**Tr8n Translation Engine**

**Integration Guide**

Version: 1.0

Author: Michael Berkovich



Table of Contents

Introduction 3

Why Is i18n Not A Good Solution? 3

Why Is Facebook Solution Not Going to Solve Your Problems? 3

What Does Translations Engine Offer You Out Of The Box? 4

Installation Instructions 5

Third Party Plugins 5

Translations Engine Customizations 5

Label Internationalization 7

Label And Description 7

Translations With Tokens 8

Nested Translations With Tokens 8

Gender Dependent Tokens 9

Viewing User Token and Multiple Gender Dependent Tokens 10

Translation Options 12

Number Dependent Tokens And Hidden Tokens 13

Putting It All Together 16

Lambda Tokens 17

Extended Syntax Support 19

Strftime Syntax 20

Language Localization 21

Appendix 22

Supported Languages 22

# Introduction

Geni translations engine allows for crowd-source translations and management of any internationalized text throughout any rails based web site. The power of the engine comes from its simple and friendly user interface that allows user to rapidly translate the site into any supported language. The flexible and robust rules engine that powers the translations engine allows for any types of language specific numeric rules as well as any combinations of gender and number rules in one translatable sentence. Users themselves can provide information on what sentences depend on gender rules, number rules or both. The language specific numeric rules can be created and managed for any language in the advanced user interface. The engine also provides powerful admin tools that allow admins to manage translators, languages, translation keys, translations and glossary items.

## Why Is i18n Not A Good Solution?

1. String organization overhead + naming convention overhead
2. No support for complex numeric and gender based rules
3. New translations addition overhead - slow process for translating new features
4. Translators cost and language limitations

Bottom line: TRANSLATIONS MUST BE DONE USING CROWD-SOURCED APPROACH

Facebook is a pioneer in this approach and has proven to successfully translate their entire site into dozens of languages. It worked for them IT WILL work for us.

## Why Is Facebook Solution Not Going to Solve Your Problems?

1. Facebook translations support for third party web sites is done through Facebook Connect Client Side XFBML library, which entails that the strings will be substituted using a javascript method by first presenting the string in English, then sending the strings to Facebook Server for translations and substituting them in the page. This introduces a delay.
2. It only works for Facebook Connected users, who must enable languages on Facebook and choose the language in Facebook.
3. XFBML interface is very buggy - popup windows freeze and don't work.
4. Translating context based strings takes users to Facebook.
5. Facebook has a limitation of at most 4000 phrases
6. The entire site will have to be wrapped in XFBML tags.

## What Does Translations Engine Offer You Out Of The Box?

The following is a list of features that translations engine supports by default after the installation:

* Support for over 100 languages
* Support for left to right and right to left languages (requires css integration)
* Inline and bulk mode translations
* Translations based on gender and numeric rules
* Translations voting system
* Translations reporting system
* Software keyboard for over 50 languages
* Google suggestions for over 50 languages
* Keyword glossary
* Translator awards
* Translations site map
* Language discussion boards
* Regional translator management
* Translator monitoring system
* Language rule management tools
* Glossary management tools
* Help section
* Administration tools for configuring all aspects of the translations engine

# Installation Instructions

This section will describe how to get the translations engine up and running on your system.

## Third Party Plugins

Translations engine depends on the following plugins:

* will\_paginate, created by Mislav Marohnić
* will\_filter, created by Michael Berkovich

To install the third party plugins with translations engine, follow these steps:

|  |
| --- |
| $ script/plugin install git://github.com/mislav/will\_paginate.git    $ script/plugin install git://github.com/berk/will\_filter.git  $ rake will\_filter:sync  $ rake db:migrate    $ script/plugin install git://github.com/berk/translations.git  $ rake translations:sync  $ rake db:migrate |

