
    lib/main.dart lib/l10n/intl_*.arb
```
content_copy

   Windows does not support file name wildcarding. Instead, list the .arb files that were generated by the`intl_translation:extract_to_arb` command.

```
$ flutter pub run intl_translation:generate_from_arb \
    --output-dir=lib/l10n --no-use-deferred-loading \
    lib/main.dart \
    lib/l10n/intl_en.arb lib/l10n/intl_fr.arb lib/l10n/intl_messages.arb
```
content_copy

   The `DemoLocalizations` class uses the generated `initializeMessages()` function (defined in `intl_messages_all.dart`) to load the localized messages and `Intl.message()` to look them up.