Internationalization (i18n)

Application internationalization is a many-faceted area of development, focused on making applications available and user-friendly to a worldwide audience. This page describes Angular's internationalization (i18n) tools, which can help you make your app available in multiple languages.

See the i18n Example for a simple example of an AOT-compiled app, translated into French.

{@a angular-i18n}

Angular and i18n

Angular simplifies the following aspects of internationalization: * Displaying dates, number, percentages, and currencies in a local format. * Translating text in component templates. * Handling plural forms of words. * Handling alternative text.

This document focuses on **Angular CLI** projects, in which the Angular CLI generates most of the boilerplate necessary to write your app in multiple languages.

{@a setting-up-locale}

Setting up the locale of your app

A locale is an identifier (id) that refers to a set of user preferences that tend to be shared within a region of the world, such as country. This document refers to a locale identifier as a "locale" or "locale id".

A Unicode locale identifier is composed of a Unicode language identifier and (optionally) the character `-` followed by a locale extension. (For historical reasons the character `_` is supported as an alternative to `-`.) For example, in the locale id `fr-CA` the `fr` refers to the French language identifier, and the `CA` refers to the locale extension Canada.

Angular follows the Unicode LDML convention that uses stable identifiers (Unicode locale identifiers) based on the norm [BCP47](http://www.rfc-editor.org/rfc/bcp/bcp47.txt). It is very important that you follow this convention when you define your locale, because the Angular i18n tools use this locale id to find the correct corresponding locale data.

By default, Angular uses the locale `en-US`, which is English as spoken in the United States of America.

To set your app's locale to another value, use the CLI parameter `--locale` with the value of the locale id that you want to use:

ng serve --aot --locale fr

If you use JIT, you also need to define the `LOCALE_ID` provider in your main module:

For more information about Unicode locale identifiers, see the CLDR core spec.

For a complete list of locales supported by Angular, see the Angular repository.

The locale identifiers used by CLDR and Angular are based on BCP47. These specifications change over time; the following table maps previous identifiers to current ones at time of writing:

Locale name	Old locale id	New locale id
Indonesian	in	id
Hebrew	iw	he
Romanian Moldova	mo	ro-MD
Norwegian Bokmål	no, no-NO	nb
Serbian Latin	sh	sr-Latn
Filipino	tl	fil
Portuguese Brazil	pt-BR	pt
Chinese Simplified	zh-cn, zh-Hans-CN	zh-Hans
Chinese Traditional	zh-tw, zh-Hant-TW	zh-Hant
Chinese Traditional Hong Kong	zh-hk	zh-Hant-HK

i18n pipes

Angular pipes can help you with internationalization: the `DatePipe`, `CurrencyPipe`, `DecimalPipe` and `PercentPipe` use locale data to format data based on the `LOCALE_ID`.

By default, Angular only contains locale data for `en-US`. If you set the value of `LOCALE_ID` to another locale, you must import locale data for that new locale. The CLI imports the locale data for you when you use the parameter `--locale` with `ng serve` and `ng build`.

If you want to import locale data for other languages, you can do it manually:

The first parameter is an object containing the locale data imported from `@angular/common/locales`. By default, the imported locale data is registered with the locale id that is defined in the Angular locale data itself. If you want to register the imported locale data with another locale id, use the second parameter to specify a custom locale id. For example, Angular's locale data defines the locale id for French as "fr". You can use the second parameter to associate the imported French locale data with the custom locale id "fr-FR" instead of "fr".

The files in `@angular/common/locales` contain most of the locale data that you need, but some advanced formatting options might only be available in the extra dataset that you can import from `@angular/common/locales/extra`. An error message informs you when this is the case.

All locale data used by Angular are extracted from the Unicode Consortium's Common Locale Data Repository (CLDR).

Template translations

This document refers to a unit of translatable text as "text," a "message", or a "text message."

The i18n template translation process has four phases:

1. Mark static text messages in your component templates for translation.
2. An Angular i18n tool extracts the marked text into an industry standard translation source file.
3. A translator edits that file, translating the extracted text into the target language, and returns the file to you.
4. The Angular compiler imports the completed translation files, replaces the original messages with translated text, and generates a new version of the app in the target language.

You need to build and deploy a separate version of the app for each supported language.

{@a i18n-attribute}

Mark text with the i18n attribute

The Angular `i18n` attribute marks translatable content. Place it on every element tag whose fixed text is to be translated.