## Translations Engine Customizations

|  |
| --- |
| **enabled**: true  **site\_info:**  default\_locale: 'en-US'  title: 'Geni'  default\_url: '/home'  user\_class\_name: 'Profile'  user\_name\_method: 'name'  user\_mugshot\_method: 'mugshot'  user\_link\_method: 'link'  **rules\_engine:**  viewing\_user\_token: 'viewing\_user'  key\_source\_tracking\_enabled: true  paranoia\_mode\_enabled: true  number\_based\_tokens: 'count, num, age, hours, minutes, years, seconds'  gender\_based\_tokens: 'user, profile, actor, target, partner, parent, child, sibling'  minimal\_translation\_rank: 1  # specify localization values in the default language of the site  **localization:**  default\_day\_names: 'Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday'  default\_abbr\_day\_names: 'Sun, Mon, Tue, Wed, Thu, Fri, Sat'  default\_month\_names: 'January, February, March, April, May, June, July, August, September, October, November, December'  default\_abbr\_month\_names: 'Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec'    # add any necessary formats here and use them througout the site  **custom\_date\_formats:**  default: '%m/%d/%Y', # 07/4/2008  short\_numeric: '%m/%d', # 07/4  short\_numeric\_year: '%m/%d/%y', # 07/4/08  long\_numeric: '%m/%d/%Y', # 07/4/2008  verbose: '%A, %B %d, %Y', # Friday, July 4, 2008  monthname: '%B %d', # July 4  monthname\_year: '%B %d, %Y', # July 4, 2008  monthname\_abbr: '%b %d', # Jul 4  monthname\_abbr\_year: '%b %d, %Y', # Jul 4, 2008  date\_time: '%m/%d/%Y at %H:%M', # 07/4/2008 at 10:15 |

# Label Internationalization

The function for internationalizing labels has the following signature:

|  |
| --- |
| tr(LABEL, DESCRIPTION = "", TOKENS = {}, OPTIONS = {}) |

* LABEL is a string and is always required
* DESCRIPTION is a string and is not required
* TOKENS and OPTIONS are hashes and are not required
* LABEL and DESCRIPTION

## Label And Description

Simple Translations:

|  |
| --- |
| tr("Hello World!!!")  tr("This is a simple fully self-contained sentence that does not need to have any description.")  tr("Invite", "An invitation button for inviting relatives to the site") |

The DESCRIPTION is not mandatory, but it should be used in cases when the label alone is not sufficient enough to determine the meaning of the sentence being translated. For example, when the translators use the bulk translation mode and look at the list of strings, they won't be able to tell what the "Invite" by itself means. In that case DESCRIPTION will be very helpful. The translations engine works by creating a hash value from the text and description of a phrase. That means that even a slight change to the original text or description will cause your string to be counted as a completely new one. So make sure you start off with a complete description you won't have to clarify later.

Here is an example when the same word will have two different meanings based on the context:

|  |
| --- |
| tr("Invite", "An invitation button for inviting relatives to the site")  tr("Invite", "An invitation received by the user") |

In Russian language those two Invites will translate to "Пригласить" as a verbal form and "Приглашение" as a noun, respectively.

**Rule:** Any text under 20 characters should have a description. So when our users use the bulk translation mode and will not see the text in the context of our application, they should get the meaning from the description.

## Translations With Tokens

Translations with tokens:

|  |
| --- |
| tr("Hello {name}!", "Welcome message to the user", :name => own\_profile.name) |

Tokens in the Geni translations engine are defined as any string surrounded by curly brackets inside of a label. So anything of this form {TOKEN\_NAME\_GOES\_HERE} is considered a token.

When the above label gets translated the {name} will be substituted with the value pointed by the :name symbol. And if own\_profile.name results in "Michael", the final translated string will be "Hello Michael!"

Another example of simple token usage:

|  |
| --- |
| tr("Hello {name}!", "Welcome message to the user", :name => link\_to(own\_profile.name, own\_profile.path)) |

This is almost the same example, but in this case the {name} would be substituted with a link to a profile.

**Rule:** All of the HTML elements submitted by the user in the translation will be escaped by the engine. So we should never include any HTML elements in the LABEL.

