Next Generation Internet

Best practices for localization and internationalization

This documentation offers best practices and guidance for localization and internationalization of software, multimedia content, and websites. The guidance is aimed at Free and Open Source Software and Libre/Open Hardware projects funded by the Next Generation Internet research and development initiative. This guide may also help readers generally interested in localization and internationalization.

What is localization? What is Internationalization?

We live in a connected world. There is a proliferation of digital content, software, web apps, and mobile apps. The concepts of localization and internationalization are becoming crucial in such a globalized world.

Localization is adapting computer software, video games, multimedia content, and websites for use in a new language. One can think of localization as the translation of the text and other changes required to make it function properly in a specific language or place. Localization is often referred to as l10n (as in: '`L`', followed by ten more letters, and then '`N`').

Internationalization is a set of engineering activities to make the software adaptable to other languages and cultures. Thus, internationalization enables localization. Internationalization makes the product translatable, and also makes it aware of language-specific and location-specific conventions that differ. Conventions differ in the formats for time, date, currency, numbers, and more. One can think of internationalization as the work that will make localization into many languages easy. Internationalization is often referred to as i18n (as in: '`I`', followed by eighteen more letters, and then '`N`').

This document provides a highly summarized overview of the main concepts and steps to take, and aims to provide guidance for the process.

What do we need to do?

Step 1: Prepare the project

- Consult the internationalization documentation for the programming language, platform, and/or build system. Some relevant best practices might be covered there, for example, default libraries for automated testing using pseudolocalization.
- Make the necessary changes to enable a locale. Locales may need to be enabled implicitly, for example, from an HTTP `Accept-Lang` header or from environment variables. Locales may need to be enabled as a result of user action.
- Replace all translatable text in the code with localization calls prior to display. For example, replace the text string `"Keyboard shortcuts"` with a call to `getMessage(locale, "Keyboard shortcuts")` or if GNU Gettext is used, then with a call to `_("Keyboard shortcuts")`.
- Ensure all texts from libraries or third-party components that could be presented to the user are localized.
- Ensure the code works with incomplete translations, not only 100% complete ones. Many translation libraries handle missing keys gracefully with automatic fall-back.
- Automated tests may need to be enhanced to accommodate translations.

If done properly, adding support for a language may become simply adding the translation file(s) for that language.

Step 2: Gather source text

- Ensure that all localizable material is together in one place. Depending on the platform, the source text to translate might already be isolated in certain source text templates, resource bundles, or message catalogs. An ICU/CLDR system will likely expect text key-value trees, whereas Gettext will use a `.pot` file extracted with `xgettext`.
- Review the text for quality, such as spelling, consistency, etc.
- If translations are managed in a dedicated system, upload this new version.
- Consider announcing a period before a release with no text changes so that translators have enough time. This is sometimes called a "string freeze" period.

Step 3: Update existing translations

If there are existing translations from a previous version, they need to be updated to the new source material. Many systems have tools to assist in this process. For i18next projects, `i18next-scanner` and `i18next-locales-sync`. For Mozilla Fluent projects, `fluent-merge` and `compare-locales`. For Gettext projects, `msgmerge`. This step might be performed automatically by a build system or online translation management tool.

Step 4: Open the process to localizers

- Provide instructions for the localizers. Be clear about what is expected from them, including deadlines.
- Developers or localizers might prioritize certain files or sections over others.
- Try not to limit what tools localizers can use. Localizers might have a set of tools that they prefer to use. There are standalone tools and web-based tools for translation.
- Document how localizers can test localized versions. It is highly beneficial if localizers can easily test their translations themselves.
- Localizers translate, review, and resubmit the work.

Step 5: Test and integrate translations

- Translations should be committed to the version control system, and integrated with the build system.
- Ensure that the software builds and works with each translation.

Step 6: User testing

- Reach out to existing users for each language. Real users are often ideal testers.
- If required, provide special builds suitable for testing new translations.
- Keep in mind that a localized version could expose bugs in the code.

When updating the software, collaborators might go back to the translation step 3, or the whole process could be repeated as part of the normal software development life-cycle for each version.

General Best Practices

- Get started with localization process during the early stage of development.
- Make internationalization part of the project's coding guidelines.
- Include checking for internationalization issues as part of the code review process.
- Use Unicode for text encoding.
- Support multilingual user input, do not assume English.
- Ensure that sorted items presented to the user are ordered as expected. For example, `ä` sorts after `z` in Unicode default sorting, but in the German alphabet `ä` sorts immediately after `a`.
- Allow for punctuation differences. For example, in Spanish a sentence may have punctuation at both the beginning and the end.
- Design the system to support dialects. For example, in French the comma is used as the decimal separator in France and the dot is used as the decimal separator in Switzerland.
- Allow for differences in capitalization. For example, German and English have different capitalization rules for adjectives and nouns.
- Prefer presenting dates as unambiguously as possible for the date formats in the locale. For example, Javascript's `Intl.DateTimeFormat` with `dateStyle:"medium"` uses four-digit years and short text for the month.
- Use the localization library when concatenating numbers with text for correct pluralization. For example, the Polish language has more pluralization categories than English.
- Be especially careful if the code manipulates text.
- Where possible, correctly annotate the language. For example, with the `lang="xx"` attribute in HTML/XML.
- Consider common localization practices in the outside world. For example, Wikipedia is highly multilingual.
- Leave ample space in the user interface elements to allow text expansion. A sentence may have different lengths in each language to which it is translated.
- Keep right-to-left (RTL) languages in mind. Your platform (Qt, GTK, Web, etc.) should have best practices to support such languages.
- Be aware of cultural differences. Icons, images, and even color can represent different meanings in different cultures.
- Avoid text in images. Plain text is much easier to manage and localize than images with embedded text.

Further Information

- Wikipedia on language localization: wikipedia.org/wiki/Language_localisation
- Wikipedia on internationalization: wikipedia.org/wiki/Internationalization_and_localization
- W3C on ECMAScript internationalization: Guide to the ECMAScript Internationalization API
- Mozilla JavaScript reference for built-in internationalization: Intl
- Unicode Common Locale Data Repository: Unicode CLDR
- International Components for Unicode: ICU-TC
- GNU Gettext (internationalization and localization system) manual: gnu.org/software/gettext/manual/
- Coloring and cultural symbolism: translation-blog.multilizer.com/color-localization-infographics

Acknowledgements

This Guide was primarily created by Translate House, experts in community localization. Translate House has created multiple software solutions for localization. You can check them out here: toolkit.translatehouse.org.

Additional contributions were provided by Commons Caretakers.

The creation of this guide was made possible with financial support from the European Union's Next Generation Internet program, under the aegis of DG Communications Networks, Content and Technology.