PUBLICLY SHARED

Flutter
Internationalization User Guide

SUMMARY

Guide for building internationalized Flutter apps

Authors: Hans Muller (@hansmuller), Shi Hao Hong (@shihaohong)
Go Link: flutter.dev/go/i18n-user-guide
Created: April 2020 / **Last updated:** September 2020

Glossary

Locale - a compact representation of a written language. The Locale class represents locales, for example Locale('en', 'US') represents English as it's written in the United States.

Localization - the process of adding support for one or more locales to an application. An app supports a locale when the text the app displays is in the corresponding language.

Internationalization - the process setting up an app so that it can be localized.

Introduction

We will refer to an app's "user-facing strings" as messages, and refer to its complete list of user facing strings as its message catalog. A Flutter application is localized by defining a version of its message catalog for each locale that the application supports. The localized value of a message is often referred to as its "translation".

PUBLICLY SHARED

Flutter apps define message catalogs with "Application Resource Bundles": JSON format files with a ".arb" filename extension. The messages themselves are defined using a subset of a syntax called ICU, which is supported by many organizations and tools, like Google, Apple, IBM, ICU4J (Java), and ICU4C (C, C++).

Although this may all sound like a bit much, it's not complicated in practice. The following file, called "app_en.arb", defines a message catalog with a single message called helloWorld.

```
{
  "@@locale": "en",

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

The first line indicates that the catalog defines localizations of messages for English.

The "helloWorld" line defines the English translation for the app's helloWorld message, which is "Hello World!".

The JSON object that follows "@helloWorld" contains a description of the message that's intended to help translators. It also becomes a comment in the method generated for the message.

A Flutter application can look up a message's translation by using the generated AppLocalizations class.

```
Text(AppLocalizations.of(context).helloWorld)
```

Here, we've created a Text widget that will display a localized version of the helloWorld message. The static AppLocalizations.of() method looks up the message catalog for the locale of the current BuildContext, and returns the translation.

Setting up an internationalized application

Create a new Flutter application in a directory of your choice with the `flutter create` command:

```
flutter create <name_of_flutter_app>
```

The following sections describe the process for setting up an internationalized application in more detail. Here's a quick outline of the process:

- Add the i18n dependencies to your app's pubspec.yaml file.
- Create a new configuration file for localizations called l10n.yaml.
- Create a new "template" message catalog, like lib/l10n/app_en.arb.
- When the application is run, a new class that provides access to the message catalog will be generated automatically. You'll import this class.
- (optional) Internationalizing iOS applications

Update the pubspec.yaml file

Update the Flutter project's pubspec.yaml to include the `flutter_localizations` and `intl` packages. These packages will be used by your Flutter application and by the code that the localizations tool will generate. In your pubspec.yaml, add the following:

```
dependencies:
  flutter:
    sdk: flutter
  # Internationalization support.
  flutter_localizations:
    sdk: flutter
  intl: 0.16.1
  # the rest of your dependencies
flutter:
  # Adds code generation (synthetic package) support
  generate: true
```

Create the l10n.yaml file

In the root directory of your flutter application, create a new `l10n.yaml` file that contains the following:

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

The `l10n.yaml`configuration file is used to customize the tool that generates the localization classes your application will import.

- `arb-dir` specifies the directory where the tool expects to find input files. This directory will contain ".arb" format message catalogs.
- `template-arb-file` A message catalog that defines all of the messages your application supports as well as metadata for each message. This file must be created in the arb-dir.
- `output-localization-file` defines the main Dart class file that the tool will generate and your application will import. All of the files generated by the tool will be generated in `arb-dir`.

The l10n.yaml file supports many other configuration options. See the “Configuring the l10n code generator: The l10n.yaml file” below for details.

Create the template ARB file, lib/l10n/app_en.arb

Create the arb-dir directory and template-arb-file specified in the top-level l10n.yaml file. The template-arb-file is an ARB format message catalog for one, convenient, locale which defines all of the messages that the application supports.

Note: Filenames for all arb files cannot contain underscores other than for describing the locale. The internationalization tool uses underscores to parse out the language, country, and script codes for each arb file.

We could start with the same sample English message catalog discussed earlier.

```
{
  "@@locale": "en",

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

The name of each message in the template catalog will become the name of the Dart method that the application will use to retrieve the localized value of that message. As noted below, message names like “helloWorld” must be valid Dart method names.