## Nested Translations With Tokens

But what if a sentence comprises of a simple text and a link? Well, translations engine handles it very easily.

|  |
| --- |
| tr("If you would like to get more information about privacy changes, visit our {more\_info\_link}.", "",  :more\_info\_link => link\_to(tr("updated privacy page", "Link to the updated privacy page"), link\_url)) |

This is an example of nested translation. Notice that the tr function is used twice in this example: to translate the text string and to translate the text inside of the link.

**Rule**: You can nest the translation as many times as you would like, but lets try to keep it readable. Two levels of nesting should be sufficient enough. Otherwise make it a variable and pass it as a token, as in the example below.

|  |
| --- |
| link\_text = link\_to(tr("updated privacy page", "Link to the updated privacy page"), link\_url)  tr("If you would like to get more information about privacy changes, visit our {more\_info\_link}.", "", :more\_info\_link => link\_text) |

Notice that i provided "" as a description. In this case the first part of the sentence is clear enough, so you can provide the second parameter as nil or "".

Make sure you **don't do this**:

|  |
| --- |
| link\_text = link\_to(tr("updated privacy page", "Link to the updated privacy page"), link\_url)  tr("If you would like to get more information about privacy changes, visit our #{link\_text}.") |

This will result in users seeing the html link in the translation key instead of the link token.

## Gender Dependent Tokens

The "Hello Michael!" example was great, but what if the translation depends on the gender of the user, like in the case of "Dear Michael," The word "Dear" is an adjective and it will change in many languages based on the gender of the user. The engine solves this problem by using the following approach:

|  |
| --- |
| tr("Dear {user.name},", "Reference to a user in a heading of an email", :user => page\_profile) |

There are a couple of new things here. The first thing is obvious - our translations engine allows for method invocation on token objects by using the dot notation! The second part is hidden. There is a number of tokens names that are considered special. Among them: user, profile, actor, target. Any time the engine sees a token which ends in one of these string, it knows that the token may be gender dependent and it offers an option to the user to translate the label for each gender type. So the following tokens will receive a special treatment: friend\_user, email\_profile, friend\_profile, etc...

Not all languages will have to be gender specific in every example, so it is up to the user to specify whether the translation for a specific language needs gender dependency. The case above is gender specific in Russian. So when the user chooses the gender option for user, the engine will pre-create 3 translations, where each translation will have a different context:

|  |
| --- |
| "Dear {user.name}!"  Context: user is a male  "Dear {user.name}!"  Context: user is a female  "Dear {user.name}!"  Context: user has an unknown gender |

User will then have to translate each one of those labels based on the label context. For example, the above three sentences will be translated into Russian as follows:

|  |
| --- |
| "Дорогой {user.name}!"  Context: user is a male  "Дорогая {user.name}!"  Context: user is a female  "Дорогие {user.name}!"  Context: user has an unknown gender |

I am using a "respected" form for the case when we don't know the gender of the user. Well, if users don't like my approach they will vote the last option down and provide an alternative. That's what crowd-sourcing is for!

When the translation engine evaluates/substitutes those translation keys, it will be smart to check the gender of the token value (in this case it is page\_profile) and choose the right translation based on the context rules. The translations engine is AWESOME!

## Viewing User Token and Multiple Gender Dependent Tokens

But what if we have the following complex case when we are trying to translate a predefined email that the current user (own\_profile) is trying to send to his/her family member or a friend (page\_profile)?!

|  |
| --- |
| tr("Dear {user.name}, I would like to invite you to join Geni!", "An invitation email heading", :user => page\_profile) |

