Use and extract i18n from the source code

author: Olivier Combe
last update: 2017-03-16
status: draft (public)
doc url: https://goo.gl/kzQQ1H feedback url: https://goo.gl/jQ6tQf
design doc template version: 1.2.0
*This document is published in a shared folder on the angular.io domain, and possibly **publicly accessible**.*

Warning: this document is currently being revised and will change extensively. Don't take anything here as granted.

Objective

This document explores the possibilities of using/extracting i18n from the source code. Using i18n in the source code should be as simple and as unobtrusive as possible if we want the developers to use our solution instead of an alternative.

Approvals

We are seeking for approval from the following people:

- angular: berchet@, iminar@, alexeagle@ (for the extraction part), vikram@ (for g3)

Background

At the moment i18n is only supported in the templates and not in the source code.

Translating an app is a two step process:
- extraction: the templates are parsed to extract the keys (the text from the source language, usually english) to generate a messages bundle file, either xmb or xliff.
- merge: the translated strings are merged to replace the original strings. This process happens during the template compilation: runtime for JIT and build time for AOT

The original issue https://github.com/angular/angular/issues/11405 is a feature request for a method to use the translated text in your code (not just in your templates). There are many use cases for this, such as passing the translations to external libs, translating (error) messages from services, ...

Prior Art

There is no library that lets you translate strings with no overhead at runtime in AOT, but there are many JS libs (Angular or not) that let you use translations in your source code. They all use the same concept: a function that takes a string and some parameters and returns the translation. The parameters vary depending on what the library supports:

angular translate (AngularJS):
`$translate.instant(key, parameters, id, forceLanguage, sanitizeStrategy)`

ngx-translate (Angular):
`translate.instant(key, parameters)`

angular l10n (Angular):
`translate(key, parameters, lang)`

i18next (Vanilla JS):
`i18next.t(key, parameters);`

Polyglot (Vanilla JS):
`polyglot.t(key, parameters);`

Features supported by each library:

Library	angular translate	ngx translate	angular l10n	i18next	polyglot
Meaning & description					
AOT (if Angular lib)		works but not supported [3]	works but not supported [3]		
Interpolations	✔	✔	✔	✔	✔
Lang[1]	✔		✔	✔	
Fallback lang	✔	✔		✔	
HTML	✔	✔	✔		
Compiled HTML[2]	✔				
ICU	✔	external module	Not in code	✔	✔
Extraction	✔	✔			✔
Formats	JSON, or PO (plugin)	JSON, PO or any (plugins)	JSON	JSON	JSON

[1] choose the lang specifically for this translation
[2] use framework-specific code (directive, components, …) inside of your html
[3] works with AOT but still translates at runtime

To extract the translations from the source code, most existing solutions either use regular expressions, or a parser to generate an AST and a visitor to navigate through the tree.

Detailed Design

Based on existing solutions, the proposal is that developers import a service in their code and use it to return the translations wherever they need.
It will not require any additional parameters for the cli when extracting/merging the translations nor require any additional provider at bootstrap for JIT.

I18n service:

To use i18n in our code, we will need a new service:

```
@Component()
class MyComp {
  constructor(private i18n: I18nService) {
    monday = this.i18n.__('Text', {desc?, meaning?, id?});
  }
}
```

Its signature is the following:

```
function __(source: string, parameters?: {description?: string,
meaning?: string, id?: string}): string
```

In JIT the service will use the source and the parameters to generate an id that it will match with the corresponding message from the TranslationBundle containing the translations.
In AOT the service calls will be replaced by static translations at build time.

Example of code for the service:

```
__(source: string, parameters?: I18nParameters): string {
  return this.getTranslation(source, parameters);
}

getTranslation(source: string, parameters?: I18nParameters) {
  let id;
  if(!parameters || !parameters.id) {
    id = this.generateId(source);
  } else {
    id = parameters.id;
  }
  return this.translationsBundle[id] || source;
}
```

Extraction:

For the extraction we will use tsc and a visitor to parse each source file and find all calls to our service in order to retrieve i18n strings. We will add those messages to the ones extracted from the templates by the compiler.