In the example below, an `<h1>` tag displays a simple English language greeting, "Hello i18n!"

To mark the greeting for translation, add the `i18n` attribute to the `<h1>` tag.

`i18n` is a custom attribute, recognized by Angular tools and compilers. After translation, the compiler removes it. It is not an Angular directive.

{@a help-translator}

Help the translator with a description and meaning

To translate a text message accurately, the translator may need additional information or context.

You can add a description of the text message as the value of the `i18n` attribute, as shown in the example below:

The translator may also need to know the meaning or intent of the text message within this particular app context.

You add context by beginning the `i18n` attribute value with the *meaning* and separating it from the *description* with the `|` character: `<meaning>|<description>`

All occurrences of a text message that have the same meaning will have the same translation. A text message that is associated with different meanings can have different translations.

The Angular extraction tool preserves both the meaning and the description in the translation source file to facilitate contextually-specific translations, but only the combination of meaning and text message are used to generate the specific id of a translation. If you have two similar text messages with different meanings, they are extracted separately. If you have two similar text messages with different descriptions (not different meanings), then they are extracted only once.

{@a custom-id}

Set a custom id for persistence and maintenance

The angular i18n extractor tool generates a file with a translation unit entry for each `i18n` attribute in a template. By default, it assigns each translation unit a unique id such as this one:

When you change the translatable text, the extractor tool generates a new id for that translation unit. You must then update the translation file with the new id.

Alternatively, you can specify a custom id in the `i18n` attribute by using the prefix `@@`. The example below defines the custom id `introductionHeader`:

When you specify a custom id, the extractor tool and compiler generate a translation unit with that custom id.

The custom id is persistent. The extractor tool does not change it when the translatable text changes.

Therefore, you do not need to update the translation. This approach makes maintenance easier.

Use a custom id with a description

You can use a custom id in combination with a description by including both in the value of the `i18n` attribute. In the example below, the `i18n` attribute value includes a description, followed by the custom `id`:

You also can add a meaning, as shown in this example:

Define unique custom ids

Be sure to define custom ids that are unique. If you use the same id for two different text messages, only the first one is extracted, and its translation is used in place of both original text messages.

In the example below the custom id `myId` is used for two different messages:

```
    <h3 i18n="@@myId">Hello</h3>
    <!-- ... -->
    <p i18n="@@myId">Good bye</p>
```

Consider this translation to French:

```
    <trans-unit id="myId" datatype="html">
      <source>Hello</source>
      <target state="new">Bonjour</target>
    </trans-unit>
```

Because the custom id is the same, both of the elements in the resulting translation contain the same text, `Bonjour`:

```
    <h3>Bonjour</h3>
    <!-- ... -->
    <p>Bonjour</p>
```

{@a no-element}

Translate text without creating an element

If there is a section of text that you would like to translate, you can wrap it in a `<span>` tag. However, if you don't want to create a new DOM element merely to facilitate translation, you can wrap the text in an `<ng-container>` element. The `<ng-container>` is transformed into an html comment:

{@a translate-attributes}

Add i18n translation attributes

You also can translate attributes. For example, assume that your template has an image with a `title` attribute:

This `title` attribute needs to be translated.

To mark an attribute for translation, add an attribute in the form of `i18n-x`, where `x` is the name of the attribute to translate. The following example shows how to mark the `title` attribute for translation by adding the `i18n-title` attribute on the `img` tag:

This technique works for any attribute of any element.

You also can assign a meaning, description, and id with the `i18n-x="<meaning>|<description>@@<id>"` syntax.

{@a plural-ICU}

Translate singular and plural

Different languages have different pluralization rules.

Suppose that you want to say that something was "updated x minutes ago". In English, depending upon the number of minutes, you could display "just now", "one minute ago", or "x minutes ago" (with x being the actual number). Other languages might express the cardinality differently.

The example below shows how to use a `plural` ICU expression to display one of those three options based on when the update occurred:

- The first parameter is the key. It is bound to the component property (`minutes`), which determines the number of minutes.
- The second parameter identifies this as a `plural` translation type.
- The third parameter defines a pluralization pattern consisting of pluralization categories and their matching values.

This syntax conforms to the ICU Message Format as specified in the CLDR pluralization rules.

Pluralization categories include (depending on the language):

- =0 (or any other number)
- zero
- one
- two
- few
- many
- other

After the pluralization category, put the default English text in braces (`{}`).