Even though this example looks like it uses one token, in reality there are two tokens here. The first one is obvious, it is the user we are sending the message to (or page\_profile). We would have to translate this string correctly as the word "Dear" will depend on the gender of page\_profile. But what about the other one?! "What other one?" - you may say. Well, in Russian language the sentence "I would like to invite you to join Geni" will actually be translated based on the gender of the "I" or the viewing user. The viewing\_user token is special. It is actually so special that it is included for every translation. Pretty much every single label, whether it has tokens or not, can be depended on the gender of the viewing\_user. So when a translator attempts to translate the above sentence, he/she will actually have to translate not one, and not even three, but actually nine sentences. Below is the list of those sentences:

|  |
| --- |
| "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a male and viewing\_user is a male  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a female and viewing\_user is a male  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user has an unknown gender and viewing\_user is a male  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a male and viewing\_user is a female  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a female and viewing\_user is a female  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user has an unknown gender and viewing\_user is a female  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a male and viewing\_user has an unknown gender  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user is a female and viewing\_user has an unknown gender  "Dear {user.name}, I would like to invite you to join Geni!"  Context: user has an unknown gender and viewing\_user has an unknown gender  Here is what this would look like in Russian:  "Дорогой {user.name}, я бы хотел пригласить вас на сайт Geni!"  Context: user is a male and viewing\_user is a male  "Дорогая {user.name}, я бы хотела пригласить вас на сайт Geni!"  Context: user is a female and viewing\_user is a male  .  .  . |

Well, i hope you got the point and there is no need for me to translate all nine sentences! Probably someone who can read Russian would say that there is another way to translate the sentence without having to rely on the gender of the viewing user. Well, great, that's why you have an option to translate it yourself and then we will let other users decide which one is better...! The one that gets more votes wins.

The good news is that the rules engine of the translation engine will choose the right translation based on the context, which can be comprised of any number of tokens. The viewing\_user token is always passed to the translations engine and if a gender dependency is specified for the viewing\_user token, the appropriate context will be used. Another good news is that we are using crowd-sourced approach and translating these labels shouldn't take much time. Well, you were expecting that i would mention some bad news now... but there aren't any, actually!

Alright, if that wasn't hard enough already, lets make it even harder:

|  |
| --- |
| tr("Dear {user.name}, I would like you to meet my friend, {friend.name}.", "Friend recommendation email", :user => page\_profile, :friend => friend\_profile) |

You probably guessed it right... We now have to deal with 27 combinations! (3 \* 3 \* 3 = 27) Why? Because in Russian, for example, the word "Dear" will depend on the gender of page\_profile, the words "I would like you" will depend on the gender of the viewing\_user hidden token and the word "my friend" will depend on the gender of friend. In Russian, the word "friend" changes based on wether the friend is a male of a female. I will not write out all of those combinations, but i am sure you can figure them out yourself.

## Translation Options

Up until now, we haven't looked at what this OPTIONS parameter is for. Well, let's look at the following simple example again:

|  |
| --- |
| tr("My friend, {user.name}...", "Reference to a user in a heading of an email", :user => page\_profile) |

This is great that we can invoke methods on objects, assuming that the method is actually defined (if not, it will blow up). But, what if we want to make the {user.name} not just a string, but a link to the profile page of the user. Nested translations in this case won't work and putting the link into the token definition is also not going to work because we expect the user token to contain an object that has a gender. Well, this is where options will come handy. See the example below:

|  |
| --- |
| tr("My friend, {user}...", "Reference to a user in a heading of an email", {:user => page\_profile}, {"{user}" => display\_profile(page\_profile)} ) |

If you haven't noticed, we are actually killing two birds with one stone here: first of all, we are making the label token much shorter and we no longer need the method invocation on the object. And second, and most importantly, we are giving the designer/programмer full control over what the token value should look like at the time of the translation evaluation/substitution. When the translation engine sees that options are provided, it will use whatever value you provide in the options for the value to be substituted in the original label. So the label is a representation of a tokenized string, the token is used for the context of the translation and the options are used as the final substitution for the value in the label.

Alternative approach is to pass the display options as the second value of the token array:

|  |
| --- |
| tr("My friend, {user}...", "Reference to a user in a heading of an email", {:user => [page\_profile, display\_profile(page\_profile)]}) |

This feature was added at a later point, and once all of the internationalized strings have been switched to use the array approach and we do not make the case for keeping the OPTIONS parameter, the OPTIONS will be deprecated. For now, i support both of the above approaches.

