Web application localization with translatewiki.net
- Lessons learnt and reflections from translating EntryScape

The following report is the result of localizing the web application EntryScape (entryscape.com) and making it work with translatewiki.net. The first part provides a short technology background of EntryScape to give a better understanding of the challenges faced. The second part lists development considerations with respect to localization. The third part goes through some general lessons learnt with respect to localization of web applications similar to EntryScape. The final part provides reflections regarding the integration with translatewiki.net.

1. Introduction to the technology

EntryScape is a Single Page Application. That means that it is loaded as a single web page that fetches data over REST services and maintains state without ever reloading or forcing the user to switch to another web page. The user interface is constructed from a range of templates that are provided upon inital page load (some lazy loading may take place). The effect is that all localization happens in the browser by combining the templates with the loaded localization bundles.
EntryScape uses an AMD compatible loader (Dojo) for loading JavaScript and other modules such as localization bundles, templates, configurations etc. To simplify the localization a specific loader plugin, i18n is provided to loading and selecting the right translation for the current locale. EntryScape originally used the Dojo implementation of this loader plugin. Unfortunately this plugin does not handle plurals, grammar etc. Hence, to solve this an extended loader plugin, di18n, was developed to support plurals (it can easily be further extended to support grammar, gender etc. as well). The di18n loader plugin also integrates nicely with the Dojo templating mechanism. Translatewiki.net has also been extended with a specific File Format Support class, AmdFFS, that support the AMD i18n way of handling localization bundles.

2. Development issues related to localization

When developing single page applications many things work differently. For example the initialization process where the application bootstraps itself by loading data over REST services. This means that it is not clear upon page load (in general) who the user is and which language he prefers. The same is of course true if the user has not signed in or even signed up yet. In such cases, the application must build the user interface based on the browser’s default language and first at a later stage change to a more suitable language. Now, if we hold on to the requirement of not reloading the page, the consequence is that the UI components must be able to update themselves upon locale change. Either by re-rendering their part of the DOM from scratch or by using a more precise approach which is updating just the parts of the DOM that contain the localized strings. In both cases this has an effect on the choice of the rendering mechanism, e.g. the choice of the client side templating language.

It is crucial to have good tooling support for the actual localization step, that is, handling the loading process of localization bundles, choosing the right translation for the current locale, as well as supporting plurals, grammar and other more advanced features. The di18n loader plugin mentioned above provides this tooling support.

If a templating mechanism is used that integrates well with the localization mechanism it saves a lot of time. Especially if it is possible to insert the localization keys directly in the template and to make the localization step part of the rendering step. The opposite, which involves a much larger effort, requires to write code in the controller layer that translates the localization keys to points in the DOM. To have this tooling support even for keys that rely on plurals is somewhat more complicated as it requires a mechanism to declaratively specify how a localization key depends on a specific variable for deciding the right plural from the template.

3. Lessons learnt regarding localization

Reusing keys

As a developer there is an ever ongoing quest to minimize the amount of duplicated code. Naturally, this behaviour spills over when introducing keys to internationalize your application. A typical example is to define generic keys that provide text for confirming or aborting dialogs. However, reusing keys has the following unfortunate consequences:

1. Less flexibility, sooner or later, e.g. when a UI evolves, you might want to change generic text into something more specific in one place but not in another.
2. False generalizations, text that appears to be alike in one language may be more appropriate to have different expressions in another.
3. Documentation of keys becomes harder, should all places a key is reused at be listed?
4. Harder for translators, it is less obvious for those doing the translation to oversee the consequences. Also, if the key is reused in a new place, will translators need to revise the translation to see if it still is valid?
5. More difficult to do refactoring as you have to keep track of when a key is still in use.

Recommendation: *Do not reuse keys.*

Organizing your keys

Most applications grow over time and incorporate multiple views with localization keys in them. Consequently an important question is whether all keys should be kept together or split into multiple modules.

Reasons to use a single module	**Counter argument**
Limits the amount of requests needed	In production, files are often concatenated
Reuse of keys is easier	Reuse of keys is bad, see above

Provides a good overview	Tools like translatewiki.net provide an overview
Hard to find a consistent modularization	Most code is organized into modules for reasons of maintainability, the same should apply to the corresponding keys
Reasons to use several modules	**Counter argument**
Shorter key names and less risk of conflicts	Complicates refactoring
Lazy loading in large applications	Few if any of well known build processes supports creating several “layers” of translation strings
Better match with modularized code	Modules may not make sense to translators

Recommendation: *Organize your keys and translations into bundles that coincide with major modules in your application. Maintain this mirroring when refactoring to make sure keys are deprecated and not just kept unused.*

Naming of keys

One often underestimates that a problematic area within software engineering is finding good names at variables, classes, modules as well as higher level concepts and architectural elements. Unfortunately, introducing localization into your application introduces an additional category of naming problems. However, a good naming scheme can reduce, if not eliminate, this problem significantly.