In the example above, the three options are specified according to that pluralization pattern. For talking about about zero minutes, you use `=0 {just now}` . For one minute, you use `=1 {one minute}` . Any unmatched cardinality uses `other {{{minutes}} minutes ago}` . You could choose to add patterns for two, three, or any other number if the pluralization rules were different. For the example of "minute", only these three patterns are necessary in English.

You can use interpolations and html markup inside of your translations.

{@a select-ICU}

Select among alternative text messages

If your template needs to display different text messages depending on the value of a variable, you need to translate all of those alternative text messages.

You can handle this with a `select` ICU expression. It is similar to the `plural` ICU expressions except that you choose among alternative translations based on a string value instead of a number, and you define those string values.

The following format message in the component template binds to the component's `gender` property, which outputs one of the following string values: "m", "f" or "o". The message maps those values to the appropriate translations:

{@a nesting-ICUS}

Nesting plural and select ICU expressions

You can also nest different ICU expressions together, as shown in this example:

{@a ng-xi18n}

Create a translation source file with *ng xi18n*

Use the `ng xi18n` command provided by the CLI to extract the text messages marked with `i18n` into a translation source file.

Open a terminal window at the root of the app project and enter the `ng xi18n` command:

ng xi18n

By default, the tool generates a translation file named `messages.xlf` in the XML Localization Interchange File Format (XLIFF, version 1.2).

If you don't use the CLI, you can use the `ng-xi18n` tool directly from the `@angular/compiler-cli` package, or you can manually use the CLI Webpack plugin `ExtractI18nPlugin` from the `@ngtools/webpack` package.

{@a other-formats}

Other translation formats

Angular i18n tooling supports three translation formats: * XLIFF 1.2 (default) * XLIFF 2 * XML Message Bundle (XMB)

You can specify the translation format explicitly with the `--i18nFormat` flag as illustrated in these example commands:

ng xi18n --i18nFormat=xlf ng xi18n --i18nFormat=xlf2 ng xi18n --i18nFormat=xmb

The sample in this guide uses the default XLIFF 1.2 format.

XLIFF files have the extension .xlf. The XMB format generates .xmb source files but uses .xtb (XML Translation Bundle: XTB) translation files.

{@a ng-xi18n-options}

Other options

You can specify the output path used by the CLI to extract your translation source file with the parameter `--outputPath`:

ng xi18n --outputPath src/locale

You can change the name of the translation source file that is generated by the extraction tool with the

parameter `--outFile` :

ng xi18n --outFile source.xlf

You can specify the base locale of your app with the parameter `--locale` :

ng xi18n --locale fr

The extraction tool uses the locale to add the app locale information into your translation source file. This information is not used by Angular, but external translation tools may need it.

{@a translate}

Translate text messages

The `ng xi18n` command generates a translation source file named `messages.xlf` in the project `src` folder.

The next step is to translate this source file into the specific language translation files. The example in this guide creates a French translation file.

{@a localization-folder}

Create a localization folder

Most apps are translated into more than one other language. For this reason, it is standard practice for the project structure to reflect the entire internationalization effort.

One approach is to dedicate a folder to localization and store related assets, such as internationalization files, there.

Localization and internationalization are different but closely related terms.

This guide follows that approach. It has a `locale` folder under `src/` . Assets within that folder have a filename extension that matches their associated locale.

Create the translation files

For each translation source file, there must be at least one language translation file for the resulting translation.

For this example:

1. Make a copy of the `messages.xlf` file.

2. Put the copy in the `locale` folder.
3. Rename the copy to `messages.fr.xlf` for the French language translation.

If you were translating to other languages, you would repeat these steps for each target language.

{@a translate-text-nodes}

Translate text nodes

In a large translation project, you would send the `messages.fr.xlf` file to a French translator who would enter the translations using an XLIFF file editor.

This sample file is easy to translate without a special editor or knowledge of French.

1. Open `messages.fr.xlf` and find the first `<trans-unit>` section:

> This XML element represents the translation of the `<h1>` greeting tag that you marked with the `i18n` attribute earlier in this guide.
>
> Note that the translation unit `id=introductionHeader` is derived from the custom `id` that you set earlier, but without the `@@` prefix required in the source HTML.

1. Duplicate the `<source/>` tag, rename it `target`, and then replace its content with the French greeting. If you were working with a more complex translation, you could use the the information and context provided by the source, description, and meaning elements to guide your selection of the appropriate French translation.

