Localizing your app

Contents ›

Internationalization (i18n) is the process of designing and preparing your app to be usable in different locales around the world. *Localization* is the process of building versions of your app for different locales, including extracting text for translation into different languages, and formatting data for particular locales.

A *locale* identifies a region (such as a country) in which people speak a particular language or language variant. The locale determines the formatting and parsing of dates, times, numbers, and currencies as well as measurement units and the translated names for time zones, languages, and countries.

> Create an adaptable user interface for all of your target locales that takes into consideration the differences in spacing for different languages. For details, see How to approach internationalization.

Use Angular to internationalize your app:

- Use built-in pipes to display dates, numbers, percentages, and currencies in a local format.
- Mark text in component templates for translation.
- Mark plural forms of expressions for translation.
- Mark alternate text for translation.

After preparing your app for an international audience, use the Angular CLI to localize your app by performing the following tasks:

- Use the CLI to extract marked text to a *source language* file.
- Make a copy of this file for each language, and send these *translation files* to a translator or service.
- Use the CLI to merge the finished translation files when building your app for one or more locales.

> To explore the sample app with French translations used in this guide, see the live example / download example.

Prerequisites

To prepare your app for translations, you should have a basic understanding of the following:

- Templates
- Components
- Angular CLI command-line tool for managing the Angular development cycle
- Extensible Markup Language (XML) used for translation files

Steps to localize your app

To localize your app, follow these general steps:

1. Add the localize package.
2. Refer to locales by ID.
3. Format data based on locale.
4. Prepare templates for translations.
5. Work with translation files.
6. Merge translations into the app.
7. Deploy multiple locales.

While following these steps, you can explore the translated example app.

The following are optional practices that may be required in special cases:

- Set the source locale manually if you need to set the LOCALE_ID token.
- Import global variants of the locale data for extra locale data.
- Manage marked text with custom IDs if you require more control over matching translations.

Add the localize package

To take advantage of Angular's localization features, use the Angular CLI to add the `@angular/localize` package to your project:

```
ng add @angular/localize
```

This command updates your project's `package.json` and `polyfills.ts` files to import the `@angular/localize` package.

> For more information about `package.json` and polyfill packages, see Workspace npm dependencies.

If `@angular/localize` is not installed, the Angular CLI may generate an error when you try to build a localized version of your app.

Refer to locales by ID

Refer to a locale using the Unicode *locale identifier* (ID), which specifies the language, country, and an optional code for further variants or subdivisions.

UNICODE LOCALE IDENTIFIERS

- For a list of language codes, see ISO 639-2.
- IDs conform to the Unicode Common Locale Data Repository (CLDR). For more information about Unicode locale identifiers, see the CLDR core specification.
- CLDR and Angular base their identifiers on BCP47 tags.

The ID consists of a language identifier, such as `en` for English or `fr` for French, followed by a dash (`-`) and a locale extension, such as `US` for the United States or `CA` for Canada. For example, `en-US` refers to English in the United States, and `fr-CA` refers to French in Canada. Angular uses this ID to find the correct corresponding locale data.

> Many countries, such as France and Canada, use the same language (French, identified as `fr`) but differ in grammar, punctuation, and formats for currency, decimal numbers, and dates. Use a more specific locale ID, such as French for Canada (`fr-CA`), when localizing your app.

Angular by default uses `en-US` (English in the United States) as your app's source locale.

The Angular repository includes common locales. You can change your app's source locale for the build by setting the source locale in the `sourceLocale` field of your app's workspace configuration file (`angular.json`). The build process (described in Merge translations into the app in this guide) uses your app's `angular.json` file to automatically set the `LOCALE_ID` token and load the locale data.

Format data based on locale

Angular provides the following built-in data transformation pipes that use the `LOCALE_ID` token to format data according to the locale's rules:

- `DatePipe`: Formats a date value.
- `CurrencyPipe`: Transforms a number to a currency string.
- `DecimalPipe`: Transforms a number into a decimal number string.
- `PercentPipe`: Transforms a number to a percentage string.

For example, `{{today | date}}` uses `DatePipe` to display the current date in the format for the locale in `LOCALE_ID`.

To override the value of `LOCALE_ID`, add the `locale` parameter. For example, to force the currency to use `en-US` no matter which language-locale you set for `LOCALE_ID`, use this form: `{{amount | currency : 'en-US'}}`.

