Apa itu **BASE64** Encoding? Coba kita implementasikan sendiri dengan **Rust**

wuriyan.to

Base64

Berdasarkan IETF, **Base64** Encoding adalah encoding yang merepresentasikan **Sequence Octet/ Buffer/ Binary Data/ Sequence 8-bit (1 byte)** dengan menerjemahkan setiap **3 Sequence byte atau setara 24 bits(8 x 3 byte)** data tadi menjadi format **4 Sequence byte atau setara (6 x 4 byte)**.

Base64 Encoding didesain untuk memenuhi kebutuhan sistem-sistem yang hanya mendukung text content, seperti **Web**. Sehingga kita dapat meng-*embed* **binary data** seperti Image file ke dalam halaman **Web** dengan merubahnya menjadi Base64 Format. Base64 Encoding juga digunakan protokol **SMTP** yang memang saat pertama kali didesain hanya mendukung **7-bit ASCII** karakter saja.

Strategi yang digunakan **Base64 Encoding** adalah menggunakan **63(2^6-1) ASCII** karakter yang *printable* (bisa ditampilkan), jadi karakter seperti **SPACE**, **TAB**, atau **NEW LINE** tidak digunakan, karena tidak *printable*.

Karakter-karakter **ASCII** yang digunakan hampir semua *printable*. Dibawah ini adalah **64 Karakter** yang digunakan pada Base64 Encoding. **Index dimulai dari 0**.

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz123456789+/

Setiap karakter tersebut diwakili oleh **index 0 - 63**. Kemudian akan digunakan **padding ‘=’** jika tiap kelipatan **3 byte (bit yang ada kurang dari 24 bit)** tidak terpenuhi.

Algoritma

Base64 Index Table (sumber: wikipedia).

Kita akan menggunakan tabel dibawah ini sebagai acuan untuk me-*lookup* karakter-karakter **Base64** dari data hasil encoding.

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Kita akan gunakan text yang simpel saja

Wury

Pisahkan menjadi setiap **3 byte segmen.** Secara kebetulan pada text Wury hanya ada **4 karakter**. Sehingga ketika dipisahkan dalam setiap **3 byte segmen** akan tersisa **1 karakter**.

Wur y

Kita akan mulai dari segmen Wur

Ambil setiap Codepoint dari segmen Wur

W = 87

u = 117

r = 114

Tampilkan dalam bentuk binary digit

W	u	r
87	117	114
01010111	01110101	01110010

W	u	r
87	117	114
01010111	01110101	01110010

Kelompokan Binary Digit-nya menjadi setiap 6-bit kelompok (24 / 4)

010101110111010101110010

010101 110111 010101 110010

Konversi setiap satu Binary Digit diatas menjadi format ***decimal***, kemudian jadikan setiap format ***decimal*** tersebut menjadi **index** untuk **me-*lookup*** tabel **Base64**, sehingga diperoleh:

010101	110111	010101	110010
21	55	21	50
V	3	V	y

Sehingga diperoleh hasil **Base64 Encoding** untuk segmen “Wur” dalam format **Base64** = “V3Vy”

Selanjutnya segmen y

Segmen y hanya memiliki karakter, sehingga sudah dipastikan kita akan menambahkan sekitar **2 padding =**. Sebaliknya jika segmen memiliki **2 karakter** misalnya yy, maka sudah dipastikan kita akan membutuhkan **1 padding =**.

Ambil Codepoint dari y

y = 121

Tampilkan dalam bentuk binary digit

y
121
01111001

Kelompokan Binary Digit-nya menjadi setiap 6 kelompok (24 / 4)

01111001

011110 01

011110 01

Konversi setiap satu Binary Digit diatas menjadi format ***decimal***,

Genapkan menjadi **6 digit** jika jumlah digitnya tidak mencapai **6**, **padding dengan 0**. Contoh 01 => 010000

Kemudian jadikan setiap format ***decimal*** tersebut menjadi **index** untuk **me-*lookup*** tabel **Base64**, sehingga diperoleh:

011110	010000		
30	16		
e	Q	=	=

Sehingga diperoleh hasil **Base64 Encoding** untuk segmen “y” dalam format **Base64** = “eQ==”

Jadi text Wury ketika di *encode* dengan menggunakan **Base64 Encoding** akan menghasilkan

Text	**Wury**
Base64	**V3VyeQ==**

Implementasi dengan Rust

Seperti implementasi dari kebanyakan encoding, untuk mengimplementasikan **Base64 Encoding**, minimal anda harus paham **manipulasi Byte dan Bit**. Selain itu anda juga harus paham konsep **Bitwise dan Bit Masking**.

Proses Encoding

```
pub fn encode(input: &mut dyn Read, out: &mut dyn Write) -> Result<(), String> {
   let mut buffer = vec![0 as u8; 3];


   loop {
       let line_read = match input.read(&mut buffer[..]) {
           Ok(o) => o,
           Err(e) => return Err(format!("error reading input {}", e))
       };
```

.....