1. Translate the other text nodes the same way:

The Angular i18n tools generated the ids for these translation units. Don't change them. Each `id` depends upon the content of the template text and its assigned meaning. If you change either the text or the meaning, then the `id` changes. For more information, see the **[translation file maintenance discussion](#custom-id)**.

{@a translate-plural-select}

Translate plural and select expressions

Plural and *select* ICU expressions are extracted separately, so they require special attention when preparing for translation.

Look for these expressions in relation to other translation units that you recognize from elsewhere in the source template. In this example, you know the translation unit for the `select` must be just below the translation

unit for the logo.

{@a translate-plural}

Translate *plural*

To translate a `plural`, translate its ICU format match values:

You can add or remove plural cases, with each language having its own cardinality. (See CLDR plural rules.)

{@a translate-select}

Translate *select*

Below is the content of our example `select` ICU expression in the component template:

The extraction tool broke that into two translation units because ICU expressions are extracted separately.

The first unit contains the text that was outside of the `select`. In place of the `select` is a placeholder, `<x id="ICU">`, that represents the `select` message. Translate the text and move around the placeholder if necessary, but don't remove it. If you remove the placeholder, the ICU expression will not be present in your translated app.

The second translation unit, immediately below the first one, contains the `select` message. Translate that as well.

Here they are together, after translation:

{@a translate-nested}

Translate a nested expression

A nested expression is similar to the previous examples. As in the previous example, there are two translation units. The first one contains the text outside of the nested expression:

The second unit contains the complete nested expression:

And both together:

The entire template translation is complete. The next section describes how to load that translation into the app.

{@a app-pre-translation}

The app and its translation file

The sample app and its translation file are now as follows:

{@a merge}

Merge the completed translation file into the app

To merge the translated text into component templates, compile the app with the completed translation file. Provide the Angular compiler with three translation-specific pieces of information:

- The translation file.
- The translation file format.
- The locale (`fr` or `en-US` for instance).

The compilation process is the same whether the translation file is in `.xlf` format or in another format that Angular understands, such as `.xtb`.

How you provide this information depends upon whether you compile with the JIT compiler or the AOT compiler.

- With AOT, you pass the information as a CLI parameter.
- With JIT, you provide the information at bootstrap time.

{@a merge-aot}

Merge with the AOT compiler

The AOT (*Ahead-of-Time*) compiler is part of a build process that produces a small, fast, ready-to-run application package.

When you internationalize with the AOT compiler, you must pre-build a separate application package for each language and serve the appropriate package based on either server-side language detection or url parameters.

You also need to instruct the AOT compiler to use your translation file. To do so, you use three options with the `ng serve` or `ng build` commands:

- `--i18nFile` : the path to the translation file.
- `--i18nFormat` : the format of the translation file.
- `--locale` : the locale id.

The example below shows how to serve the French language file created in previous sections of this guide:

ng serve --aot --i18nFile=src/locale/messages.fr.xlf --i18nFormat=xlf --locale=fr

{@a merge-jit}

Merge with the JIT compiler

The JIT compiler compiles the app in the browser as the app loads. Translation with the JIT compiler is a dynamic process of:

1. Importing the appropriate language translation file as a string constant.
2. Creating corresponding translation providers for the JIT compiler.
3. Bootstrapping the app with those providers.

Three providers tell the JIT compiler how to translate the template texts for a particular language while compiling the app:

- `TRANSLATIONS` is a string containing the content of the translation file.
- `TRANSLATIONS_FORMAT` is the format of the file: `xlf`, `xlf2`, or `xtb`.
- `LOCALE_ID` is the locale of the target language.

The Angular `bootstrapModule` method has a second `compilerOptions` parameter that can influence the behavior of the compiler. You can use it to provide the translation providers:

Then provide the `LOCALE_ID` in the main module:

{@a missing-translation}

Report missing translations

By default, when a translation is missing, the build succeeds but generates a warning such as `Missing translation for message "foo"`. You can configure the level of warning that is generated by the Angular compiler:

- Error: throw an error. If you are using AOT compilation, the build will fail. If you are using JIT compilation, the app will fail to load.
- Warning (default): show a 'Missing translation' warning in the console or shell.
- Ignore: do nothing.

If you use the AOT compiler, specify the warning level by using the CLI parameter `--missingTranslation`. The example below shows how to set the warning level to error:

ng serve --aot --missingTranslation=error

If you use the JIT compiler, specify the warning level in the compiler config at bootstrap by adding the 'MissingTranslationStrategy' property. The example below shows how to set the warning level to error: