# Angular and Internationalization: The New World

*This document is published to the web in the public* [*Angular Design Docs*](https://drive.google.com/#folders/0BxgtL8yFJbacUnUxc3l5aTZrbVk)*/*[*i18n*](https://drive.google.com/folderview?id=0BxyFnbmFLIVxfkpVQzMtcjlTb00zM0xaS0dISXZ4RllnSmp4NGJ6WmNsS21SOFM5ckltZjQ) *folder*

[bit.ly/1pTqd3C](http://bit.ly/1pTqd3C) (permalink)

*Chirayu Krishnappa <chirayu@google.com>  
Igor Minar <iminar@google.com>  
Status: Draft  
  
Last updated: Jan 15, 2015*

[Angular and Internationalization: The New World](#h.ixg45w3363q)

[Meetings (reverse chronological order)](#h.p9fa1uvwqlvz)

[Background](#h.59mdo0ye6e33)

[Interested Parties](#h.q9hybtwhovlp)

[Goal](#h.88csf6ek2vm3)

[Plan](#h.xavmvssm587c)

[Current approaches (in a nutshell)](#h.qipgnv1d77tr)

[angular-translate](#h.6fnc4175k2rs)

[Closure Translation](#h.5j1zo2pz2iwk)

[Issues with adopted Closure tools approach](#h.kclnebm7v2r1)

[Issues with the angular-translate solution](#h.ovtnkc5vs1l4)

[Design considerations](#h.ougg1hltb8h9)

[Locale change and application reload](#h.a4ly5txzzuuo)

[Server side support](#h.gos681ubxetm)

[Tool support](#h.e3ks1rmo95kt)

[What should the solution encompass?](#h.sgxizamjjcw3)

[Corner Cases?](#h.8yij241caj23)

[Proposal](#h.pjtyd32z777w)

[Syntax](#h.5rz5mvvi1sni)

[Marking up text blocks](#h.im1h1zj25aus)

[Marking up html blocks](#h.azaskj186jfo)

[Marking up attributes](#h.mxqwfqv0anmz)

[Marking up URL attributes](#h.2uhjl4tf4p5f)

[Pluralization and Gender](#h.w9i6a28m8efm)

[Pluralization](#h.7jqja3kex513)

[Escaping](#h.pxml8avwu5fb)

[Gender](#h.zddcgnw7hzy7)

[Nesting i18n sections](#h.t4hed34em9mt)

[File Formats](#h.4g39qcn8jioj)

[Transport Formats](#h.tddjreip1kn5)

[JSON runtime format](#h.tlp7qufbm5c1)

[Message ID generation](#h.dh5aq0n03lpr)

[What uniquely determines a message ID?](#h.450fzp4gm8mi)

[When does the message ID change?](#h.xmz1lpfmup1d)

[Full pipeline dry-runs](#h.w6o9slh8vhg2)

[The "Hello World" example](#h.yx4xlhinbw3o)

[Angular template HTML](#h.o24u9t6mo880)

[Extracted message](#h.orytsbllzqj9)

[Message ID](#h.w79lilddjh0m)

[XMB file](#h.skm6zsdt9a6t)

[Translator's view](#h.9o3xots83az0)

[XTB file for German](#h.52ooh6deo7cb)

[Static template for German](#h.n0jcj47xmmwa)

[Dynamic template](#h.13hzbv3lxr7h)

[Using a placeholder (Hello {{user}})](#h.m4qkmdl0l4x1)

[Angular template HTML](#h.tre03qorjx4j)

[Placeholder Name](#h.gseof2lpepuf)

[Message ID](#h.nts86zs8yqjd)

[XMB file](#h.ughrn4agtb29)

[Translator's view](#h.smm6p67prg8f)

[XTB file for German](#h.9zcmega0hctp)

[Static template for German](#h.s5f0g94s3cgl)

[Specifying / overriding the placeholder name](#h.8el03xvhze7v)

[Providing an example](#h.7w160yv2jj7j)

[Message ID](#h.4fqvprjdwt9j)

[XMB file](#h.37wsop40olll)

[Translator's view](#h.dfz2wmfvjune)

[XTB file for German](#h.oyksopk2535)

[Using more than one placeholder](#h.578gwnkr1zd2)

[Angular template HTML](#h.9vokdqmbyt32)

[Message ID](#h.dq03hzozp2c3)

[XMB file](#h.osw89ee29mai)

[Translator's view](#h.70vf8oifphgd)

[XTB file for English](#h.465utjq30nml)

[Tools and plugins](#h.ow1n8zgehrwb)

[Static Extraction](#h.n3r6dbak4x6c)

[Static App Generation](#h.9gm19lekm74i)

[Dynamic mode pre compilation plugin](#h.tk8i82p1nusw)

[IDE and editor support, linting tool](#h.cfil2vj363ku)

[Journey to version 1.0](#h.3emtm35hlw0i)

[$locale files](#h.bqthfptgr1jc)

*[#heading=h.bqthfptgr1jc](https://docs.google.com/document/d/1mwyOFsAD-bPoXTk3Hthq0CAcGXCUw-BtTJMR4nGTY-0/edit?pli=1#heading=h.bqthfptgr1jc)*