Prepare templates for translations

To translate your app's templates, you need to prepare the text for a translator or translation service by marking text, attributes, and other elements with the Angular `i18n` attribute. Follow these general steps:

1. Mark text for translations.
2. Add helpful descriptions and meanings to help the translator with additional information or context.
3. Translate text not for display.
4. Mark element attributes for translations, such as an image's `title` attribute.
5. Mark plurals and alternates for translation in order to comply with the pluralization rules and grammatical constructions of different languages.

Mark text for translations

Mark the static text messages in your component templates for translation using the `i18n` attribute. Place it on every element tag with fixed text to be translated.

For example, the following `<h1>` tag displays a simple English language greeting, "Hello i18n!"

src/app/app.component.html

```
<h1>Hello i18n!</h1>
```

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

src/app/app.component.html

```
<h1 i18n>Hello i18n!</h1>
```

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

Add helpful descriptions and meanings

To translate a text message accurately, the translator may need additional information or context. Add a *description* of the text message as the value of the `i18n` attribute, as shown in the following example:

src/app/app.component.html

```
<h1 i18n="An introduction header for this sample">Hello i18n!</h1>
```

The translator may also need to know the meaning or intent of the text message within this particular app context, in order to translate it the same way as other text with the same meaning. Start the `i18n` attribute value with the *meaning* and separate it from the *description* with the `|` character: `<meaning>|<description>`.

For example, you can add the meaning that this `<h1>` tag is a site header that needs to be translated the same way not only when used as a header, but also when referred to from another section of text:

src/app/app.component.html

```
<h1 i18n="site header|An introduction header for this sample">Hello i18n!</h1>
```

As a result, any text marked with `site header` as the *meaning* is translated exactly the same way.

HOW MEANINGS CONTROL TEXT EXTRACTION AND MERGING

The Angular extraction tool (described in Work with translation files in this guide) generates a translation unit entry for each `i18n` attribute in a template. It assigns each translation unit a unique ID based on the *meaning* and *description*.

The same text elements with different *meanings* are extracted with separate IDs. For example, if the word "right" appears with the meaning `correct` (as in "You are right") in one place, and with the meaning `direction` (as in "Turn right") in another place, the word is translated differently and merged back into the app as different translation entries.

If the same text elements have different *descriptions* but the same *meaning*, they are extracted only once, with only one ID. That one translation entry is merged back into the app wherever the same text elements appear.

Translate text not for display

While you can translate non-displayed text using a `<span>` tag, you are creating a new DOM element. To avoid doing so, wrap the text in an `<ng-container>` element, which is transformed into a non-displayed HTML comment as shown in this example:

```
<ng-container i18n>I don't output any element</ng-container>
```

Mark element attributes for translations

HTML attributes such as `title` include text that should be translated along with the rest of the displayed text in the template. The following example shows an image with a `title` attribute:

src/app/app.component.html

```
<img [src]="logo" title="Angular logo">
```

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

src/app/app.component.html

```
<img [src]="logo" i18n-title title="Angular logo" />
```

You can use `i18n-`*attribute* with any attribute of any element. You also can assign a meaning, description, and custom ID with the `i18n-`*attribute*`="<meaning>|<description>@@<id>"` syntax.

Mark plurals and alternates for translation

Different languages have different pluralization rules and grammatical constructions that can make translation difficult. To simplify translation, use International Components for Unicode (ICU) clauses with regular expressions, such as `plural` to mark the uses of plural numbers, and `select` to mark alternate text choices.

> The ICU clauses adhere to the ICU Message Format specified in the CLDR pluralization rules.

Mark plurals

Use the `plural` clause to mark expressions that may not be meaningful if translated word-for-word.

For example, if you want to display "updated x minutes ago" in English, you may want to display "just now", "one minute ago", or "*x* minutes ago" (with *x* as the actual number). Other languages might express this cardinality differently. The following example shows how to use a `plural` clause to express these three options:

src/app/app.component.html

```
<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other
{{{minutes}} minutes ago}}</span>
```

In the above example:

- The first parameter, `minutes`, 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 pattern of pluralization categories and their matching values:
  - For zero minutes, use `=0 {just now}`.
  - For one minute, use `=1 {one minute}`.
  - For any unmatched cardinality, use `other {{{minutes}} minutes ago}`. You can use HTML markup and interpolations such as `{{{minutes}}` with the `plural` clause in expressions.
  - After the pluralization category, put the default text (English) within braces (`{}`).

Pluralization categories include (depending on the language):

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

LOCALES MAY NOT SUPPORT SOME PLURALIZATION CATEGORIES

Many locales don't support some of the pluralization categories. For example, the default locale (`en-US`) and other locales (such as `es`) have very simple `plural()` functions that don't support the `few` category. The following shows the en-US `plural()` function:

```
function plural(n: number): number {
  let i = Math.floor(Math.abs(n)), v = n.toString().replace(/^[^.]*\.?/,
'').length;
  if (i === 1 && v === 0) return 1;
  return 5;
}
```

The function will only ever return 1 (`one`) or 5 (`other`). The `few` category will never match. If none of the pluralization categories match, Angular will try to match `other`. Use `other` as the standard fallback for a missing category.

For more information about pluralization categories, see Choosing plural category names in the CLDR - Unicode Common Locale Data Repository.

Mark alternates and nested expressions

If you need to display alternate text depending on the value of a variable, you need to translate all of the alternates.

The `select` clause, similar to the `plural` clause, marks choices for alternate text based on your defined string values. For example, the following clause in the component template binds to the component's `gender` property, which outputs one of the following string values: "male", "female" or "other". The clause maps those values to the appropriate translations:

src/app/app.component.html

```
<span i18n>The author is {gender, select, male {male} female {female} other {other}}
</span>
```

You can also nest different clauses together, such as the `plural` and `select` clauses in the following example:

src/app/app.component.html

```
<span i18n>Updated: {minutes, plural,
  =0 {just now}
  =1 {one minute ago}
  other {{{minutes}} minutes ago by {gender, select, male {male} female {female} other
{other}}}}
</span>
```

Work with translation files

After preparing a template for translation, use the Angular CLI `xi18n` command to extract the marked text in the template into a *source language* file. The marked text includes text marked with `i18n` and attributes marked with `i18n-`*attribute* as described in the previous section. Follow these steps:

1. Extract the source language file. You can optionally change the location, format, and name.
2. Create a translation file for each language by copying the source language file.
3. Translate each translation file.
4. Translate plurals and alternate expressions separately.

Extract the source language file

To extract the source language file, open a terminal window, change to the root directory of your app project, and run the following CLI command:

```
ng xi18n
```

The `xi18n` command creates a source language file named `messages.xlf` in your project's root directory using the XML Localization Interchange File Format (XLIFF, version 1.2).

Use the following `xi18n` command options to change the source language file location, format, and file name:

- `--output-path`: Change the location.
- `--format`: Change the format.
- `--outFile`: Change the file name.

Note: The `--i18n-locale` option is deprecated. Angular 9 uses the source locale configured in your app's workspace configuration file (`angular.json`).

The ng xi18n extraction mechanism uses the ViewEngine compiler, which is not able to cope with some code syntax that the Ivy compiler can handle. A fix for this issue is currently under development. To follow or contribute to this fix see PR 32912.

Change the source language file location

To create a file in the `src/locale` directory, specify the output path as an option, as shown in the following example:

```
ng xi18n --output-path src/locale
```

Change the source language file format

The `xi18n` command can read and write files in three translation formats:

- XLIFF 1.2 (default)
- XLIFF 2
- XML Message Bundle (XMB)

Specify the translation format explicitly with the `--format` command option, as shown in the following examples:

```
ng xi18n  --format=xlf
ng xi18n  --format=xlf2
ng xi18n  --format=xmb
```

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

Change the source language file name

To change the name of the source language file generated by the extraction tool, use the `--outFile` command option:

```
ng xi18n --out-file source.xlf
```

Create a translation file for each language

The `ng xi18n` command (with no options) generates a source language file named `messages.xlf` in the project `src` folder. Create *translation* files for each language by copying the source language file. To avoid confusion with multiple translations, you should organize the language translation files by locale in a dedicated `locale` folder under `src/`. Use a filename extension that matches the associated locale, such as `messages.fr.xlf`.

For example, to create a French translation file, follow these steps:

1. Make a copy of the `messages.xlf` source language file.
2. Put the copy in the `src/locale` folder.
3. Rename the copy to `messages.fr.xlf` for the French language (`fr`) translation. Send this translation file to the translator.

Repeat the above steps for each language you want to add to your app.

