RFC 4648 : The Base16, Base32, and Base64 Data Encodings

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 14 janvier 2008. Dernière mise à jour le 29 septembre 2012

Date de publication du RFC : Octobre 2006

https://www.bortzmeyer.org/4648.html

Même les plus simples encodages peuvent réveler des surprises. Base64 en est désormais à son quatrième RFC, mais il est vrai que c'est un des formats les plus utilisés, qu'on le trouve partout dans les protocoles. Ce RFC normalise donc Base64, ainsi que Base32 et Base16.

Quel est le problème que tentent de résoudre ces encodages ? Le fait que certains environnements n'acceptent pas le jeu de caractères qu'on voudrait utiliser. C'est bien sûr le cas si ce jeu de caractères est Unicode mais même le simple ASCII peut être trop vaste pour certains usages. Par exemple, les URL ne peuvent pas comporter d'espace, et nécessitent donc un encodage de ces espaces, pour les protéger. Base64 et ses camarades sont des encodages généralistes pour tous ces cas où on doit réduire d'un jeu de caractères trop vaste à un jeu restreint : Base64 (sections 4 et 5) utilise 64 caractères (les majuscules et les minuscules d'ASCII, les chiffres, le signe plus et la barre oblique). Base32 (sections 6 et 7) utilise uniquement les majuscules et les chiffres et Base16 (section 8, mais il faut noter que ce nom de Base16 n'a jamais pris) utilise uniquement les caractères des chiffres hexadécimaux (il sert par exemple, sous un autre nom, à l'encodage des URL, cf. RFC 3986 [1], section 2.1).

La section 3 du RFC est très détaillée car elle doit expliquer tous les problèmes d'interopérabilité qui peuvent se poser avec l'encodage, notamment si celui-ci est incomplètement spécifié (ce qui était le cas avec les RFC précédant celui-ci, comme le RFC 2045). Ainsi, la section 3.1 parle du difficile problème des sauts de ligne, ou bien 3.3 se demande ce que doit faire le récepteur si des caractères en dehors du jeu normalement utilisé apparaissent néanmoins dans les données. Le monde des encodages est vaste et peu organisé... Heureusement que ce RFC décrit bien l'état actuel du problème.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc3986.txt

Historiquement, des tas de variantes plus ou moins importantes d'un de ces encodages ont été utilisées, ce qui contribue à la difficulté de la tâche de décodeur. La section 5 normalise une de ces variantes, un encodage de Base64 qui n'utilise pas le / et le +, qui ont souvent une signification particulière, mais le _ et le -. Comme il est surtout utilisé pour les URL, cet encodage est souvent nommé "*URL encoding*".

La section 12, sur la sécurité, est également très complète car, historiquement, plusieurs problèmes de sécurité ont été liés à ces encodages. Par exemple, beaucoup de spams sont encodés en Base64, même lorsque cela n'est pas nécessaire, pour tenter d'échapper à des logiciels de filtrage naïfs. (C'est au point que SpamAssassin donne désormais une mauvaise note aux messages ainsi encodés.)

Du fait de l'étendue des usages de Base64, il existe des bibliothèques pour l'encoder et le décoder dans tous les langages de programmation (Base32 est moins bien traité). Le RFC cite également une implémentation de référence <http://josefsson.org/base-encoding/> en C.

Voici un exemple d'encodeur en Perl :

```
#!/usr/bin/perl

use MIME::Base64;

foreach $word (@ARGV) {
    print("$word: " . encode_base64($word) . "\n");
}
```

Et un exemple de décodage en une seule ligne de Perl :

```
% echo Y2Fm6Q= | perl -MMIME::Base64 -e 'printf "%s\n", decode_base64(<>)'
```

Cet autre exemple est en D :

```
static import std.base64;
import std.stdio;

int main(char[][] argv)
{
  int argc = argv.length;
  char[] word;
  if (argc <= 1) {
    writefln("Usage: %.*s word ...", argv[0]);
    return 1;
  }
  for (int i = 1; i < argc; i++) {
    // It would be cooler to use 'foreach (char[] word; argv)' but it
    // does not allow to skip 0th arg (and slices do not seem to work
    // in that context)
    word = argv[i];
    writefln("%s: %s", word, std.base64.encode(word));
  }
  return 0;
}
```

Voici un exemple d'encodage Base64 en Go, montrant la différence entre l'encodage standard et l'≪ encodage URL ≫ :

```
package main

import (
"bufio"
"encoding/base64"
"flag"
"fmt"
"io"
"os"
)

const MAXSIZE int = 4096

func main() {
result := make([]byte, MAXSIZE)
value := make([]byte, MAXSIZE)
rd := bufio.NewReader(os.Stdin)
n, error := rd.Read(value)
        ...
base64.StdEncoding.Encode(result, value[0:n])
fmt.Printf("Standard encoding of %s is %s\n", value[0:n],
result)
base64.URLEncoding.Encode(result, value[0:n])
fmt.Printf("URL encoding of %s is %s\n", value[0:n],
result)
}
```

Depuis le shell, on cite parfois le logiciel aish, mais je l'ai toujours trouvé inutilisable. Je préfère OpenSSL qui a une option d'encodage en Base64 (l'argument -n d'`echo` permet de voir la différence selon qu'il y a un saut de ligne ou pas) :

```
% echo -n café | openssl enc -base64
Y2Fm6Q==
% echo café | openssl enc -base64
Y2Fm6Qo=
```

Et, pour décoder :

```
% echo Y2Fm6Qo=  | openssl enc -d -base64
```

Il y a aussi dans les coreutils un utilitaire nommé `base64`.

Notre RFC 4648 succède au RFC 3548 (qui lui même complétait le RFC 2045 sur ce point). La section 13 liste les changements (modérés), comme par exemple l'ajout d'exemples à des fins de tests des logiciels.