## Meetings (reverse chronological order)

* [Video](http://youtu.be/sfoVdgm9bZo) for meeting on [Dec 19, 2014](https://plus.google.com/events/csite01o9nvnmu7o41v5mlcflu0?authkey=CPj13fyFpPzb2gE)
* [Video](https://plus.google.com/hangouts/onair/watch?hid=hoaevent%2Fcmulf8f2s0tetcoo7vceh10vdkg&ytl=x2s3T1BhZUY) for meeting on [Dec 02, 2014](https://plus.google.com/events/cmulf8f2s0tetcoo7vceh10vdkg)
* [Video](https://plus.google.com/hangouts/onair/watch?hid=hoaevent/cd2s08v86ostviej7j0rfnqno8s&ytl=upb_K4W3oVg&hl=en) for meeting on [Nov 14, 2014](https://plus.google.com/events/cd2s08v86ostviej7j0rfnqno8s)

## Background

**What do we mean by Internationalization(i18n), Localization(L10n) and Globalization?** From [section 4.1 of ecma-402](http://www.ecma-international.org/ecma-402/1.0/#sec-4.1): Internationalization of software means designing it such that it supports or can be easily adapted to support the needs of users speaking different languages and having different cultural expectations, and enables worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization of software is commonly understood to be the combination of internationalization and localization.

**AngularJS:**  AngularJS' support for localization/internationalization is limited to date, numbers and currency filters and pluralization via [ngPluralize](https://docs.angularjs.org/api/ng/directive/ngPluralize). The developer is responsible for handling the translation of static text (and resources). While there are tools out there to handle this for web applications in general, Angular applications need special support since a lot of content contains text interpolation and bindings. The existing solutions – both internally at Google and third party – are insufficient and inelegant.

## Interested Parties

Ido Sela - [idos@google.com](mailto:idos@google.com)

Jessica Martin - [jessica.martin@messagesystems.com](mailto:jessica.martin@messagesystems.com)

Martin Probst - [martinprobst@google.com](mailto:martinprobst@google.com)

Pascal Precht - [pascal.precht@gmail.com](mailto:pascal.precht@gmail.com)

Pawel Kozlowski - [pkozlowski.opensource@gmail.com](mailto:pkozlowski.opensource@gmail.com)

Pete Bacon Darwin - [pete@bacondarwin.com](mailto:pete@bacondarwin.com)

Ryan Burke - [ryan@2lemetry.com](mailto:ryan@2lemetry.com)

Stefan Reuter - [stefan.reuter@reucon.com](mailto:stefan.reuter@reucon.com)

Thomas Kruse - [tk@trion.de](mailto:tk@trion.de)

Thorsten Späth - [ts@conserata.com](mailto:ts@conserata.com)

Meeting doodle: <http://doodle.com/346e54gas6pf5v9d>

## Goal

Provide a first class internationalization story for Angular internationalization that works for both AngularJS v1 and v2, and AtScript with a consistent API for internal use at Google as well as external developers.

## Plan

* Requirements/design docs out for review
* Consensus with internal and external folks on the design
* Split out work between external folks, Angular team and internal folks.
* Demonstrate viability for some chosen open source projects. AngularJS' external docs site uses this solution for translation.
* Demonstrate viability for some chosen Google internal project at scale.
* Wider adoption both externally and internally.

## Current approaches (in a nutshell)

### [angular-translate](http://angular-translate.github.io/)

This is an external project supporting AngularJS 1.X. A build step is not required. The application is served as-is. Minimal server side support is needed. All the magic occurs in the application code, mostly in the application template. It is fully dynamic and uses Angular's bindings heavily. It supports switching languages at runtime without a reload, loading language "packs" from the cache or the network (promise based) and fires events when the locale changes. It also supports a variety of approaches for detecting the default locale. Pluralization support requires you to opt-in to a different interpolation scheme (messageformat.js). This approach is transparent with the template cache.

### [Closure Translation](https://developers.google.com/closure/templates/docs/translation)

The Closure tools handle translation by generating multiple versions of the application files – one set of each locale. This approach typically requires server-side support to take advantage of the performance gains that can be obtained by avoiding extra network calls. It requires a full application reload when the locale is changed. There is robust support for pluralization and interpolation baked into the framework. However, it is incompatible with the Angular way of doing thing and there are some Google internal tools to do something similar for AngularJS applications. This is also involves generating multiple versions of the template cache and a build system is a must.

[**angular-gettext**](https://angular-gettext.rocketeer.be/)

TODO

## Issues with adopted Closure tools approach

* The current tools uses regular expressions to parse the template. This makes it brittle and fairly buggy.
* Context sensitive escaping is unclean. (e.g. bugs when the translation introduces quotes.)
* Support is not first class and it's unclear how everything fits together (that is, surprising behavior may not be so uncommon.)
* Missing tool support.
* Does not integrate with external solutions.

## Issues with the [angular-translate](http://angular-translate.github.io/) solution

* Poor pluralization support
  + Pluralization and gender selection is provided by [messageformat.js](https://github.com/SlexAxton/messageformat.js) and is a completely different interpolation syntax than used for everything else. Ref: [Pluralization/The Drawback](http://angular-translate.github.io/docs/#/guide/14_pluralization).
* Security considerations: Ref [Escaping of variable contentla](http://angular-translate.github.io/docs/#/guide/19_security)
* Hacky ways to provide the Angular context for interpolations. (e.g. translate-values.)
* Performance considerations: There are extra bindings introduced. There's typically an extra network requests before content can be shown or all the translations for all the languages are served on initial load. There's support to mitigate this but it requires a good amount of extra code.
* Lots of extra markup. It isn't how one would like to write an Angular application.
* Typically uses made up IDs (hierarchical.) You could use the message text itself, in which case you would have to make it a proper JS string by surrounding it with quotes and escaping characters.
* No clear way to localize chunks of HTML.