That's pretty much it for the all four parameters and what they are used for. Lets move on to some other cases and examples...

## Number Dependent Tokens And Hidden Tokens

The gender depended tokens are actually the easy case. Hah?! Well, there are only 3 options for the gender dependent tokens: male, female or undefined. But when we deal with numbers, we have an unlimited number of cases. Check out the following example:

|  |
| --- |
| tr("{message\_count} messages", "Inbox message count label", : message\_count => count) |

The variable count can be anywhere from -infinity to +infinity and if we were to create a case for each value of the variable "count" then we would be translating our site forever. But the rules engine of translation engine is sophisticated enough to take care of this scenario as well. Here are the simple context rules for English language and a few other languages:

|  |
| --- |
| "{message\_count} messages"  Context: message\_count is 1  "{message\_count} messages"  Context: message\_count is not 1 |

There are a couple of issues here related to the word "messages" that we will look into in a bit, but other than that, the context rules seem pretty simple and there are only 2 translations to take care of. Well, it may be so for the English language, but not so much for Russian and other languages. The Russian context rules will look like the following:

|  |
| --- |
| "{message\_count} messages"  Context: message\_count ends in 1 and is not 11  "{message\_count} messages"  Context: message\_count ends in 2, 3, 4 and is not 12, 13, 14  "{message\_count} messages"  Context: message\_count ends in 0, 5, 6, 7, 8, 9 or is 11, 12, 13, 14 |

The translation engine can handle any type of a numeric rule like that and even allows advanced users to build those rules in a friendly user interface. Similarly to how the engine identifies user, profile and others as gender based tokens, the translation engine knows that any token that ends in "count", "num", "age", "hours", "minutes", "years", "seconds" can be number dependent. So be aware that when you name a token "message\_count" or "message\_num" or "friends\_count" the engine will offer the translator an option to make the translation dependent on the numerical value of the token.

The numeric rules are comprised of one or two parts, where each part is comprised of a logical operator and values. Supported operators are "is", "is not", "ends in" and "does not end in" and values are just lists of numbers. The two parts of the rules can be combined together using "and" or "or" operator. The rules are stored in the engine in the following format:

For English-like languages:

|  |
| --- |
| {:type => :number, :complex => false, :name1 => :is, :value1 => [1]},  {:type => :number, :complex => false, :name1 => :is\_not, :value1 => [1]}, |

For Russian:

|  |
| --- |
| {:type => :number, :complex => true, :name1 => :ends\_in, :value1 => [1], :operator => :and, :name2 => :is\_not, :value2 => [11]},  {:type => :number, :complex => true, :name1 => :ends\_in, :value1 => [2,3,4], :operator => :and, :name2 => :is\_not, :value2 => [12,13,14]},  {:type => :number, :complex => true, :name1 => :ends\_in, :value1 => [0,5,6,7,8,9], :operator => :or, :name2 => :is, :value2 => [11,12,13,14]}, |

Admin users can create or change those rules for any languages.

When a user tries to translate the above example, the user will have an option to make the translation string dependent on the value of the {message\_count} token and the rules engine will present the user with the combinations of the context rules based on the rules previously provided by the admin users for that language. The rest is very similar to the gender based rules.

If you were wondering, the gender based rules are also stored in the rules engine using the following format:

|  |
| --- |
| {:type => :gender, :name1 => :male},  {:type => :gender, :name1 => :female},  {:type => :gender, :name1 => :unknown} |

But in the case of the gender, the rules are global and language independent. And unless we run into a language that has additional genders, we won't have to change those at all.

