Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Accessibility
Internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

Internationalizing Flutter apps

Contents

- Setting up an internationalized app: the flutter_localizations package
- Advanced locale definition
- Tracking the locale: The Locale class and the Localizations widget
- Loading and retrieving localized values
- Using the bundled LocalizationsDelegates
- Defining a class for the app's localized resources
- Specifying the app's supportedLocales parameter
- An alternative class for the app's localized resources
- Adding support for a new language
- Appendix: Using the Dart intl tools
- Appendix: Updating the iOS app bundle

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'll nee 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.

The tutorial that follows is largely written in terms of the Flutter `MaterialApp` class, since most applications are written that way. Applications written in terms of 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

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 additio `MaterialApp` properties, and include a separate package called `flutter_localizations`. As of February 2020, this package supports 77 languages.

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

```
dependencies:
  flutter:
    sdk: flutter
  flutter_localizations:
    sdk: flutter
```

Next, import the flutter_localizations library and specify `localizationsDelegates` and `supportedLocales` for `MaterialApp`:

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Accessibility
Internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

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

MaterialApp(
 localizationsDelegates: [
   // ... app-specific localization delegate[s] here
   GlobalMaterialLocalizations.delegate,
   GlobalWidgetsLocalizations.delegate,
   GlobalCupertinoLocalizations.delegate,
 ],
 supportedLocales: [
    const Locale('en'), // English
    const Locale('he'), // Hebrew
    const Locale.fromSubtags(languageCode: 'zh'), // Chinese *See Advanced Locales below*
    // ... other locales the app supports
  ],
  // ...
)
```

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 full 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 librar

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

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. Thi 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:

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

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 w likely contain 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 suppo 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 syst 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);
```

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Accessibility
Internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

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` widge 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. Instan 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);
```

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,
```

Using the bundled LocalizationsDelegates

To keep things as small and uncomplicated as possible, the flutter package includes implementations of the `MaterialLocalizations` and `WidgetsLocalizations` interfaces that only provide US English values. These implementation class are called `DefaultMaterialLocalizations` and `DefaultWidgetsLocalizations`, respectively. They're included automatically unl a different delegate of the same base type is specified with the app's `localizationsDelegates` parameter.

The `flutter_localizations` package includes multi-language implementations of the localizations interfaces called `GlobalMaterialLocalizations` and `GlobalWidgetsLocalizations`. International apps must specify localization delegates for th classes as described in Setting up an internationalized app.

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

MaterialApp(
 localizationsDelegates: [
   // ... app-specific localization delegate[s] here
   GlobalMaterialLocalizations.delegate,
   GlobalWidgetsLocalizations.delegate,
 ],
 supportedLocales: [
    const Locale('en'), // English
    const Locale('he'), // Hebrew
    const Locale('zh'), // Chinese
    // ... other locales the app supports
  ],
  // ...
)
```

The global localization delegates construct locale-specific instances of the corresponding classes. For example, `GlobalMaterialLocalizations.delegate` is a `LocalizationsDelegate` that produces an instance of `GlobalMaterialLocalizations`.

As of February 2020, the global localization classes support 77 languages.

Defining a class for the app's localized resources

Putting all of this together for an internationalized app usually starts with the class that encapsulates the app's localized values. 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 resources describes 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 suppo
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.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,
    );
  }
}
```

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

Specifying the app's supportedLocales parameter

Although Flutter's flutter_localizations library includes support for 77 languages, only English language translations are available default. It's up to the developer to decide exactly which languages to support, since it wouldn't make sense for the toolkit libraries support a different set of locales than the app does.

The `MaterialApp` `supportedLocales` parameter limits locale changes. When the user changes the locale setting on their device, t app's `Localizations` widget only follows suit if the new locale is a member of the 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.

In terms of the previous DemoApp example, the app only accepts the US English or French Canadian locales, and it substitutes US English (the first locale in the list) for anything else.

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

```
class DemoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      localeResolutionCallback: (Locale locale, Iterable<Locale> supportedLocales) {
        return locale;
      }
      // ...
    );
  }
}
```

An alternative class for the app's localized resource

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 dire in per language Maps.

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Accessibility
Internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

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

  final Locale locale;

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

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

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

In the minimal app the `DemoLocalizationsDelegate` is slightly different. Its `load` method returns a `SynchronousFuture` because asynchronous loading needs to take place.

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

  @override
  bool isSupported(Locale locale) => ['en', 'es'].contains(locale.languageCode);

  @override
  Future<DemoLocalizations> load(Locale locale) {
    return SynchronousFuture<DemoLocalizations>(DemoLocalizations(locale));
  }

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

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 mus provide about 70 translations ("localizations") for words or phrases.

See the following for an example of how to add support for the Belarusan 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 define

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

The locale-specific `GlobalMaterialLocalizations` subclass is called `BeMaterialLocalizations`, and the `LocalizationsDeleg` subclass is `_BeMaterialLocalizationsDelegate`. The value of `BeMaterialLocalizations.delegate` is an instance of the deleg 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` valu property getters in `BeMaterialLocalizations`, like this:

```
@override
String get backButtonTooltip => r'Back';

@override
String get cancelButtonLabel => r'CANCEL';

@override
String get closeButtonLabel => r'CLOSE';

// etc..
```

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

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

Get started
Samples & tutorials
Development
User interface
Data & backend
Accessibility & internationalization
Accessibility
Internationalization
Platform integration
Packages & plugins
Add Flutter to existing app
Tools & techniques
Migration notes
Testing & debugging
Performance & optimization
Deployment
Resources
Reference
Widget index
API reference
Package site

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

@override
String aboutListTileTitle(String applicationName) {
  final String text = aboutListTileTitleRaw;
  return text.replaceFirst(r'$applicationName', applicationName);
}
```

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

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

```
MaterialApp(
  localizationsDelegates: [
    GlobalWidgetsLocalizations.delegate,
    GlobalMaterialLocalizations.delegate,
    BeMaterialLocalizations.delegate,
  ],
  supportedLocales: [
    const Locale('be', 'BY')
  ],
  home: ...
)
```

Appendix: 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 o 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 us 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
   ```

   The `intl_messages.arb` file is a JSON format map with one entry for each `Intl.message()` function defined in `main.dart`. 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>.arl` 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
   ```

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

Appendix: Updating the iOS app bundle

iOS applications define key application metadata, including supported locales, in an `Info.plist` file that is built into the applicati bundle. To configure the locales supported by your app, you'll need to edit this file.

First, open your project's `ios/Runner.xcworkspace` Xcode workspace file then, in the **Project Navigator**, open the `Info.plist` file under the `Runner` project's `Runner` folder.

Next, select the **Information Property List** item, select **Add Item** from the **Editor** menu, then select **eLocalizations** from the pop-up menu.

Select and expand the newly-created `Localizations` item then, for each locale your application supports, add a new item and sel 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.

Once all supported locales have been added, save the file.

Get started

Samples & tutorials

Development

- User interface
- Data & backend
- Accessibility & internationalization
  - Accessibility
  - Internationalization
- Platform integration
- Packages & plugins
- Add Flutter to existing app
- Tools & techniques
- Migration notes

Testing & debugging

Performance & optimization

Deployment

Resources

Reference

- Widget index
- API reference
- Package site