## Design considerations

### Locale change and application reload

There are two popular approaches to handling a locale change. One approach, common to applications at Google, is to reload the entire application/page. The other approach, used by angular-translate, is to have the app re-render itself after loading the new locale. Each approach has its tradeoffs.

|  |  |
| --- | --- |
| **Full reload** | **No reload** |
| * Performance win: By reloading the entire app, there is no need to track and update the bindings/UI that change as a result of the change in locale. * The locale changes so rarely so the cost of the reload is incurred only rarely. * No information/state is (typically) lost. The user is presumably changing the language because they could not understand the earlier language. This means that they don't have unsaved information in the application. * Cannot support multiple languages in the same application view without extra pain and bloat. * Extra server side support is needed: Typically, the server must perform cookie/user agent analysis to decide which localized version of the application code should be returned to the server. This also causes a cache miss. * The server is now responsible for determining the default localized version to serve. (e.g. cookies / geo-ip / Accept-Language header, etc.) | * Minimal server side support: The same version of the application code is served by the server. However, the server must also serve translated message bundles back to the application (much easier server side change) or have tools that embed all translations in the application code at build time. * Must track all the pieces of the UI that need to be updated when the locale changes and perform this upon the change. In addition, if the new language strings are being loaded over the network, this could take time and the UI needs to indicate this in some way to the user. * Allows one to support multiple languages in the same view. As an example, a page could display a table showing how the user's advertising message might look in different locales. This is fairly easy to do with this approach since it's fairly simple to have an locale per root node. * The server is not *required* to determine the locale from the request – the client side can use cookies, navigator.language, navigator.browserLanguage,navigator.systemLanguage, navigator.userLanguage, and JS APIs to determine the language. However, it's still be beneficial for the server to do some of this (e.g. to serve the likely language pack together with the application) but that can be done at a later stage in development or for prod. |

### 

### Server side support

What extra support is needed by the server? This might include:

* Perform request header/cookie/geo-ip. sniffing and serve different localized versions of the application. (geo-ip sniffing also requires receiving the true external IP of the client in the presence of reverse proxies, etc.)
* API to return translated message packs given a locale.
* Remember the user's language locale preference in the database. (Out of scope for this doc.)
* Serve localized versions of static resources – images, pdf documents, etc.

### Tool support

* Tools to extract messages with description to provide to translators / translation services. Different translation services have different incompatible formats so this needs to be pluggable to support the various output formats (both internal and external such as xliff, gettext/po, etc.)
* Tools to convert the translation results into a format that can be consumed by our translation solution. Since we should support multiple approaches, we should standardize on a default format for our solution and provide tools to translate it into other formats (e.g. the JSON format used by angular-translate.) The tools that inline the results to produce different application bundles per locale would work off of this format.
* Tools to inline the translations and produce different application bundles per locale.
* Application layout to support fallback schemes, overrides, etc. efficiently (less duplication.)

### What should the solution encompass?

* Tool support. Some server side support plugins.
* First class support for pluralization with all its nuances.
* BiDi support and best practices. (nice explanation for all of you that are not so familiar with BiDi/RTL (like me): <http://www-01.ibm.com/software/globalization/topics/bidi/> )
* Visual layout support. This one is more complicated. Suppose you have a component that has a visual layout like DROPDOWN\_FOR\_FIELD\_NAME DROPDOWN\_FOR\_PREDICATE text field to support something like "'username' 'must contain' user-input". This component might need to have a different layout per locale. Or be rewritten to avoid that. How are folks dealing with this today? What are the best practices? How can we make sure that our solution is helping this case?
* Multiple languages in the same view? (e.g. different columns of a rendered table could be in different languages. Detect HTML [lang](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes#lang) attribute and use that instead of current locale?) (Financial/Accounting applications may need to display multiple formatted currency values on the same screen, so number and currency filters should allow this. This is passed as a parameter in ECMA402.)
* Existing filters would continue to work – e.g. <https://docs.angularjs.org/api/ng/filter/currency>. Provide support for parsing dates, numbers and currency in the current locale.
* Lazy loading of additional languages?
* Client side caching of "language packs".
* A list of best practices, common pitfalls and an FAQ. In addition to all the things called out here, it should also answer questions like: "How should relationships be represented canonically in the backend?" (e.g. while English has "sister", some languages can only say "older sister" or "younger sister" and you would need a conjunction to include both.) Best practices for storing dates server-side (UTC is best). Common concerns and and solutions should be supported. If users go down our solution not having all this information, they'll continue to feel like i18n is super hard and will not have a great experience. (Sort of like the "battle" for unicode-clean code.)
* Handle translations in:
  + Static HTML that isn't part of Angular. (We need this because we would like folks to use the tools we provide for their entire application.)
  + Text interpolations in templates.
  + Stylesheets and templates corresponding to Angular components.
    - Handle text content and URLs in stylesheets if the same stylesheet is being re-used for all locales.
    - Handle using different style sheets per locale with fallbacks. (allows much more flexibility such as adjusting margins, spacing, etc. per locale.)
  + URLs. (need support to automatically pick localized versions of static resources.)
    - Dynamic URLs typically do not need to be rewritten if the server is producing localized content based on the cookie/etc.
    - Cachable/Static URLs typically do need to be rewritten. To avoid rewrites and have everything work, the server would have to supply a "Vary" HTTP header for the cookie used (or query parameter if that's the case.) However, this fails for the default/not logged in case where there is no cookie and the locale is being automatically determined by geo-IP, etc. The first response can be cached and served to users with a different geo-IP by intermediate caching proxies.
* Locale determination: Each application may want to do something slightly different. We should ship with some common configurable strategies. The following would be some of the inputs used:
  + When a user has signed in
    - If the application stores the default locale for the user, that might be the one to use even when the user has traveled to a different country.
    - If a logged in user changes their locale via a menu, this might set a cookie or change a query parameter. Such a setting would typically override the user's default locale choice. It's up to the application to choose to update the user's locale choice in the backend if it makes sense.
    - The application might want to do some things differently when the user's locale as determined without cookies/user info changes. e.g. When a user visits a different country, an application might choose to stick with the user's locale choices for language but use the timezone as determined by geo-IP (typically involving prompting the user.)
  + When a user has *not* signed in
    - In the absence of cookies or query parameters, use the geo-IP and/or the HTML Accept-Language header.
    - When there are locale cookies or query parameters set, they should typically override the geo-IP / Accept-Language based determination.
* Avoid page reloads on language changes.
* Ensure we provide the same security with translations enabled as we do today.
* Performance. It's a feature.
* Versioning concept for translation files / keys on how to get a diff out of already released translations compared with new or changed keys to be translated by agency/whoever (so they don’t have to scan all translations and only get a excerpt of changed or new keys).
* Support for re-running extraction where only changed messages are sent for re-translation.