As i mentioned earlier, i didn't like something about the word "messages" in the label of the example. Let's look at the example one more time:

|  |
| --- |
| tr("{message\_count} messages", "Inbox message count label", : message\_count => count) |

First of all, this example is grammatically incorrect even in English. What if the count is 1? Then we get a string "1 messages" in English. The quick fix would be to use pluralization forms in English, like the following:

|  |
| --- |
| tr("{message\_count} #{pluralize\_word(count, "message")}", "Inbox message count label", : message\_count => count) |

That will fix the issue in English, but it would introduce two unique keys for translation: "{message\_count} message" and "{message\_count} messages" and our users will have to translate both - that's a duplicated effort for them to translate and for us to maintain. So this is not a good solution. This is when we use special hidden tokens.

A hidden token is a token that starts with "\_". The following is an example that does work and fixes the above issue:

|  |
| --- |
| tr("{message\_count} {\_messages}", "Inbox message count label", : message\_count => count, :\_messages => pluralize\_word(count, "message")) |

Hidden tokens act as real tokens at the time of evaluation of the English language, but they are not presented as tokens to the users. When the user will choose to translate the above label, he/she will be presented with the following translation label:

|  |
| --- |
| Original Phrase:  "{message\_count} messages" |

And the user will not be given {\_messages} token as an option to use in the translation.

Rule: When you use hidden tokens, make sure you use a readable and meaningful string, as the string (minus the brackets and the first underscore) will be used as the sanitized label to be translated by the user.

Another example of hidden token would be:

|  |
| --- |
| tr("{user.name} commented on {\_his\_her} document", "Document comment newsfeed story", : user => own\_profile, : \_his\_her => own\_profile.his\_her) |

The above example will use the correct for for English, but prompt the user to translate the following label:

|  |
| --- |
| Original Phrase:  "{user.name} commented on his/her document" |

You noticed it correctly {\_his\_her} magically turned into "his/her". It's actually not magic, but a thoughtful deliberate implementation of the hidden tokens rules. When a token rule {\_his\_her} is sanitized, it removes the curly brackets and the first underscore. Then it replaces any double underscores with spaces and single underscores with "/". This way you can do things like: {\_his\_her} resulting in "his/her" and {\_posted \_ \_ items} resulting in "posted items".

You are probably wondering if you will get into troubles if you have something like this:

|  |
| --- |
| tr("{hours} {\_hours} ago", "Newsfeed time header", :hours => elapsed\_hours, :\_hours => pluralize\_word(elapsed\_hours, "hour")) |

Didn't i mention earlier that anything that ends with the word "hours" will receive a special treatment by offering the user to specify dependency on the numeric value of the token {hours}. Well, good that you are paying attention. But, no, this is perfectly fine. Hidden tokens are not even on the list of tokens that may receive a special treatment. So the above key will be presented to the user as:

|  |
| --- |
| "{hours} hours ago" |

And the user will have only options to make the translation dependent on the token {hours} or the gender of the viewing user. You thought of it, but i thought of it too! :)

## Putting It All Together

So to summarize it all together, take a look at the following funky, but common and surprisingly simple example:

|  |
| --- |
| tr("Dear {user}, you have {message\_count} {\_messages} in your inbox.", "Inbox message count label",  {:user => page\_profile, : message\_count => count, :\_messages => pluralize\_word(count, "message")},  {"{user}" => display\_profile(page\_profile)}) |

The above example has a dependency on gender of the user and message count. It also makes the user name a link to the user profile. If we were to translate that string into Russian, we would have to provide a total of 9 translations:

|  |
| --- |
| "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a male and message\_count ends in 1 and is not 11  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a female and message\_count ends in 1 and is not 11  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user has an unknown gender and message\_count ends in 1 and is not 11  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a male and message\_count ends in 2, 3, 4 and is not 12, 13, 14  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a female and message\_count ends in 2, 3, 4 and is not 12, 13, 14  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user has an unknown gender and message\_count ends in 2, 3, 4 and is not 12, 13, 14  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a male and message\_count ends in 0, 5, 6, 7, 8, 9 or is 11, 12, 13, 14  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user is a female and message\_count ends in 0, 5, 6, 7, 8, 9 or is 11, 12, 13, 14  "Dear {user}, you have {message\_count} messages in your inbox."  Context: user has an unknown gender and message\_count ends in 0, 5, 6, 7, 8, 9 or is 11, 12, 13, 14 |