Translate each translation file

Unless you are fluent in the language and have the time to edit translations, you would likely send each translation file to a translator, who would then use an XLIFF file editor to create and edit the translation.

To demonstrate this process, see the `messages.fr.xlf` file in the live example / download example, which includes a French translation you can edit without a special XLIFF editor or knowledge of French. Follow these steps:

1. Open `messages.fr.xlf` and find the first `<trans-unit>` element. This is a *translation unit*, also known as a *text node*, representing the translation of the `<h1>` greeting tag that was previously marked with the `i18n` attribute:

   src/locale/messages.fr.xlf (<trans-unit>)

   ```
   <trans-unit id="introductionHeader" datatype="html">
     <source>Hello i18n!</source>
     <note priority="1" from="description">An introduction header for this
   sample</note>
     <note priority="1" from="meaning">User welcome</note>
   </trans-unit>
   ```

   The `id="introductionHeader""` is a custom ID, but without the `@@` prefix required in the source HTML.

2. Duplicate the `<source>...</source>` element in the text node, rename it `target`, and then replace its content with the French text:

   src/locale/messages.fr.xlf (<trans-unit>, after translation)

   ```
   <trans-unit id="introductionHeader" datatype="html">
     <source>Hello i18n!</source>
     <target>Bonjour i18n !</target>
     <note priority="1" from="description">An introduction header for this
   sample</note>
     <note priority="1" from="meaning">User welcome</note>
   </trans-unit>
   ```

In a more complex translation, the information and context in the description and meaning elements described previously would help you choose the right words for translation.

3. Translate the other text nodes the same way as shown in the following example:

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="ba0cc104d3d69bf669f97b8d96a4c5d8d9559aa3" datatype="html">
  <source>I don&apos;t output any element</source>
  <target>Je n'affiche aucun élément</target>
</trans-unit>
<trans-unit id="701174153757adf13e7c24a248c8a873ac9f5193" datatype="html">
  <source>Angular logo</source>
  <target>Logo d'Angular</target>
</trans-unit>
```

Don't change the IDs for translation units. Each `id` is generated by Angular and depends on 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 about managing text updates and IDs, see the previous section on custom IDs.

Translate plurals and alternate expressions

The `plural` and `select` ICU expressions are extracted as additional messages, so you must translate them separately.

Translate plurals

To translate a `plural`, translate its ICU format match values as shown in the following example:

- `just now`
- `one minute ago`
- `<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago`

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="5a134dee893586d02bffc9611056b9cadf9abfad" datatype="html">
  <source>{VAR_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago} }</source>
  <target>{VAR_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes} }</target>
</trans-unit>
```

You can add or remove plural cases as needed for each language.

For language plural rules, see CLDR plural rules.

Translate alternate expressions

Angular also extracts alternate `select` ICU expressions as separate translation units. The following shows a `select` ICU expression in the component template:

src/app/app.component.html

```
<span i18n>The author is {gender, select, male {male} female {female} other {other}}
</span>
```

In this example, Angular extracts the expression into two translation units. The first contains the text outside of the `select` clause, and uses a placeholder for `select` (`<x id="ICU">`):

src/locale/messages.fr.xlf (<trans-unit>)

```
</trans-unit>
<trans-unit id="f99f34ac9bd4606345071bd813858dec29f3b7d1" datatype="html">
  <source>The author is <x id="ICU" equiv-text="{gender, select, male {...} female
{...} other {...}}"/></source>
  <target>L'auteur est <x id="ICU" equiv-text="{gender, select, male {...} female {...}
other {...}}"/></target>
</trans-unit>
```

When translating the text, you can move the placeholder if necessary, but don't remove it. If you remove the placeholder, the ICU expression will not appear in your translated app.

The second translation unit contains the `select` clause:

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="eff74b75ab7364b6fa888f1cbfae901aaaf02295" datatype="html">
  <source>{VAR_SELECT, select, male {male} female {female} other {other} }</source>
  <target>{VAR_SELECT, select, male {un homme} female {une femme} other {autre} }
</target>
</trans-unit>
```

The following example shows both translation units after translating:

src/locale/messages.fr.xlf (<trans-unit>)