#### Corner Cases?

* How about names of users? e.g. If a user has a Chinese name and an American name and you're displaying a form for the user, should you display the Chinese name when you're showing the user list in Chinese and the English name for English? This seems fairly out of scope but something we shouldn't preclude in our solution.
* Support for the lang="\_\_" attribute. When the HTML contains different DOM trees with different lang= values, we're dealing with an override there.

**Super quick example:**

Google internal example: `[[Save|Save post button]]`

[[Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.|Label to show the current length of the comments and the maximum length allowed.]]  
  
 [[<span class="err">This post does not exist.</span>

<a href="#{{app.basePath}}/posts?status=1">Return to posts</a> or

<a href="#{{app.basePath}}/posts/new">create a new post</a>.|Shown when you

try to view a post that does not exist. The first

link goes to the list of all posts. The second link

creates a new post.]]

Angular Translate example: `{{ 'TRANSLATION\_ID' | translate }}`

1. <p translate>PASSED\_AS\_TEXT</p>
2. <p translate="PASSED\_AS\_ATTRIBUTE"></p>
3. <p translate>{{ 'PASSED\_AS\_INTERPOLATION' }}</p>
4. <p translate="{{ 'PASSED\_AS\_INTERPOLATION' }}"></p>
5. <p translate="VARIABLE\_REPLACEMENT" translate-values="{ name: 'PascalPrecht'}"></p>
6. <p translate="VARIABLE\_REPLACEMENT" translate-value-name="PascalPrecht"></p>

<div ng-show="!isClosed()">

{{backers | number}}

<ng-pluralize count="backers" when="{'one': 'person has', 'other': 'people have'}"></ng-pluralize>

funded {{funded | currency}}. Project has {{ timeLeft() }} until close.

</div>

<div ng-show="isClosed()">

{{backers | number}}

<ng-pluralize count="backers" when="{'one': 'person has', 'other': 'people have'}"></ng-pluralize>

funded {{funded | currency}}. Project closed on {{end | date }}

</div>

# Proposal

The nuances of dynamic mode

A fully dynamic implementation would neither reload the app nor destroy/recreate DOM sections when the locale changes. This is how angular-translate behaves. A less dynamic approach would avoid reloading the application but would destroy/recreate the DOM upon a locale change. Preserving the JS state – particularly on the child scopes – might require some support from the application (TBD). Supporting the fully dynamic mode is not that much harder but it does involve setting up extra watches. At this point, I'm unsure if it adds a lot of extra complexity – and if it does, then it will be supported in the final version but perhaps not in the prototypes along the way. (Personally, I don't think it's that much more work – but I'd rather not surprise people if the initial prototypes don't support it.)

questions:

Support for rendering dates and numbers (and number abbreviations) is included? For each country region? You have to be able to force some decimal representation, (some foreign people working on different decimal char representation countries need this - for example American people working on Spain). Support for timezones -user’s current and business logic fixed-).

## Syntax

This syntax would be recognized during a precompilation step (in dynamic mode) or extracted and replaced by tools (in static mode via a build step.)