Pada fungsi **encode** tersebut kita menggunakan **Trait Object** untuk mengabstraksi *input* maupun *output*. **Trait Object** untuk *input* saya menggunakan **Read** dari *package* **std::io**. **Trait Read** adalah **Base Trait** untuk semua operasi **IO Read**. Jadi fungsi **encode** ini fleksibel terhadap input apapun, selama **dia adalah implementasi dari Trait Read**, misalnya **File**, **Socket**, **Vector uint8**, **Standar Input**, atau bahkan anda bisa membuat **implementasi sendiri dari Trait Read tersebut**.

Implementasi dengan Rust

```
pub fn encode(input: &mut dyn Read, out: &mut dyn Write) -> Result<(), String>
```

Trait Object untuk *output* saya menggunakan **Write** dari *package* **std::io**. **Trait Write** adalah **Base Trait** untuk semua operasi **IO Write**. Jadi fungsi **encode** ini fleksibel terhadap *output* apapun, selama **dia adalah implementasi dari Trait Write**, misalnya **File**, **Socket**, **Vector uint8**, **Standar Output**, atau bahkan anda bisa membuat **implementasi sendiri dari Trait Write tersebut**.

Trait Object Read dan **Write** mengharuskan setiap *object*-nya ***mutable***. Sehingga kita berikan flag *mutable reference* `&mut`.

```
let mut buffer = vec![0 as u8; 3];
```

Kita buat variabel buffer, dan set buffernya menjadi 3, karena kita akan mengambil setiap 3 bytes untuk kita jadikan segmen.

```
loop {
       let line_read = match input.read(&mut buffer[..]) {
           Ok(o) => o,
           Err(e) => return Err(format!("error reading input {}", e))
       };
```

Input akan terus dibaca secara berulang dengan setiap operasi read mengambil 3 bytes segmen.

Implementasi dengan Rust

```
if line_read <= 0 {
    break;
}
```

Periksa jika variabel **line_read**, **jika 0 atau kurang dari 0**, berarti semua data sudah dibaca, kemudian keluar dari *looping*.

```
let seg_data = &buffer[..line_read];

let mut segment_count = 0;
let mut dec = 0;
for i in seg_data {
   let l_shift: u64 = 16 - segment_count * 8;
   let i = *i as u64;
   dec |= i << l_shift;
   segment_count = segment_count + 1;
}
```

Operasi diatas secara singkat digunakan untuk mengubah setiap **3 bytes chunk** data menjadi format decimal **Unsigned Integer 64 bit**.

Implementasi dengan Rust

```
for i in 0..segment_count+1 {
            let r_shift: u64 =  18 - i * 6;
            let b = ((dec >> r_shift) & SIX_BIT_MASK) as u8;
            let r = BASE64_TABLE[b as usize];
            if let Err(e) = out.write(r.as_bytes()) {
                return Err(format!("error write buffer out {}", e));
            }
        }
```

Selanjutnya setiap **8-bit grup atau setara 3 bytes (24/ 3 bytes)** dalam format **decimal Unsigned Integer 64 bit** dari proses sebelumnya kita ubah menjadi format **6-bit grup atau setara 4 bytes (24/ 4 bytes)**.

```
let r = BASE64_TABLE[b as usize];
```

Selanjutnya setiap **4 bytes chunk** tersebut kita gunakan untuk me-lookup **`BASE64_TABLE`** Sehingga akan kita dapatkan **4 karakter** dalam format **Base64 Encoding** untuk setiap **3 bytes data input**.

```
if let Err(e) = out.write(r.as_bytes()) {
```

Kemudian kita kirim data ke Output

Implementasi dengan Rust

```
if segment_count == 1 {
            if let Err(e) = out.write(&[PADDING, PADDING]) {
                return Err(format!("error write buffer out {}", e));
            }
        } else if segment_count == 2 {
            if let Err(e) = out.write(&[PADDING]) {
                return Err(format!("error write buffer out {}", e));
            }
        }
```

Kita gunakan variabel **segment_count** untuk memeriksa berapa jumlah **padding “=”** yang dibutuhkan, jika **segment_count = 1** kita tambah **padding “=”** dengan **2 padding**, yang berarti sisa segmen dari total data setiap kelipatan **3 bytes tersisa 1 byte**.

Jika **segment_count = 2**, berarti sisa segmen dari total data setiap kelipatan **3 bytes tersisa 2 byte**. Jadi kita tambah **padding “=”** dengan **1 padding**.

Implementasi dengan Rust

Proses Decoding

Implementasi **Base64 Decoding** adalah kebalikan dari proses Encoding-nya.

Anda bisa melihat full source code implementasinya disini:

https://github.com/wuriyanto48/b64rs

Kesimpulan

Untuk *text-text* sederhana seperti yang digunakan pada penjelasan diatas, tidak begitu berguna. Karena jika hanya sekedar text “Wury”, secara logika *text* tersebut sudah aman jika ditransfer ke sistem-sistem tertentu yang bisa menangani **Base64**. Penggunaan paling berguna biasanya pada sistem-sistem yang membutuhkan **binary data**, tetapi sistem tersebut hanya mendukung konten *text*, seperti aplikasi **Web**. Sehingga kita bisa meng-*embed* **binary data** seperti *Image file*, *Video file*, *Sound file*, dan lain sebagainya pada proses transmisi datanya.

Silahkan buka source codenya untuk contoh lain:

https://github.com/wuriyanto48/b64rs

Spec:

https://datatracker.ietf.org/doc/html/rfc4648#page-5