```
</trans-unit>
```

```
<trans-unit id="f99f34ac9bd4606345071bd813858dec29f3b7d1" datatype="html">
  <source>The author is <x id="ICU" equiv-text="{gender, select, male {...} female
{...} other {...}}"/></source>
  <target>L'auteur est <x id="ICU" equiv-text="{gender, select, male {...} female {...}
other {...}}"/></target>
</trans-unit>
<trans-unit id="eff74b75ab7364b6fa888f1cbfae901aaaf02295" datatype="html">
  <source>{VAR_SELECT, select, male {male} female {female} other {other} }</source>
  <target>{VAR_SELECT, select, male {un homme} female {une femme} other {autre} }
</target>
</trans-unit>
```

Translate nested expressions

Angular treats a nested expression in the same manner as an alternate expression, extracting it into two translation units. The first contains the text outside of the nested expression:

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="972cb0cf3e442f7b1c00d7dab168ac08d6bdf20c" datatype="html">
  <source>Updated: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other
{...}}"/></source>
  <target>Mis à jour: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other
{...}}"/></target>
</trans-unit>
```

The second translation unit contains the complete nested expression:

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="7151c2e67748b726f0864fc443861d45df21d706" datatype="html">
  <source>{VAR_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago by {VAR_SELECT, select, male
{male} female {female} other {other} }} }</source>
  <target>{VAR_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes par {VAR_SELECT, select, male {un
homme} female {une femme} other {autre} }} }</target>
</trans-unit>
```

The following example shows both translation units after translating:

src/locale/messages.fr.xlf (<trans-unit>)

```
<trans-unit id="972cb0cf3e442f7b1c00d7dab168ac08d6bdf20c" datatype="html">
  <source>Updated: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other
{...}}"/></source>
```

```
  <target>Mis à jour: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other
{...}}"/></target>
</trans-unit>
<trans-unit id="7151c2e67748b726f0864fc443861d45df21d706" datatype="html">
  <source>{VAR_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago by {VAR_SELECT, select, male
{male} female {female} other {other} }} }</source>
  <target>{VAR_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x
id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes par {VAR_SELECT, select, male {un
homme} female {une femme} other {autre} }} }</target>
</trans-unit>
```

Merge translations into the app

To merge the completed translations into the app, use the Angular CLI to build a copy of the app's distributable files for each locale. The build process replaces the original text with translated text, and sets the `LOCALE_ID` token for each distributable copy of the app. It also loads and registers the locale data.

After merging, you can serve each distributable copy of the app using server-side language detection or different subdirectories, as described in the next section about deploying multiple locales.

The build process uses ahead-of-time (AOT) compilation to produce a small, fast, ready-to-run app. With Ivy in Angular version 9, AOT is used by default for both development and production builds, and AOT is required to localize component templates.

> For a detailed explanation the build process, see Building and serving Angular apps. This build process works for translation files in the `.xlf` format or in another format that Angular understands, such as `.xtb`.

> Ivy does not support merging i18n translations when using JIT mode. If you disable Ivy and are using JIT mode, see merging with the JIT compiler.

To build a separate distributable copy of the app for each locale, define the locales in the build configuration in your project's workspace configuration file `angular.json`. This method shortens the build process by removing the requirement to perform a full app build for each locale.

You can then generate app versions for each locale using the `"localize"` option in `angular.json`. You can also build from the command line using the Angular CLI `build` command with the `--localize` option.

> You can optionally apply specific build options for just one locale for a custom locale configuration.

Define locales in the build configuration

Use the `i18n` project option in your app's build configuration file (`angular.json`) to define locales for a project. The following sub-options identify the source language and tell the compiler where to find supported translations for the project:

- `sourceLocale`: The locale you use within the app source code (`en-US` by default)
- `locales`: A map of locale identifiers to translation files

For example, the following excerpt of an `angular.json` file sets the source locale to `en-US` and provides the path to the `fr` (French) locale translation file:

angular.json

```
...
"projects": {
  "angular.io-example": {
    ...
    "i18n": {
      "sourceLocale": "en-US",
      "locales": {
        "fr": "src/locale/messages.fr.xlf"
      }
    },
    "architect": {
    ...
    }
}
```

Generate app versions for each locale

To use your locale definition in the build configuration, use the `"localize"` option in `angular.json` to tell the CLI which locales to generate for the build configuration:

- Set `"localize"` to `true` for *all* the locales previously defined in the build configuration.
- Set `"localize"` to an array of a subset of the previously-defined locale identifiers to build only those locale versions.
- Set `"localize"` to `false` to disable localization and not generate any locale-specific versions.

> Note: Ahead-of-time (AOT) compilation is required to localize component templates. If you changed this setting, set `"aot"` to `true` in order to use AOT.

The following example shows the `"localize"` option set to `true` in `angular.json` so that all locales defined in the build configuration are built:

angular.json