Our choice of syntax is based upon the following constraints:

* Developers should be able to load their application without an explicit build / preprocessor step using a simple web server.
* There should be an easy way for developers to internationalize their existing applications. This means that, for example, we can't require that messages in templates are surrounded by div tags since that would affect CSS, DOM queries, etc.
* The syntax should be a standard or based on a standard. I'm striving for compatibility with the [ICU MessageFormat syntax](http://userguide.icu-project.org/formatparse/messages)
* Valid HTML – to support the dynamic mode without precompilation

**Note:** Unless explicitly called out, i18n messages *cannot* be nested. See [Nesting i18n sections](#h.t4hed34em9mt).

### Marking up text blocks

**<span>** *<!--i18n: Label to show the current length of the comments and the maximum length allowed.-->*  
 Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.  
 *<!--/i18n-->***</span>**

If you don't want to provide the description, you can simplify to:

**<span>** *<!--i18n-->*  
 Limit is 255 characters for comments. Current: {{item.comment.length || 0 // ex}}.  
 *<!--/i18n>***</span>**

As a special shorthand, if the entire contents of an element is one translatable block, you can avoid using begin and end tags. e.g. If the above was the exact contents of a span block, you would write the following:

**<span i18n="***Label to show the current length of the comments and the maximum length allowed."***>**  
 Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.  
**</span>**

The description is optional like before. When not providing a description, you can also omit the i18n attribute and use the [implicit syntax](#id.kr6g463io40p) as shown below.

**<span>**  
 Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.  
**</span>**

### Marking up html blocks

*<!--i18n: Shown when you try to view a post that does not exist. The first*  
 *link goes to the list of all posts. The second link creates a new*  
 *post.Label to show the current length of the comments and the*  
 *maximum length allowed.-->*  
 **<span** class="err"**>**This post does not exist.**</span>**  
 **<a** href="#{{app.basePath}}/posts?status=1"**>**Return to posts**</a>** or  
 **<a** href="#{{app.basePath}}/posts/new"**>**create a new post**</a>**.  
*<!--/i18n>*

The variant with the i18n attr and for omitting the description are also available here. For example, if you are OK with surrounding the HTML snippet with a div, then you could have also written that as:

**<div i18n="***Shown when you try to view a post that does not exist. The first*  
 *link goes to the list of all posts. The second link creates a new*  
 *post.Label to show the current length of the comments and the*  
 *maximum length allowed*"**>**  
 **<span** class="err"**>**This post does not exist.**</span>**  
 **<a** href="#{{app.basePath}}/posts?status=1"**>**Return to posts**</a>** or  
 **<a** href="#{{app.basePath}}/posts/new"**>**create a new post**</a>**.  
**</div>**

### Marking up attributes

*<!-- With a description -->*  
**<input** placeholder="First Name" i18n-placeholder="Placeholder for user input control"**>**  
  
*<!-- No description -->*  
**<input** placeholder="First Name" i18n-placeholder**>**

*<!-- No description for standard HTML attributes containing user-visible text -->*  
**<input** placeholder="First Name"**>**  
This syntax also works for custom elements and web components.

### Marking up URL attributes

**<img** src="/static/logo.png" i18n-url-src="Product logo shown in the title area of every page"**>**

The use of the i18n-url prefix specifies that the corresponding attribute is a URL and should be rewritten with the locale specific version of that url. This should play with ng-src and friends for Angular 1.x. Translators would provide localized versions of such assets.

The translated URL would be "/en/US/static/logo.png". Note that this rewrite is the default / suggestion. You will be able to configure the rewrites to be something else but the essential structure would stay the same (i.e. you'd have to use the "/en/US/" substring somewhere in the rewritten URL. This is to support easy fallbacks where, say, the same logo will be used for all English locales regardless of region. This doesn't have to be the case and we can revisit if it's a burden.)

For images in particular, we need a way to mark up related URLs together. For example, related image URLs will occur in image-set for 1x / 2x / high dpi resources. This bundling is also part of the picture element. TBD: A method to groups such related URLs together that will be future proof (support web components / custom elements.)

### Pluralization and Gender

This syntax here is based upon the [ICU MessageFormat Syntax](http://userguide.icu-project.org/formatparse/messages). Pluralization is handled via the plural syntax and gender via the generic select syntax.

**Note:** In the initial default configuration, the following syntax for both pluralization and gender will not be recognized in the [*implicit syntax*](#id.kr6g463io40p) sections. This preserves backwards compatibility. The linter and tools should warn when such syntax is recognized and is missing the i18n markup around it.

#### Pluralization

{{{messages.length}}, **plural**,  
 **=0** {You have <b>no</b> messages.}  
 **=1** {You have one message.}  
 **other** {You have # messages.}  
}

In this proposal, it should be noted that the # symbol is used to indicate the actual count. The translator can move it around or throw it out if it's not needed.

##### **Escaping**

All ICU syntax, including the # symbol, can be escaped with single quotes. A single quote itself can be escaped with a single quote. (This is similar to how a backslash (\) escapes work in C strings.)

#### Gender

This is based on the more general select syntax. Here's an example:

{**{{friend.gender}}**, **select**,  
 **male** {Invite him to your party.}  
 **female** {Invite her to your party.}  
 **other** {Invite them to your party.}  
}

**Note**: The choices (male, female, etc.) are case sensitive.

Translation text extracted from i18n sections

The text extracted between in the above methods (except for attributes), will be canonicalized a little bit before being used as the text sent to translators. The operations performed would include:

* Strip leading and trailing whitespace.
* Remove comment tags and collapse whitespace where they were removed.

e.g. The extracted text for the following HTML

**<span>  
 <!-- todo: add more unicorns -->**  
 Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.  
**</span>**

is

Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.

Implicit syntax

In many cases, we can extract text that should be localized without explicit markup. As an example, the contents of all span tags that contain any text are candidates for localization. e.g.

**<span>** Limit is 255 characters for comments. Current: {{item.comment.length || 0}}.  
**</span>**

The extraction tools should extract this text out of the span, but should (by default) warn the developer that the extracted text did not provide any description. One could silence the warning by adding an i18n attribute with description to the span.

Similarly, we can extract known HTML5 attributes for localization. An example is the placeholder attribute of input[text]. Here too, the tool would warn about the missing i18n-placeholder attribute that would provide the description to the translators.

In order to support web components and future HTML elements, the tool should be configurable with an external file that would specify attributes of elements that should be internationalized.

## Nesting i18n sections

i18n sections can be nested in select circumstances. The following uses are allowed:

* You can always specify \*[i18n-ATTR] in a message that's already being extracted for i18n.
* Gender messages (i.e. using the select syntax of ICU MessageFormat) can be nested to arbitrary depth. But note that this can easily result in a combinatorial explosion for the translator!
* Plural messages (i.e. using the select syntax of ICU MessageFormat) can only contain simple messages and cannot contain other plural or select messages. Unlike the select ICU syntax, plural nesting creates a much bigger explosion of combinations (other languages can have many more than the standard 3 selectors typical of English.)

The following is allowed:

*<!--i18n: Shown when you try to view a post that does not exist. -->*  
 **<span** class="err"**>**This post does not exist.**</span>**  
 **<a** href="/posts"  
 title="All Posts"  
 i18n-title="Title for a link to show all posts"**>**Return to posts**</a>** or  
 **<a** href="/posts/new"  
 title="New Post"  
 i18n-title="Title for a link to create a new post"**>**create a new post**</a>**.  
*<!--/i18n>*

The following is NOT allowed:

*<!--i18n: Shown when you try to view a post that does not exist. -->*  
 Message  
 <!-i18n: THIS NESTING IS NOT ALLOWED -->  
 Sub-message  
 *<!--/i18n>*  
*<!--/i18n>*

Opting out of translation

The HTML5 "translate" attribute can be used to opt out of translation. This will be honored even inside sections that are enclosed in section that have been marked explicitly for translation. TODO: how would this work with message extraction? Perhaps we can replace them with placeholders—with the message id depending on the contents of the placeholder.

## File Formats

### Transport Formats

We will define common (JS) structures for representing extracted and translated messages that will be used for communication between all our tools.

The two most popular formats appear to be [XLIFF](http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html) and [gettext/po](http://www.gnu.org/software/gettext/manual/gettext.html#PO-Files) (portable object.) We will ship with adapters translate between these files and our JS structures. Folks can write their own adapters for their other file formats.

The ICU standard uses the XMB/XTB file formats that have several tools that can translate back and forth. This will be a primary supported format.

The [translate/translate](https://github.com/translate/translate) github project (GPLv2, [documentation](http://docs.translatehouse.org/projects/translate-toolkit/en/stable-1.12.0/)) might be handy for translating between formats.

### JSON runtime format

These files will be loaded by applications that do translations dynamically. As such, it will be an easy to use and smallish JSON format (tbd).

## Message ID generation

**See also:** [Message IDs](https://docs.google.com/document/d/1MdvNoNxHWU6J9UzVvXegTnH3Kmk7FxedFyZmXE8aP-M/view) document.

Message IDs are IDs that uniquely identify a translatable message. If a message, M, appears in multiple parts of the application and means the same thing, it is considered to be the same message and should only be translated once. It will have a unique message ID. However, if there is another message, which is identical in the source language to M (e.g. there might even be no difference visually), but has different semantics/meaning, then it should be considered a different message, assigned a different message ID and has to be translated on its own.

As an example, consider the word "crane". It might appear in a dropdown for a list of birds. It might also appear in a dropdown for machines. Though it is spelled exactly the same in English, that won't be true in other languages and we can't use the same translation in both dropdowns. This word needs to be translated two times and assigned two different message IDs.

#### What uniquely determines a message ID?

* the message including all the placeholders (canonicalized)
* the meaning of the message

This can be written as (e.g. python pseudocode):

*# message id is a hash of the 2-tuple (canonical message, meaning)*  
message\_id = hash((canonical\_message, meaning))

NOTE: The exact hash function is unspecified for now. A good default will be chosen. Applications will be able to plug in their own so that they can use their method of message ID construction that might already be using.

In the example for the word crane, this might be indicated the following way:

**<select>**  
 **<optgroup** label="birds"**>**  
 …  
 **<option** i18n="bird|tall wading bird with long legs, bill and neck"**>**Crane**</option>**  
 …  
 **</optgroup>**  
 **<optgroup** label="machines"**>**  
 …  
 **<option** i18n="machine|large machines for lifting or moving heavy objects"**>**Crane**</option>**  
 …  
 **</optgroup>**  
**</select>**

During extraction, here is how the message IDs are calculated.

For the 1st instance of “crane”: message\_id = hash(("Crane", "bird")) → H1

For the 2nd instance of “crane”: message\_id = hash(("Crane", "machine")) → H2

In an XMB file, this would appear as:

**<msg** id="H1"  
 meaning="bird"  
 desc="tall wading bird with long legs, bill and neck"**>**Crane**</msg>**  
**<msg** id="H2"  
 meaning="machine"  
 desc="large machines for lifting or moving heavy objects"**>**Crane**</msg>**

#### When does the message ID change?

As stated earlier, the message ID should be equivalent to the tuple (canonical message, meaning).

Therefore, if one were to change one of the messages from:

**<option** i18n="bird|tall wading bird with long legs, bill and neck"**>**Crane**</option>**

to

**<option** i18n="bird|tall wading bird with long legs, bill and neck that looks like a heron"**>**Crane**</option>**

the message ID will **not** change because neither the actual message nor the meaning changed (a change in description alone makes no difference—no new message ID ⇒ no retranslation.)

However, changing either the meaning or the message **will** result in a new message ID (and any changes in the description will show up in the new extraction.)

Original:

**<option** i18n="bird|tall wading bird with long legs, bill and neck"**>**Crane**</option>**

Change in the message (a change in case is also a change in the message):

**<option** i18n="bird|tall wading bird with long legs, bill and neck"**>**crane**</option>**

Changing the meaning will also cause a change in the message:

**<option** i18n="gruiform|tall wading bird with long legs, bill and neck"**>**Crane**</option>**

Here, the extracted message will look like this:

**<msg** id="H11"  
 meaning="gruiform"  
 desc="tall wading bird with long legs, bill and neck"**>**Crane**</msg>**

## Full pipeline dry-runs

In this section, we will see some sample dry run's of a full translation flow of some translatable snippets. This is intended to provide insights into implementation details. Since we will be supporting multiple formats and the process can be customized, I've picked a sample configuration to keep it simple. We will focus on Angular template HTML, extraction into XMB files, translation into XTB files, and production of translated per-locale templates.

### The "Hello World" example

#### Angular template HTML

**<span** i18n="Refer http://www.wikipedia.org/wiki/hello\_world"**>**Hello, world!**</span>**

#### Extracted message

The message that's extracted for translation would be:

Hello, world!

#### Message ID

The message ID is constructed as message\_id = hash(("Hello, world!", "")). Let's say this is ex1m1.

#### XMB file

**<messagebundle>**  
 …  
 **<msg** id="ex1m1" desc="Refer http://www.wikipedia.org/wiki/hello\_world"**>**Hello, world!**</msg>**  
 …  
**</messagebundle>**

#### Translator's view

The translator, when they pick their locale of choice, should see the original English message—"Hello, world!". They can replace the entire message with the final translated version.

#### XTB file for German

**<translationbundle** lang="de"**>**  
 …  
 **<translation** id="ex1m1"**>**Hallo Welt!**</translation>**  
 …  
**</translationbundle>**

#### Static template for German

**<span>**Hallo Welt!**</span>**

#### Dynamic template

In the dynamic version of the application, Angular, in a pre-compile step, will compute the hash for the message and then look up the hash in the loaded locale JSON file. It will then replace the English text with the translated text before continuing with compilation.

### Using a placeholder (Hello {{user}})

#### Angular template HTML

**<span** i18n="Message greeting the user"**>**Hello {{user}}!**</span>**

#### Placeholder Name

**TODO(chirayu):** Add a section to this doc describing how placeholder names are constructed, how they can be specified/overridden in Angular expressions, when they **must** be specified/overridden (if automatic generation results in a conflict), etc.

In this case, let's say that the placeholder name is USER.

#### Message ID

The message ID is constructed as:

message\_id = hash(("Hello, <ph name="USER"/>!", ""))

Let's say this is ex2m1.

#### XMB file

**<messagebundle>**  
 …  
 **<msg** id="ex2m1" desc="Message greeting the user"**>**Hello, **<ph** name="USER"**/>**!**</msg>**  
 …  
**</messagebundle>**

#### Translator's view

The translator, when they pick their locale of choice, should see the original English message with a placeholder. e.g.

Hello USER!

The translation tool will indicate to the translator that USER here is a placeholder. The translator does not know anything more about USER—is it their first name? full name? etc. They can only make an educated guess. (If you're interested in names in general, check out [Falsehoods Programmers Believe About Names](http://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/).)

In addition, upon translation, the tool will verify that the placeholder is still present in the translated message.

#### XTB file for German

**<translationbundle** lang="de"**>**  
 …  
 **<translation** id="ex2m1"**>**Hallo **<ph** name="USER"**/>**!**</translation>**  
 …  
**</translationbundle>**

#### Static template for German

**<span>**Hallo {{user}}!**</span>**

#### Specifying / overriding the placeholder name

You can manually specify the placeholder name that the translator sees in the following way:

**<span** i18n="Message greeting the user"**>**Hello {{user // i18n(ph="USER\_FIRST\_NAME")}}!**</span>**

On a second extraction run, our change would cause a new message ID to be generated resulting in a retranslation of the same message. A change in the placeholder name, like a change in meaning, is considered significant.

#### Providing an example

You can provide an example for the placeholder in the following way:

**<span** i18n="Message greeting the user"**>**Hello {{user // i18n-ph(USER\_FIRST\_NAME|Peter)}}!**</span>**

Note that simply providing an example does not cause a new message ID to be generated and so there will be no new translation requested for that message. However, when the message is sent to the translator (due to the meaning, placeholder name or the message itself changing), the extracted message will contain the example for the translator to see.

#### Message ID

The message ID is constructed as:

message\_id = hash(("Hello, <ph name="USER\_FIRST\_NAME"/>!", ""))

Let's say this is ex2m2.