The differents steps are:
[existing steps, in compiler-cli]

- *the developer starts an extraction with ng-xi18n or with ngtools (ex: cli)*
- *ng-xi18n or ngtools runs tsc with the source files as a parameter and then calls extract from the i18n extractor with the AST as a parameter*

[new steps, in compiler-cli]

- we use a visitor to visit each node of the AST
- we search for imports to find our service, if it is imported we store the name (if imported “as”), if not we skip this file
- for each file that imports our service we search for class constructors and we store the local name of our injected service (ex: private **i18n**: I18nService)
- for all the nodes in the current class, we search for references of this local service
- for each call to this service we extract the parameters and use them to generate a new Message
- we add this new Message to a MessageBundle
- Once we have visited all the nodes, we pass our existing MessageBundle to the compiler

[existing steps, in compiler]

- *the compiler uses the i18n template parser to add new instances of Message to our existing MessageBundle*
- *the compilation ends, we deduplicate the existing messages (based on the id that we have generated) and we create our messages file*

Using the translations:

AOT application:

For AOT the source code is transformed to replace the service calls by static translations. To do that we will use a TypeScript transformer.

We will inject our transformer into tsc when ngc calls it during the build.

The differents steps are:
[existing steps, in compiler-cli]

- *the developer starts a build with ngc or with ngtools (cli)*
- *we load the translations and generate a TranslationBundle*
- *tsc is called with the source files as parameter*

[new steps, in compiler-cli]

- we pass our transformer to tsc as an additional parameter to be executed *before* compilation
- the transformer is called with an AST as a parameter
- we use a visitor to visit each node
- we search for imports to find our service, if it is imported we store the name (if imported "as"), if not we skip this file
- for each file that imports our service we search for class constructors and we store the local name of our injected service (ex: `private **i18n**: I18nService`)
- for all the nodes in the current class, we search for references of this local service
- for each call to this service we extract the parameters and use them to generate a new Message
- we use the TranslationBundle to return the translation for this new Message
- we replace the service call in the source file with our translation
- the transformer returns, the compilation ends, the compiler is called with the generated AST

[existing steps, in compiler]

- *the compiler uses the i18n template parser to replace i18n strings with their translations (using the TranslationBundle)*
- *the compilation ends and the AOT files are created*

This uses the same AST visitor as for extraction.

JIT application:

For JIT the service will return the translations at runtime. There is no need to transform the source code.

The different steps are:

[existing steps]

- *the developer loads LOCALE_ID, TRANSLATIONS_FORMAT and TRANSLATIONS into the bootstrap of our application*
- *the translations are loaded into a TranslationBundle*
- *at runtime the compiler uses the TranslationBundle to retrieve the translations when it compiles a template*

[new step]

- at runtime the service uses the TranslationBundle to return the translations when it is called

Depends on

- The integration of TS transformers in tsc_wrapped is in development and is planned for 4.1.
- The TypeScript transformers are only available in TS 2.3.0 (~end of April, beginning of May 2017). Our implementation for AOT will only work with TS >2.3.0 and Angular 4.1+.

Security Considerations

I don't think there is any specific security consideration for this feature. The method will return a string. If you use it in your bindings, you can use the existing sanitizer.

Performance Considerations

Translation happens:
- at compile time in AOT, no runtime impact
- at runtime in JIT, overhead each time the string is accessed.

The impact of the service on the payload should be very small. Most of the complexity is in the extract/merge which will be in the compiler/compiler-cli (and maybe some small parts in tsc_wrapped) that are not included in AOT applications.

We need to create benchmarks on AOT generation to measure the impact. We will generate an application with X components and measure the time it takes to merge with and without this feature.

Documentation Plan

We need to improve the existing i18n cookbook to add the new functionalities. We might also need to document those new features for the cli.
I can work on both documentations if needed, but Jesus Rodriguez is probably the best suited for the angular.io docs.

Style Guide Impact

No.

Public API Surface Impact

We will add a new service that developers can import.
We will add a new compiler option to skip extraction/merge from the source code.

We need to think about the name of this new service and its interfaces. The name of the main method should be short, probably `__` or `t` since that is what most existing solutions use.

Developer Experience Impact

By adding this new functionality, developers won't have to rely on hacks anymore to use i18n in their source code. For example right now some people use hidden elements containing their text that they reference in their code in order to get the translations.

There will be an impact during extraction with ng-xi18n because we will have to parse the source files to check for i18n translations (we cannot know if developers use it until we've parsed the code).

Breaking Changes

No.

Deprecations

No.

Rollout Plan

This will be part of a minor release as a new feature, hopefully in 4.x.
We need to validate in g3 to be sure that it works for them.
Existing documentation on angular.io needs to be updated.
We should provide a real application that uses i18n, such as Ames.
We need to make sure that the cli i18n stories are updated.
Once this feature is implemented, Angular i18n will be usable by everyone and we should promote it (blog article, conferences, …).
We should discuss with the Ionic and nativescript so that they use it instead of ngx-translate.

Google3 Impact

The only negative impact should be AOT compilation time (see developer experience impact).
We also need to make sure that they know that it exists and that the implementation works for them.
We need to coordinate with the TS team to know when they will integrate TypeScript transformers into g3.