## Lambda Tokens

When you thought it couldn't get any more complicated, it did... Initially this was considered a limitation, but it no longer is and now it is a very useful new feature. So what if you have the following requirement:

We need to build the following newsfeed story:

|  |
| --- |
| “Michael commented on his document. “ |

Where "Michael" is a link to a profile page and "his document" is a link to the document. Consider the following implementation:

|  |
| --- |
| tr("{actor} commented on {\_his\_her} document", "Document comment newsfeed story",  {:actor => actor, :\_his\_her => actor.his\_her},  {"{actor}" => display\_actor(opts)}) |

The above example is a good start, but it doesn't have the "document" linked to the document.

|  |
| --- |
| tr("{actor} commented on {\_his\_her} {document}", "Document comment newsfeed story",  {:actor => actor, :\_his\_her => actor.his\_her},  {"{actor}" => display\_actor(opts), "{document}" => link\_to(tr("document"), document\_url))}) |

This is a better version, as the "document" is now linked to the document, but the word document is now translated outside of the context of the sentence. It does work in English and Russian, but we may run into limitations if in other languages the word document changes based on the gender of the actor.

|  |
| --- |
| tr("{actor} commented on {document}", "Document comment newsfeed story",  {:actor => actor},  {"{actor}" => display\_actor(opts),  "{document}" => link\_to(tr("{\_his\_her} document", "How are we supposed to explain this?", :\_his\_her => actor.his\_her), document\_url))}) |

And, finally, "his document" is now linked, but how are we supposed to explain "his/her document" to the user.... and now we have to deal with a whole bunch of "his/her document, photo, video, etc..." strings that are out of context and make no sense in the bulk translation mode...

Well, to solve this problem, thanks to Justin, we introduced the lambda tokens. Consider the following example:

|  |
| --- |
| tr("{user} commented on [link: {\_his\_her} document].", "Document comment newsfeed story",  :user => [own\_profile, link\_to(own\_profile.name, own\_profile.path)],  :\_his\_her => own\_profile.his\_her, :link => lambda{|str| return link\_to(str, link\_to\_document)}) |

Lambda is a slightly different token type. It starts and ends with [ and ] instead of the curly brackets. The first part of the token is the name of the token and the second part is the translatable portion of the expression that will be used during the lambda evaluation.

When user is prompted to translate this string, he/she will see the following key:

|  |
| --- |
| {user} commented on [link: his/her document]. |

As you can see the hidden token rules (as well as any other token rules) still apply even inside the lambda token.

Once the user translates the outside part and the inside part of the lambda token, the token will be correctly evaluated and the [link: {\_his\_her} document] will be replaced with the correct translated value passed through the lambda function defined by the designer/developer.

The translated value for the Russian language will be:

|  |
| --- |
| {user} оставил сообщение на [link: своем документе].  Context: user is a male |

If the translator keeps the lambda value inside the translation key label, it will be correctly substituted with the link to the document with the label of "своем документе".

## Extended Syntax Support

Notice a new/alternative syntax to the options parameter. You now can pass token values as arrays, where the first value must be an object (for gender or count dependency, if any) and the second value is the display option.

|  |
| --- |
| tr("{user} commented on [link: {\_his\_her} document].", "Document comment newsfeed story",  :user => [own\_profile, link\_to(own\_profile.name, own\_profile.path)],  :\_his\_her => own\_profile.his\_her, :link => lambda{|str| return link\_to(str, link\_to\_document)}) |

In the old approach, this would be implemented as:

|  |
| --- |
| tr("{user} commented on [link: {\_his\_her} document].", "Document comment newsfeed story",  :user => own\_profile, :\_his\_her => own\_profile.his\_her, :link => lambda{|str| return link\_to(str, link\_to\_document)}  {"{user}" => link\_to(own\_profile.name, own\_profile.path)}  ) |