#### XMB file

**<messagebundle>**  
 …  
 **<msg** id="ex2m2" desc="Message greeting the user"**>**Hello, **<ph** name="USER\_FIRST\_NAME"**>**  
 **<ex>**Peter**</ex></ph>**!**</msg>**  
 …  
**</messagebundle>**

#### Translator's view

This time, the translator will see the original English message with a placeholder that also has an example. e.g.

Hello USER\_FIRST\_NAME!

The translation tool will indicate to the translator that USER\_FIRST\_NAME here is a placeholder and that there are examples for it.

It should show an example message where placeholders that have examples have been filled in. The translator should be able to see this example:

Hello Peter!

#### XTB file for German

The resulting XTB files typically only contain the placeholder names but not the examples. One will need to look up the XMB file to look up the corresponding example (or another database that is storing your messages by ID for you.)

**<translationbundle** lang="de"**>**  
 …  
 **<translation** id="ex2m1"**>**Hallo **<ph** name="USER\_FIRST\_NAME"**/>**!**</translation>**  
 …  
**</translationbundle>**

### Using more than one placeholder

#### Angular template HTML

*Feel free to suggest a better example!*

**<span** i18n**>**{{phone // i18n-ph(PHONE|555-867-5309)}} belongs to {{user // i18n-ph(PERSON\_FIRST\_NAME|Jenny}}**</span>**

#### Message ID

The message ID is constructed as:

message\_id = hash(("<ph name="PHONE"/> belongs to <ph name="PERSON\_FIRST\_NAME">", ""))

Let's say this is ex3m1.

#### XMB file

**<messagebundle>**  
 …  
 **<msg** id="ex3m1"**>**  
 **<ph** name="PHONE"**><ex>**555-867-5309**</ex></ph>**  
 belongs to **<ph** name="PERSON\_FIRST\_NAME"**><ex>**Jenny**</ex></ph>**  
 **</msg>**  
 …  
**</messagebundle>**

#### Translator's view

The translator, when they pick their locale of choice, should see the original English message with a placeholder. e.g.

PHONE belongs to PERSON\_FIRST\_NAME

The translation tool will indicate to the translator that PHONE and PERSON\_FIRST\_NAME are both placeholders.

The translation UI will also show examples for each placeholder and allow the translator to see an example message. In this case, the example would be 555-867-5309 belongs to Jenny.

#### XTB file for English

*Note: I only monolingual so feel free to provide a different language translation here.*

**<translationbundle** lang="en"**>**  
 …  
 **<translation** id="ex3m1"**><ph** name="PHONE"**/>** belongs to **<ph** name="PERSON\_FIRST\_NAME"**/></translation>**  
 …  
**</translationbundle>**

## Tools and plugins

### Static Extraction

Use case: Extract messages from source files into a file that will be handed over to the translators. Both static and dynamic modes require this tool.

This tool would discover the Angular templates and HTML files of the application, perform validation and produce warnings, accept external definition files for implicit syntax (new HTML elements and Web Components) and use one of the adapters to produce XLIFF / PO / other files.

It should be easy to integrate into existing build environments (e.g. by providing something similar to GCC's -MD -MF file options.

### Static App Generation

Use case: Translation results are available and we want to generate a static version of the app for different locales.

This tool would read the translated results from a file and produce a different version of the application for each locale. The translation results would typically be inlined into the generated files. This tool should be easy to integrate into build environments.

### Dynamic mode pre compilation plugin

In fully dynamic mode, Angular will recognize i18n markup in a pre-compilation step. This can be implemented as a plugin. (The beta version may be an integral part of Angular and this can later be split off into a plugin.) It will need to use the JSON translated output to replace the i18n markup with translated results.

### IDE and editor support, linting tool

We will ship with a linter tool and provide modules for Angular Hint. In addition, we should provide rules/support for ESLint/JSHint/Esprima/etc.

It would be a good idea to ship with tools that IDEs can consume easily to perform their completion, warning and error reporting operations. We need first class support for the most popular IDEs and editors – WebStorm, Sublime Text, Vim, Emacs.

## Journey to version 1.0

See this [document](https://docs.google.com/document/d/1-pLAhklbR7CMLkY4pYgwjoDCLyNlNGVnO_lDZiuN9KA/edit). And these [github issues](https://github.com/angular/i18n/issues).

**FYI:** I created a new repo under Angular somewhat arbitrarily to keep from polluting the primary AngularJS repo. We can change this to something else later if that makes more sense.

## $locale files

It's great that FIRSTDAYOFWEEK and WEEKENDRANGE have been added in v1.4 i18n files. Would love some documentation on these. [Either I don't understand the or some of the data is wrong](http://stackoverflow.com/questions/30076505/angularjs-i18n-nglocale-firstdayofweek-and-weekendrange).

It would be great if these files could have the following attributes added, as they appear in the jQuery datepicker i18n files

* dayNamesMin: ['ح', 'ن', 'ث', 'ر', 'خ', 'ج', 'س'], // one or two character day names
* isRTL: true, // or `direction` as in github issue [#604](https://github.com/angular-translate/angular-translate/issues/604)
* showMonthAfterYear: false,
* yearSuffix: ''

We can't build as good a Angular 'native' datepicker as the jQuery one without these. And by at least adding isRTL, a developer could easily add their own ng-class for RTL languages.