Work Breakdown

We can expect a release for 4.1 (ideally), or 4.2 (most likely).

Preparation:

- design doc → 2017-03-10
- design review: with berchet@, iminar@, mhevery@, alexeagle@ (for TS transformers), vikram@ (for g3), Marc Laval and Pawel Kozlowski.

Development:

- [easy, ~3 days] interface & class for the service + tests
- [hard, ~2 weeks] TypeScript visitor for extraction + tests
- [hard, ~2 weeks] TypeScript transformer for merge + tests
- [medium] use the TS transformer during AOT compilation
- [easy, ~3 days] merge translations into the message bundle + tests
- [easy, optional]: support variables in the service
- [easy, ~2 days] integration tests
- [medium, 1 week] benchmarks for impact on extraction and AOT compilation time (merge)

Sync with other teams:

- [easy] documentation (angular.io) → check with Jesus Rodriguez
- [easy] documentation (cli): updating the i18n stories
- [easy] updating Ames
- discussing with the Ionic and NativeScript teams to see if they can use this → check with devrel (stephenfluin@ or robwormald@)
- promote the new functionalities with a blog article → check with devrel (stephenfluin@ or robwormald@)

Further developments:

- after the release we might be able to provide a new method `__html()` for html fragments where tags would be replaced by placeholders, but this is a low priority right now and it would probably be a lot of work.

Discussion / Decisions to make:

- Do we keep the variables in the service (3rd parameter) for the initial release, or do we add it to further developments?
  Having the ability to use placeholders for variables has a lot of advantages:
    - you don't have to split your text in multiple service calls if you want to add something dynamic (like the name of the user) in the middle of a sentence
    - complete sentences are way easier to reason about for translators (ex: "An error was triggered by the service {serviceName} because the parameter {parameter} is invalid" is easier to translate than "An error was triggered by the service" + "because the parameter" + "is invalid", you would have to provide context for three sentences instead of one, and if your variables have an expressive name you don't even need a context).
    - you might have to move around placeholders in certain languages, and that is only possible if they are available in the message to translate, not if the position is hardcoded

  Also we already have the ability to use placeholders in templates, not having them in the source code would feel like a regression.

  We could also add a third parameter to the service to add variables that would replace numerical placeholders at runtime:

  ```
  function __(source: string, parameters?: {description?: string,
  meaning?: string, id?: string}, variables: any[]): string
  ```

  The variables can be of any type as long as they have a `toString()` method.

  Example of usage:

  ```
  i18n.__("My name is {0}, just {0}, and my version is {1}",
  ['Angular', 4])
  ```

  Which will return "My name is Angular, just Angular, and my version is 4";

  We can replace the text by its translation in AOT, but the variables will still need to be interpolated at runtime. The previous example with french translations will be transformed to:

```
"Mon nom est " + var1 + ", juste " + var1 + ", et ma version est "
+ var2
```

If they need, the translators can move the placeholders around or even remove them, depending on the language.
Alternative: variables could also be an object with keys instead of an array (numerical placeholders) and we could add it to the parameters instead of adding a third argument. It might be easier to translate like that:

```
i18n.__("My name is {name}, just {name}, and my version is
{versionNumber}", {id: "someId", variables: {name: 'Angular',
versionNumber: 4}})
```

If we add variables, we will need to resolve the problem of escaping those parameters cf: https://github.com/angular/angular/issues/9286
Do we use "`{0}`" or "`{{0}}`"?
Do we use numerical placeholders of an object?
How do we represent them in xmb/xlf?
I propose that we treat them the same way as we treat ICU right now.

Alternatives:
- developers can already use variables in the template
- some use cases for variables can already be solved by the ICU syntax

- Where do we put the code:
    - service in @angular/core/i18n
    - visitor for extraction in @angular/compiler-cli?
    - transformer for merge in @angular/compiler?
- Do we need to add a compiler option to skip extraction/merge from the source code (because of the additional time that it can take)?
- Name of the service: I18nService, I18n, something else?
- Name of the main method of the service: __ (name commonly used in translation frameworks, ex CakePHP), or t (for translate, also used quite a lot), or something else?

Annex

Examples of visitors for TS AST:
- https://github.com/angular/tsickle/blob/master/src/tsickle.ts#L1239
- https://github.com/angular/angular/blob/master/tools/%40angular/tsc-wrapped/src/collector.ts#L50 (not really a visitor pattern, but it shows how to get info from nodes)

Examples of transformers:

- https://github.com/Microsoft/TypeScript/blob/master/src/harness/unittests/customTransforms.ts
- https://github.com/rkirov/ts-transform-demo