Abstract

Managing multi-languages support for a service with autonomous parts can be complex. Having its internal parts be polyglot, and coalesce to end-user’s language only on display is one solution.

This paper discuss a format to store, exchange, and algorithms to consume multi-language strings to this goal.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 5, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Introduction

Managing multi-languages support for a service with different parts, platforms, runtime, back-ends, and front-end; each having theirs proper way to achieve this, without a standardized way to collaborate between them can be complex. Having internal parts be polyglot, and coalesce to end-user’s language only on display is one solution to this complexity.

A common way of storing multi-languages is to split localization into different "packages", splitting strings with the same meaning and formatting apart and away from theirs context. This makes translation and maintenance efforts harder and more error-prone.

To exchange text, one part must also know the end-user’s language beforehand, requiring tight collaboration with other parts; for example a server-side API must know client-side’s language before exchanging a proper response or error, or it might use integer codes.

This paper present a format to 1) store multi-language strings keeping them together in source, and 2) exchange and consume them without requiring prior knowledge of the end-user’s locale.

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The grammatical rules in this document are to be interpreted as described in [RFC5234].
2. Polystring

The most common string format is nul-terminated [C] formatted string, compatible with C-like source and descendant, [ECMA]Script, [Swift], etc.

By extending a (nul-terminated) C-formatted string to carry multiple strings-with-identifiers, it can be multi-language while being stored and exchanged as a single string. The character ‘\’ and the NUL character is used to structure this format, called polystring hereafter.

As consumers of this string must implement polystring support, backward compatibility for consumers of non-poly string is not deemed necessary.

2.1. ABNF grammar

Polystring = *( Identifier String *%x20 ) Base
Identifier = *ANYBUT5C %x5C
String = *%x01-FF %x00
Base = *ANYBUT5C %x00
ANYBUT5C = %x01-5B / %x5D-FF

This grammar represent the string in memory.
Note that it MUST be stored and exchanged as an escaped string.

2.2. Identifier

Identifier SHOULD be [ASCII] encoded, MUST NOT contain NUL nor ‘\’ and MUST end by ‘\’. Although a custom set can be agreed upon by the producer and the consumer, it is recommended for it to be an IETF language [Tag]. It is compared to consumer’s target identifier, up-to Identifier’s own length and matches if equals; so that longer Identifiers should appear first or will be ignored. e.g. "pt-PT" (European Portuguese) before "pt" (Brazilian Portuguese).

2.3. String

String SHOULD be [UTF-8] encoded and MUST NOT contain NUL. It is paired with an Identifier, and terminated by NUL. String of the first matching Identifier is choosed by the consumer for display to the end-user. To improve readability, spaces following String are ignored.

C samples : (both are equivalent)

"fr\Bonjour\0it\Ciao\0"
"fr\Bonjour\0 it\Ciao\0 "

Bouilland Expires August 5, 2019 [Page 3]
2.4. Base

Base is the default string chosen when no Identifier matches. It MUST NOT contain NUL nor '\'. It is equivalent to a non-poly string, so that single-language string can be used as-is. It is recommended for Base to be the "en-US" String.

C samples : (both are valid polystring, with the same Base)

"Hello"
"fr\Bonjour\0 it\Ciao\0 Hello"

Base cannot contain '\', but it is possible to achieve equivalent functionality by using an all-match (zero-length) Identifier. As it always matches in this case, Base MAY be dropped. It MAY also be used to add an internal identifier or describe the usage context.

C samples : (first one drops the unusable Base)

"fr\Avec \ dedans\0 \With \ inside"
"fr\Avec \ dedans\0 \With \ inside\0 \#1234"
"fr\Avec \ dedans\0 \With \ inside\0 a sample"
^ zero-length Identifier

Polystring can be used on multiple lines if the storing source supports it.

sample for C and ECMAscript :
"es\Con cada lengua que se extingue, se borra una imagen del hombre\0 \\fr\Chaque langue qui s’eteint est une image de l’homme qui s’efface\0" For every language that become extinct, an image of man disappears

sample for Swift :
***
es\Con cada lengua que se extingue, se borra una imagen del hombre\0 \\fr\Chaque langue qui s’eteint est une image de l’homme qui s’efface\0" For every language that become extinct, an image of man disappears
***

3. Security consideration

This format is intendend to be read-only, and convey texts only. It is as safe as a standard string, as long as the formating is strictly respected.

Generating polystring dynamically MUST take care to enforce that no '\' nor NUL creeps into respective parts, as it could break the consumer, leading to crashes in worst case scenario.
4. Consumer algorithm

C sample :

const char *localize(const char *text, const char *target)
{
    for (;;) {
        const char *separator = strchr(text, '\');
        if (!separator) return text;
        if (!memcmp(text, target, separator - text))
            return separator + 1;
        text += strlen(text) + 1;
        while (*text == ' ')
            ++text;
    }
}

// usage
const char *lang = getenv("LANG");
puts(localize("fr\Bonjour\0  it\Ciao\0  Hello", lang));

EcmaScript sample :

function localize(text, target) {
    for (;;) {
        var sep = text.indexOf('\');
        if (sep < 0) return text
        var end = text.indexOf('\0');
        if (text.substring(0, sep) == target.substring(0, sep))
            return text.substring(sep + 1, end)

        text = text.substring(end + 1)
        while (text[0] == ' ')
            text = text.substring(1)
    }
}

// usage
var lang = navigator.language || navigator.userAgentLanguage
alert(localize("fr\Bonjour\0  it\Ciao\0  Hello", lang))

Other samples can be found at: http://github.com/blld/polystring
5. IANA Considerations

This document has currently no actions for IANA.

6. References

6.1. Normative References


6.2. Informative References


7. Author’s Address

Aurelien Bouilland
email: aurelien.bouilland@gmail.com