```
"build": {
  "builder": "@angular-devkit/build-angular:browser",
  "options": {
    "localize": true,
    "aot": true,
    ...
```

> Due to the deployment complexities of i18n and the need to minimize rebuild time, the development server only supports localizing a single locale at a time. Setting the `"localize"` option to `true` will cause an error when using `ng serve` if more than one locale is defined. Setting the option to a specific locale, such as `"localize": ["fr"]`, can work if you want to develop against a specific locale (such as `fr`).

The CLI loads and registers the locale data, places each generated version in a locale-specific directory to keep it separate from other locale versions, and puts the directories within the configured `outputPath` for the project. For each application variant the `lang` attribute of the `html` element is set to the locale. The CLI also adjusts the HTML base HREF for each version of the app by adding the locale to the configured `baseHref`.

You can set the `"localize"` property as a shared configuration that all the configurations effectively inherit (or can override).

Build from the command line

You can also use the `--localize` option with the `ng build` command and your existing `production` configuration. The CLI builds all locales defined in the build configuration, which is similar to setting the `"localize"` option to `true` as described in the previous section.

```
ng build --prod --localize
```

Apply specific build options for just one locale

To apply specific build options to only one locale, you can create a custom locale-specific configuration by specifying a single locale as shown in the following example:

angular.json

```
"build": {
  ...
  "configurations": {
    ...
    "fr": {
      "localize": ["fr"],
      "main": "src/main.fr.ts",
```

```
      ...
    }
  }
},
"serve": {
  ...
  "configurations": {
    ...
    "fr": {
      "browserTarget": "*project-name*:build:fr"
    }
  }
}
```

You can then pass this configuration to the `ng serve` or `ng build` commands. The following shows how to serve the French language file created in the example for this guide:

```
ng serve --configuration=fr
```

> You can use the CLI development server (`ng serve`), but only with a single locale.

For production builds, you can use configuration composition to execute both configurations:

```
ng build --configuration=production,fr
```

angular.json

```
...
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": { ... },
    "configurations": {
      "fr": {
        "localize": ["fr"],
      }
    }
  },
  ...
  "serve": {
    "builder": "@angular-devkit/build-angular:dev-server",
```

```
    "options": {
      "browserTarget": "my-project:build"
    },
    "configurations": {
      "production": {
        "browserTarget": "my-project:build:production"
      },
      "fr": {
        "browserTarget": "my-project:build:fr"
      }
    }
  }
}
```

Report missing translations

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.

Specify the warning level in the `options` section for the `build` target of your Angular CLI configuration file (`angular.json`). The following example shows how to set the warning level to `error`:

angular.json

```
"options": {
  ...
  "i18nMissingTranslation": "error"
}
```

Deploy multiple locales

If `myapp` is the directory containing your app's distributable files, you would typically make available different versions for different locales in locale directories such as `myapp/fr` for the French version and `myapp.com/es` for the Spanish version.

The HTML `base` tag with the `href` attribute specifies the base URI, or URL, for relative links. If you set the `"localize"` option in `angular.json` to `true` or to an array of locale IDs, the CLI adjusts the base `href` for each version of the app by adding the locale to the configured `"baseHref"`. You can specify the `"baseHref"` for each locale in your workspace configuration file (`angular.json`), as shown in the following example, which sets `"baseHref"` to an empty string:

angular.json

```
...
"projects": {
  "angular.io-example": {
    ...
    "i18n": {
      "sourceLocale": "en-US",
      "locales": {
        "fr": "src/locale/messages.fr.xlf"
        "baseHref": ""
      }
    },
    "architect": {
    ...
    }
}
```

You can also use the CLI `--baseHref` option with `ng build` to declare the base `href` at compile time.

Configuring servers for hosting multiple locales is outside the scope of this guide. For details on how to deploy apps to a remote server, see Deployment.

Explore the translated example app

The following tabs show the example app and its translation files:

src/app/app.component.html | src/app/app.component.ts | src/app/app.module.ts | s

```
<h1 i18n="User welcome|An introduction header for this sample@@introductionHeader">
  Hello i18n!
</h1>