Integrating the automatically generated localizations class

The final setup step is to import the generated `app_localizations.dart` file and integrate the AppLocalizations class with your Flutter app:

```
import 'package:flutter/material.dart';
import 'package:flutter_gen/gen_l10n/app_localizations.dart'; // Add this
line.

void main() {
 runApp(MyApp());
}

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     // Add the `localizationsDelegate` and `supportedLocales` lines.
     localizationsDelegates: AppLocalizations.localizationsDelegates,
```

```
      supportedLocales: AppLocalizations.supportedLocales,
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Now the application can look up localizations messages using the AppLocalizations class. For example to give the MyHomePage's AppBar a localized title:

```
class _MyHomePageState extends State<MyHomePage> {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       // Replace the title line with the following:
       title: Text(AppLocalizations.of(context).helloWorld),
     ),
    // The rest of the widget tree.
  );
}
```

The AppLocalizations class itself imports one additional class per supported locale, i.e. one additional class for each message catalog in the output directory.

The automatically generated localizations class, AppLocalizations

The tool that generates the localizations class will execute automatically each time the application is run or restarted as part of the build process. By default, the name of this class is AppLocalizations and you'll find it in `.dart_tool/flutter_gen/gen_l10n/app_localizations.dart`. This means that the generated code will not be checked into version source control, which is by design since the code is an artifact that's only needed when your Flutter app is built.

When running the Flutter app, the IDE may present you with a warning indicating that build errors exist in your project. This is because the localizations code will need to be generated for the first time (note that the warnings should be about the missing import and AppLocalizations class not existing). Proceed to run the Flutter application, which should generate your localizations code. For example, in VSCode, you can run the Flutter app by selecting “Debug Anyway” in the build error dialog that appears.

Starting with the sample template message catalog, upon successfully building the application, you'll find two files in the output directory: `.dart_tool/flutter_gen/gen_l10n/app_localizations_en.dart`, and `.dart_tool/flutter_gen/gen_l10n/app_localizations.dart`.

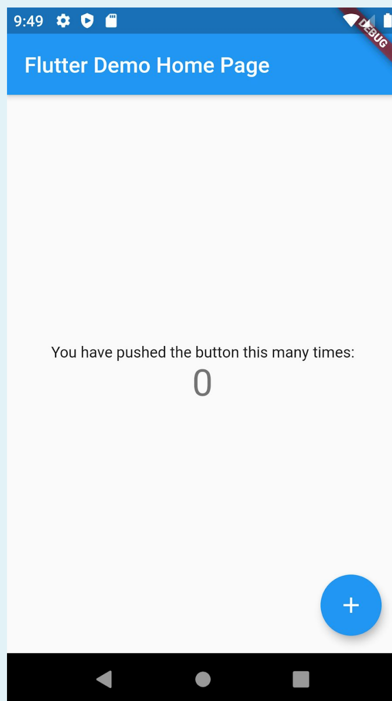
The AppLocalizations class, defined in app_localizations.dart, dispatches message

lookups to the appropriate locale-specific class, based on the locale requested by application. At this point there's just one locale-specific class, and it's defined in app_localizations_en.dart.

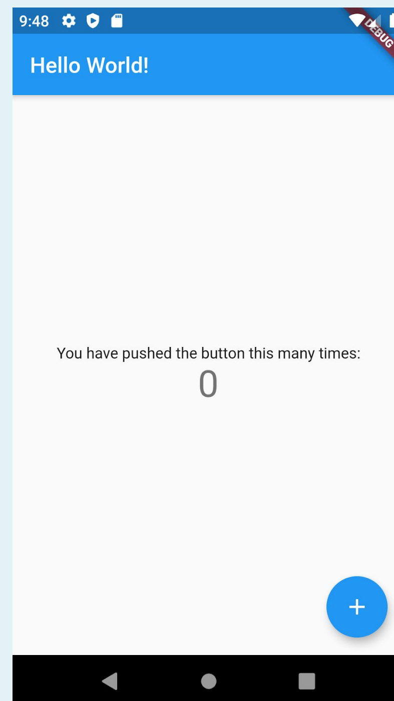
If these files were not generated, verify that there are no errors in the Flutter app, and review the steps to ensure that everything has been done correctly.

Upon running the app, you'll see "Hello World!" in the application's AppBar:

Before:



After:



Adding support for a new locale

To add a Spanish message catalog, create a new file, `lib/l10n/app_es.arb`, and add the following:

```
{
  "@@locale": "es",
  "helloWorld": "Hola Mundo!"
}
```

This file is just like the template message catalog, except that it only contains translations, it does not contain meta information.

Now, try hot reloading the application. This will generate the `app_strings_es.dart` file, which will contain the Spanish strings that your application will use. Nothing should have changed in your Flutter application (except if your test device's locale was already set to Spanish). In the next section, we'll go into more detail about how

PUBLICLY SHARED
to change the app's locale in Flutter.

(optional) Internationalizing iOS Flutter Applications

If you're testing on an iOS device or plan to support iOS devices in your Flutter application, you will need to correctly update the iOS app bundle with a list consistent with the languages listed in AppLocalizations.supportedLocales. If you plan to follow this user guide with an iOS test device, be sure to add "en" and "es" into that list.

Running An Internationalized App

After setting up your Flutter application to handle internationalization, you will need to either update the test device's locale, or use `Localizations.override` to see the localized messages.

While updating the test device's locale is the most common use-case, let's go ahead and use `Localizations.override` first to verify that the localized messages appear correctly in the Flutter application, since using `Localizations.override` only requires a code change to your Flutter application to see results. On the other hand, the steps for changing a test device's locale may vary by platform.

Localizations.override

Localizations.override is a factory constructor for the `Localizations` widget that allows for (the usually rare) situation where a section of your application needs to be localized to a different locale than the locale configured for your device.

Let's add some code to the body of your Flutter application to observe this behavior:

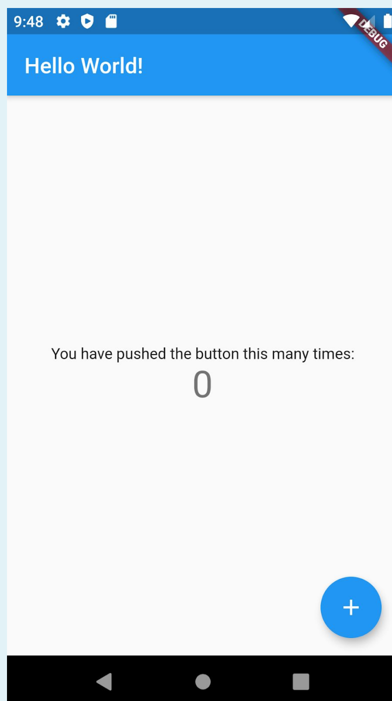
```
Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(AppLocalizations.of(context).helloWorld),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            // New code
            Localizations.override(
              context: context,
              locale: const Locale('es'),
              // Using a Builder here to get the correct BuildContext.
              child: Builder(
                builder: (BuildContext context) {
                  return Text(AppLocalizations.of(context).helloWorld);
                }
```

PUBLICLY SHARED

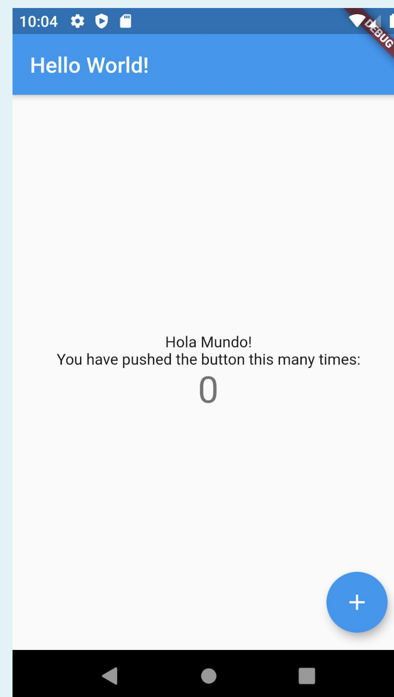
```
                ),
              ),
              // ...
            ],
          ),
        ),
        // ...
      );
    }
}
```

Upon hot-reloading, you will observe that the same call to `AppLocalizations.of(context).helloWorld` nested in `Localizations.override` returns the Spanish string, while `AppLocalizations.of(context).helloWorld` should return the English string if the test device's locale was set to a non-Spanish one.

Before:



After:

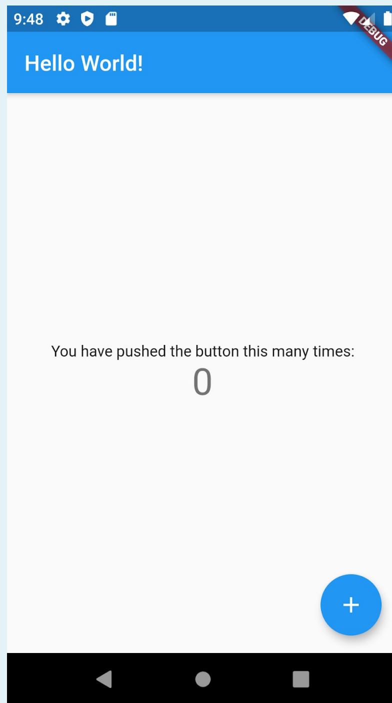


Updating the test device's locale

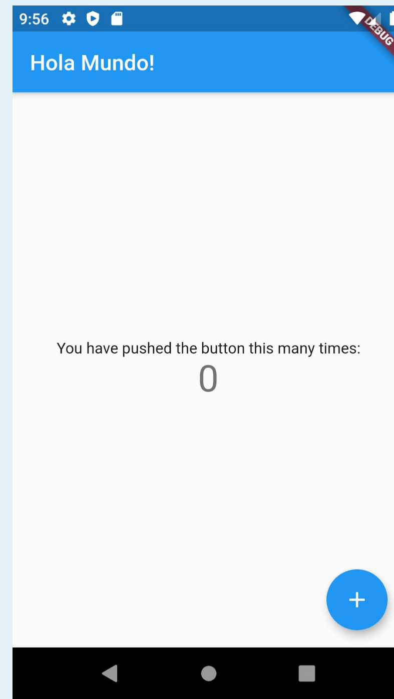
Since the steps to update a test device's locale will be different based on the platform you're working with, we will omit those steps and leave it to you to figure that out based on the test device you're using.

After updating a platform if you change your device's locale to Spanish, you will see the following:

Before:



After:



Defining Messages and Message Catalogs

All of the messages for one locale are defined in a single ".arb" file that contains one JSON object. Typically the object's first name/value pair defines the file's locale:

```
{
  "@@locale": "en",
  … message definitions
}
```

The filename can also define the catalog's locale if it ends in an underscore followed by a locale name. For example app_en_US.arb is a message catalog for the "en_US" locale. Typically apps define the locale redundantly, with both the filename suffix and the "@@locale" entry.

Each of the app's messages must have a name that's unique relative to the message catalog. The names must be suitable as Dart method names: camel case, beginning with a lowercase letter.

Each message should be defined by two name value pairs. The first names the message and specifies its translation. The second must have the same name as the first with a single '@' prefix. Its value is a JSON object that describes the message for the sake of translators and the code generation tool.

```
{
```

```
  "@@locale": "en",

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

All messages called “foo” must have a companion “@foo” entry. Although the value of @foo can be empty, it’s a good practice to include a description. Messages with parameters require some additional @foo properties, as we’ll see below.

Defining and Using Simple Messages

A simple message has no parameters. The message’s translation is exposed by a String-valued get method on the generated AppLocalizations class. So, as noted earlier, a message named helloWorld is defined like this:

```
{
  "@@locale": "en",

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

And used like this:

```
AppLocalizations.of(context).helloWorld
```

Translations can include special characters like newlines, if the usual JSON escapes are used:

- \b for backspace
- \f for form feed
- \n for newline
- \r for carriage return
- \t for tab
- \” for double quote
- \\ for backslash

For example:

```
{
  "@@locale": "en",
```

```
  "helloWorld": "Hello\nWorld",
  "@helloWorld": {
    "description": "The conventional newborn programmer greeting"
  }
}
```

Messages With Parameters

It's often useful to include application values in messages. In the catalog, message parameters are defined with "placeholders": parameter names bracketed with curly braces. These placeholders become positional method parameters in the generated AppLocalizations class. Placeholder names must be valid Dart method parameter names.

Each placeholder must be defined in the "placeholders" object. For example, to define a hello message with a userName parameter:

```
"hello": "Hello {userName}",
"@hello": {
  "description": "A message with a single parameter",
  "placeholders": {
    "userName": {
      "type": "String",
      "example": "Bob"
    }
  }
}
```

The userName parameter has type String. The generated hello() method returns a String:

```
AppLocalizations.of(context).hello(myUserName)
```

If a placeholder's type is not defined, then the corresponding parameter has type Object, and its String value is computed with toString().

The placeholder's example value is intended to help translators. In the future it might be used in generated tests.

Messages can have as many parameters as you like, although too many parameters can make creating good translations difficult.

Here's an example with two parameters. The types of the parameters aren't given, so they'll be Object in the generated greeting() method. The greeting() method will convert the parameters to String with toString().

```
"greeting": "{hello} {world}",
"@greeting": {
  "description": "A message with a two parameters",
  "placeholders": {
    "hello": {},
    "world": {}
  }
},
```

Just passing String valued arguments to the greeting() method is fine, because String.toString() is essentially a no-op.

```
AppLocalizations.of(context).greeting('Hello', 'World')
```

Messages With Numbers and Currencies

Numbers and numbers that represent currency values are displayed very differently in different locales. The Dart intl package provides support for formatting the strings they're converted to.

The localizations generation tool makes use of the intl package NumberFormat class to properly format numbers based on the locale and the desired format. For example, the following expression produces the string "1.2million":

```
NumberFormat.compactLong("en_US").format(1200000)
```

You can incorporate this format in a message with a double or int placeholder like this:

```
"numberOfDataPoints": "Number of data points: {value}",
"@numberOfDataPoints": {
  "description": "A message with a formatted int parameter",
  "placeholders": {
    "value": {
      "type": "int",
      "format": "compactLong"
    }
  }
}
```

In an app, when the locale is US English, the following expression would produce "Number of data points: 1.2million":

```
AppLocalizations.of(context).numberOfDataPoints(1200000)
```

The “format” for placeholders whose type is int or double, can be any one of the following NumberFormat named constructors.

Message “format” value	Output for numberOfDataPoints(1200000)
"compact"	"1.2M"
"compactCurrency"*	"$1.2M"
"compactSimpleCurrency"*	"$1.2M"
"compactLong"	"1.2 million"
"currency"*	"USD1,200,000.00"
"decimalPattern"	"1,200,000"
"decimalPercentPattern"*	"120,000,000%"
"percentPattern"	"120,000,000%"
"scientificPattern"	"1E6"
"simpleCurrency"*	"$1,200,000.00"

The five starred (“*”) NumberFormat constructors have optional, named parameters. Those parameters can be specified as the value of the placeholder’s “optionalParameters” object. For example, to specify the optional decimalDigits parameter for "compactCurrency":

```
"numberOfDataPoints": "Number of data points: {value}",
"@numberOfDataPoints": {
  "description": "A message with a formatted int parameter",
  "placeholders": {
    "value": {
      "type": "int",
      "format": "compactCurrency",
       "optionalParameters": {
         "decimalDigits": 2
       }
    }
  }
}
```

In this example the numberOfDataPoints() expression would produce: “USD1.20M”.

PUBLICLY SHARED

Messages With Dates

Dates strings are formatted in many different ways depending both the locale and the app's needs.

DateTime placeholder values are formatted with the DateFormat class from Dart's intl package. There are 41 format variations, identified by the names of their DateFormat factory constructors. In the following example, the DateTime value that appears in the helloWorldOn message is formatted with DartFormat.yMd:

```
"helloWorldOn": "Hello World on {date}",
"@helloWorldOn": {
  "description": "A message with a date parameter",
  "placeholders": {
    "date": {
      "type": "DateTime",
      "format": "yMd"
    }
  }
}
```

In an app, when the locale is US English, the following expression would produce "7/10/1996". If the locale was 'Russian', then it would produce "10.07.1996".

```
AppLocalizations.of(context).helloWorldOn(DateTime.utc(1996, 7, 10))
```

Messages With Plurals

There are a remarkable number of locale-specific rules for expressing plurals, for example, see the Unicode summary. Flutter relies on Dart intl package's Intl.plural method to handle all of the variations, so defining and using messages that incorporate plurals is straightforward.

A plural message must have an int parameter that represents the number of items the message is referring to. The value of this parameter must be greater than or equal to zero. The message must define between 1 and 6 variations, which depend on the int parameter's value. Each variation has a standard name and only the "other" variation is required. The table below shows how each variation is formatted. Each variation has a prefix that identifies the variation and a message variation bracketed with curly braces. Any variation can optionally include the number of items, "count" in this case, using the usual placeholder notation.

PUBLICLY SHARED

zero	**=0**{no wombats}
one	**=1**{one wombat}
two	**=2**(two wombats}
few	**few**{the {count} wombats}
3-10, fractions	**many**{{count} wombats}
other	**other**{{count} wombats}

The entire plural expression must be bracketed with curly braces and begin with the name of the int "number of items" parameter followed by ",plural". It is a little clunky in its fully general glory.

Because the message catalog is a JSON format object, the entire message, with all its variations, must appear on one line.

```
"nWombats": "{count,plural, =0{no wombats} other{{count} wombats}}",
"@nWombats": {
  "description": "A plural message",
  "placeholders": {
    "count": {
      "type": "int"
    }
  }
}
```

Using a plural method is easy enough, just pass it the item count parameter:

`nWombats(0)` *returns "no wombats"*

nWombats(5) *returns "5 wombats"*

The plural message can include other parameters. For example:

```
"nThings": "{count,plural, =0{no {thing}s} other{{count} {thing}s}}",
"@nThings": {
  "description": "A plural message with an additional parameter",
  "placeholders": {
    "count": {
      "type": "int"
    },
    "thing": {
      "example": "wombat"
    }
  }
```

PUBLICLY SHARED

```
}
```

Now nThings(0, “wombat”) and nThings(5, “wombat”) return the same strings as before.

Configuring the l10n code generator: The l10n.yaml file

The l10n.yaml file allows you to configure the l10n setup of your Flutter application, such as where all the input files are located, where all the output files should be created, and what Dart class name to give your localizations delegate. The full list of options is described in the table below:

arb-dir	The directory where the template and translated arb files are located. (defaults to "lib/l10n")
output-dir	The directory where the generated localization classes will be written. This option is only relevant if you want to generate the localizations code somewhere else in the Flutter project. You will also need to set the synthetic-package flag to false. The app must import the file specified in the 'output-localization-file' option from this directory. If unspecified, this defaults to the same directory as the input directory specified in 'arb-dir'.
template-arb-file	The template arb file that will be used as the basis for generating the Dart localization and messages files. (defaults to "app_en.arb")
output-localization-file	The filename for the output localization and localizations delegate classes. (defaults to "app_localizations.dart")
untranslated-messages-file	The location of a file that describes the localization messages have not been translated yet. Using this option will create a JSON file at the target location, in the following format: "locale": ["message_1", "message_2" ... "message_n"] If this option is not specified, a summary of the messages that have not been translated will be printed on the command line.
output-class	The Dart class name to use for the output localization and localizations delegate classes. (defaults to "AppLocalizations")
preferred-supported-locales	The list of preferred supported locales for the application. By default, the tool will generate the supported locales list in alphabetical order. Use this flag if you would like to default to a different locale. For example, pass in [en_US] if you would like your app to default to American English if a device supports it.
synthetic-package	Determines whether or not the generated output files will be generated as a synthetic package or at a specified directory in the Flutter project. This flag is set to true by default.

PUBLICLY SHARED

PUBLICLY SHARED

	When synthetic-package is set to false, it will generate the localizations files in the directory specified by arb-dir by default. If output-dir is specified, files will be generated there.
header	The header to prepend to the generated Dart localizations files. This option takes in a string. For example, pass in "/// All localized files." if you would like this string prepended to the generated Dart file. Alternatively, see the `header-file` option to pass in a text file for longer headers.
header-file	The header to prepend to the generated Dart localizations files. The value of this option is the name of the file that contains the header text which will be inserted at the top of each generated Dart file. Alternatively, see the `header` option to pass in a string for a simpler header. This file should be placed in the directory specified in 'arb-dir'.
[no-]use-deferred-loading	Whether to generate the Dart localization file with locales imported as deferred, allowing for lazy loading of each locale in Flutter web. This can reduce a web app's initial startup time by decreasing the size of the JavaScript bundle. When this flag is set to true, the messages for a particular locale are only downloaded and loaded by the Flutter app as they are needed. For projects with a lot of different locales and many localization strings, it can be a performance improvement to have deferred loading. For projects with a small number of locales, the difference is negligible, and might slow down the start up compared to bundling the localizations with the rest of the application. Note that this flag does not affect other platforms such as mobile or desktop.

PUBLICLY SHARED