The two example above are equivalent. I like both, Justin likes the first one better. You decide which one to use, i support both. ;)

Date/Time Localization

## Strftime Syntax

Examples:

|  |
| --- |
| t = Time.now  t.strftime("Printed on %m/%d/%Y") #=> "Printed on 04/09/2003"  t.strftime("at %I:%M%p") #=> "at 08:56AM" |

The following is a full list of all supported tokens:

|  |
| --- |
| %a - The abbreviated weekday name (``Sun'')  %A - The full weekday name (``Sunday'')  %b - The abbreviated month name (``Jan'')  %B - The full month name (``January'')  %c - The preferred local date and time representation  %d - Day of the month (01..31)  %H - Hour of the day, 24-hour clock (00..23)  %I - Hour of the day, 12-hour clock (01..12)  %j - Day of the year (001..366)  %m - Month of the year (01..12)  %M - Minute of the hour (00..59)  %p - Meridian indicator (``AM'' or ``PM'')  %S - Second of the minute (00..60)  %U - Week number of the current year, starting with the first Sunday as the first day of the first week (00..53)  %W - Week number of the current year, starting with the first Monday as the first day of the first week (00..53)  %w - Day of the week (Sunday is 0, 0..6)  %x - Preferred representation for the date alone, no time  %X - Preferred representation for the time alone, no date  %y - Year without a century (00..99)  %Y - Year with century  %Z - Time zone name  %% - Literal ``%'' character |

The following is a list of all the default time/date formats that the translations engine comes with out of the box:

|  |
| --- |
| default: '%m/%d/%Y', # 07/4/2008  short\_numeric: '%m/%d', # 07/4  short\_numeric\_year: '%m/%d/%y', # 07/4/08  long\_numeric: '%m/%d/%Y', # 07/4/2008  verbose: '%A, %B %d, %Y', # Friday, July 4, 2008  monthname: '%B %d', # July 4  monthname\_year: '%B %d, %Y', # July 4, 2008  monthname\_abbr: '%b %d', # Jul 4  monthname\_abbr\_year: '%b %d, %Y', # Jul 4, 2008  date\_time: '%m/%d/%Y at %H:%M', # 07/4/2008 at 10:15 |

## Language Localization

# Appendix

## Supported Languages

|  |  |  |
| --- | --- | --- |
| Abkhazian  Afrikaans  Albanian  Arabic  Armenian  Aymara  Azerbaijani  Bashkir  Basque  Belarusian  Bosnian  Bulgarian  Catalan  Cherokee  Chinese (Hong Kong)  Chinese (Simplified)  Chinese (Traditional)  Chuvash  Croatian  Czech  Danish  Dutch  English  English (Canadian)  English (Scottish)  English (UK)  Esperanto  Estonian  Faroese  Fijian  Filipino  Finnish  French  French (Canadian)  Galician | Georgian  German  Greek  Guaraní  Gujarati  Haitian Creole  Hebrew  Hindi  Hungarian  Icelandic  Indonesian  Irish  Italian  Japanese  Javanese  Kashmiri  Kazakh  Kirghiz  Korean  Kurdish  Latin  Latvian  Limburgish  Lithuanian  Malagasy  Malay  Maltese  Marathi  Mongolian  Navajo  Nepali  North Sami  Norwegian  Panjabi | Pashto  Persian  Polish  Portuguese (Brazil)  Portuguese (Portugal)  Quechua  Romanian  Romansh  Russian  Sanskrit  Serbian  Slovak  Slovenian  Spanish (Chile)  Spanish (Colombia)  Spanish (Mexico)  Spanish (Spain)  Spanish (Venezuela)  Swahili  Swedish  Tahitian  Tajik  Tamil  Tatar  Thai  Tibetan  Turkish  Turkmen  Ukranian  Urdu  Uzbek  Vietnamese  Welsh  Xhosa |