<ng-container i18n>I don't output any element</ng-container>

<br />

<img [src]="logo" i18n-title title="Angular logo" />
<br>
<button (click)="inc(1)">+</button> <button (click)="inc(-1)">-</button>
<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other
{{{minutes}} minutes ago}}</span>
({{minutes}})
<br><br>
<button (click)="male()">&#9794;</button> <button (click)="female()">&#9792;</button>
<button (click)="other()">&#9895;</button>
<span i18n>The author is {gender, select, male {male} female {female} other {other}}
</span>
<br><br>
```

```
<span i18n>Updated: {minutes, plural,
  =0 {just now}
  =1 {one minute ago}
  other {{{minutes}} minutes ago by {gender, select, male {male} female {female} other
{other}}}}
</span>
```

Optional practices

The following are optional practices that may be required in special cases:

- Set the source locale manually by setting the LOCALE_ID token.
- Import global variants of the locale data for extra locale data.
- Manage marked text with custom IDs if you require more control over matching translations.

Set the source locale manually

Angular already contains locale data for `en-US`. The Angular CLI automatically includes the locale data and sets the `LOCALE_ID` value when you use the `--localize` option with `ng build`.

To manually set an app's source locale to one other than the automatic value, follow these steps:

1. Look up the ID for the language-locale combination in the Angular repository.
2. Set the `LOCALE_ID` token. The following example sets the value of `LOCALE_ID` to `fr` (French):

src/app/app.module.ts

```
import { LOCALE_ID, NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from '../src/app/app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  providers: [ { provide: LOCALE_ID, useValue: 'fr' } ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Import global variants of the locale data

Angular will automatically include locale data if you configure the locale using the `--localize` option with `ng build` CLI command.

The Angular repository files (`@angular/common/locales`) contain most of the locale data that you need, but some advanced formatting options require additional locale data. Global variants of the locale data are available in

`@angular/common/locales/global`. The following example imports the global variants for French (`fr`):

app.module.ts

```
import '@angular/common/locales/global/fr';
```

Manage marked text with custom IDs

The Angular extractor generates a file with a translation unit entry for each `i18n` attribute in a template. As described previously (in How meanings control text extraction and merging), Angular assigns each translation unit a unique ID such as the following:

messages.fr.xlf.html

```
<trans-unit id="ba0cc104d3d69bf669f97b8d96a4c5d8d9559aa3" datatype="html">
```

When you change the translatable text, the extractor generates a new ID for that translation unit. In most cases a text change would also require a change to the translation. Therefore, using a new ID keeps the text change in sync with translations.

However, some translation systems require a specific form or syntax for the ID. To address this requirement, you can mark text with *custom* IDs. While most developers don't need to use custom IDs, some may want to use IDs that have a unique syntax to convey additional metadata (such as the library, component, or area of the app in which the text appears).

Specify a custom ID in the `i18n` attribute by using the prefix `@@`. The following example defines the custom ID `introductionHeader`:

app/app.component.html

```
<h1 i18n="@@introductionHeader">Hello i18n!</h1>
```

When you specify a custom ID, the extractor generates a translation unit with the custom ID:

messages.fr.xlf.html

```
<trans-unit id="introductionHeader" datatype="html">
```

If you change the text, the extractor does *not* change the ID. As a result, you don't have to take the extra step of updating the translation. The drawback of using custom IDs is that if you change the text, your translation may be out-of-sync with the newly changed source text.

Use a custom ID with a description

Use a custom ID in combination with a description and a meaning to further help the translator. The following example includes a description, followed by the custom `id`:

app/app.component.html

```
<h1 i18n="An introduction header for this sample@@introductionHeader">Hello i18n!</h1>
```

The following example adds a meaning:

app/app.component.html

```
<h1 i18n="site header|An introduction header for this sample@@introductionHeader">Hello i18n!</h1>
```

Define unique custom IDs

Be sure to define custom IDs that are unique. If you use the same ID for two different text elements, the extraction tool extracts only the first one, and Angular uses its translation in place of both original text elements.

For example, in the following code the same custom ID `myId` is defined for two different text elements:

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

The following shows the translation to French:

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

Both elements now use the same translation (`Bonjour`) because they were defined with the same custom ID:

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

Super-powered by Google ©2010-2020. Code licensed under an MIT-style License. Documentation licensed under CC BY 4.0.

Version 10.0.10-local+sha.84d1ba792b.