A common principle is to use camel case for variable names. However, for translatewiki.net camel case notation may be interpreted as links and is best avoided. Using dots and hyphens as suggested in the translatewiki.net documentation is bad in a JavaScript environment where this would prohibit the convenient access to localization strings from object hashes. If there still is a need to mark individual words, one solution is to focus on the use of underscore as a separating character instead of camel case.

In a user interface there might be several keys that correspond to something that is represented as a single value in the data model. Distinguishing by appending the HTML element name might solve the problem, however, this approach is a bit fragile as the expression using certain elements often change during development and refactoring (for example changing from a div to a span). Another approach is to focus on the user interface character that also influences the key’s value, e.g. something is a label, link, button, (input-)field or text. In addition, many elements may have additional attributes such as tooltips (title attribute) and placeholders. In this case it is a good idea to append ‘_title’ and ‘_placeholder’ correspondingly.

Recommendation: *Try to find key names that are meaningful and consistent and express them using camel case ending with the nature of the UI-element, e.g. label, link, button, field, text etc. For keys that correspond to attributes append '_attrName'. E.g. in a search dialog for books it would make sense to have a key 'search_book_field_placeholder' for the placeholder in the input field and the search button's tooltip via a key named 'search_book_button_tooltip'.*

Translations and limited space

A common problem is that you do not know in advance the length of the text when it is translated into another language. Hence, you can never be completely sure that the max-width you decided upon is enough to handle all possible translations. A workaround in these cases is to accept abbreviations and instead support the title attribute to provide the complete or more verbose wording. If this problem is ignored the text will be either truncated or flow outside of the confined space which will look bad and be hard to read.

Recommendation: *Inform translators in the description of the key that the text has restricted space and which key is used for the corresponding title attribute. Encourage the translators to stay within a given maximum of characters or to abbreviate. Note that it is better to abbreviate and provide a title rather than to rephrase into something shorter that is an inferior alternative or not consistent with the rest of the application.*

4. Reflections regarding the integration with translatewiki.net

The procedure to getting a project set up in translatewiki.net is unfortunately not entirely trivial. One reason may be it's roots as a solution for translating MediaWiki rather than translating generic Open Source projects. In the following subchapters we outline problems and things that are easy to miss.

Translatewiki.net's documentation

The documentation is divided into several sections, and it is unfortunately easy to get lost among all the subpages. The main entry points are:

About page:
This page contains a lot of background information that is not necessarily vital, but it provides a context that may make it easier to better understand why translatewiki.net works the way it does. One vital piece of information is who the main persons behind translatewiki.net are.

Main documentation:
Targets mainly translators and provides guidance on how to carry out translations with a focus on MediaWiki translation. There are many links to more detailed pages, including how to set up and install translatewiki.net for yourself.

Localization for developers:
This page is the most important for understanding what you need to do to get your project added to translatewiki.net. However, some information, such as how to write the YAML files are

better described in a page about YAML and message groups linked only from the main documentation.

Missing information regarding the procedure to add a project

When you have read through a lot of the documentation there are still a few things that are left unsaid:

- How do you create add a project page? Do you ask someone or do you do it yourself, if so, how do you get the rights to do it (the wiki is not open)?
- Do you need to create a patch to the translatewiki.net git repository as indicated in the repository management page or is that information targeted towards translatewiki.net maintainers?
- Who do you contact in the chat, the about page mentions the community manager Siebrand (siebrand) and Niklas Laxström (nikerabbit). What is the correct approach?
- The procedure to initiate the process is not documented, i.e., to create a ticket in phabricator and what information it should contain.

Sync and export with localization repository

Translatewiki.net has a manual sync and export process to keep in phase with an external repository for the localization bundles. This process is unfortunately somewhat unclear, it is stated that it depends on the activity of the project and its release schedule. This is unsatisfying as a fully automated approach (e.g. using post-commit hooks) would reduce the round trip from when a text is translated to when it is visible in the web application. It can be argued that a translation is better evaluated when seen in the final setting. Hence, this kind of long round trips makes a staging environment with translated strings impractical or even impossible. Unfortunately, this may affect the quality of the translations, or at least slow down the process towards reaching good quality translations.