Member-only story

ASCII, ISO 8859–1, UCS, AND UTF

Introduction to Character Encoding

In this article, we will learn about popular character encoding schemes and how we can use them in real life.

Uday Hiwarale · Follow
Published in JsPoint
32 min read · Nov 15, 2019

Listen Share More

(**Source**: unsplash.com)

The **encoding** by definition is a way to convert data from one format to another. When we have some text (*sequence of characters*) and we want to either **store it inside a computer** (*machine*) or **transfer over a digital network**, we need to convert it to binary representation because that's the only language a **binary-based** computer can understand.

A **character encoding** is a **way to convert text data into binary numbers**. In nutshell, we can assign **unique numeric values** to specific **characters** and convert those numbers in binary language. These binary numbers later can be converted back to original characters based on their values.

Before we begin, let's understand a few terminologies first.

Binary to Hex representation

The **binary number system** is very similar to the decimal number system but we have only **0** an **1** to represent a number. This **YouTube video** explains how we can convert a decimal number to the binary number. You can also convert a binary number to the decimal number, follow **this tutorial**.

The hexadecimal number system is also similar to the decimal number system but we have **16 characters** to play with, from **0–9** and **A-F**. To convert a binary to the hexadecimal number, we need to get the decimal value of the binary number and convert it to the hexadecimal number. Follow this **video tutorial** to understand decimal to hexadecimal conversion.

In a nutshell, the hexadecimal number has the base of **16**. This means a single character of the hexadecimal number system is enough to represent values between **0–15**. Similarly, in the **decimal** number system (*base 10*), a single character can represent a value between **0–9** while in the **binary** number system (*base 2*), a single character can represent a value between **0–1**.

Since a single character of hexadecimal represents a value between **0** and **15**, and similarly, a binary **4-bit** number holds a value between **0** and **15**, we can use them interchangeably.

The binary number `1101 0011` (*decimal* `211`) in hexadecimal is represented as **`D3`**. This is because of the value of `1101` is **`13`** in decimal (`D` *in hexadecimal*) and the value of `0011` is **3** in Decimal (`3` *in hexadecimal*). If we convert `D3` to the decimal number using simple formula ($13 \times 16^1 + 3 \times 16^0$), we get `211` which is the value of the binary number `11010011`.

MSB and LSB

The **MSB** (*abbreviated from **Most Significant Bit***) is the left-most bit of a binary number. We call it MSB or the **most significant** because it has the **largest contribution to the binary number**.

For example, in the binary number `0011` , the left-most bit `0` is the **MSB**, because it ads 2^3 value to the binary number, however, the right-most bit `1` is the **LSB** (***Least Significant Bit***), because it ads 2^0 value to the binary number.

MSB can also be used as an acronym for **Most Significant Byte** and similarly, **LSB** can also be used as an acronym for **Least Significant Byte**. If a binary number is represented in multiples of 8 bits (**byte**), the left-most byte is MSB and the right-most byte is LSB.

> 💡 *You can learn more about MSB and LSB in depth from this **wiki** article.*

Endianness

There are different ways a computer system can store a **multi-byte** number. For example, the decimal number **9,905,798** in binary would look like `10010111 00100110 10000110` and in hex representation is `97 26 86` .

As we know computer stores data in a sequence of bytes. For our number, we need **3 bytes** but if this number has to be represented as a **32-bit (4 bytes)** integer, we need to add a **null byte** in the beginning.

Hence our number in 32-bit hex representation becomes `00 97 26 86` . The **MSB** in this case is `00` and **LSB** is `86` .

The **endianness** of a computer system is how it stores a multi-byte number. A system is called **big-endian** when it **stores the MSB first** (*from the big end*). Hence a big-endian system will store `00` at the lowest memory address (*first*) and `86` at the highest memory address (*last*).

Hence, in a big-endian system, our number is stored in memory as below.

```
MSB                         LSB
00        97        26      86
0x00      0x01      0x02    0x03      <- memory address
```

In contrast, the **little-endian** system **stores LSB first** (*from the little end*). Hence a little-endian system will store `86` at the lowest memory address (*first*) and `00` at the highest memory address (*last*).

Hence, in a little-endian system, our number is stored in memory as below.

```
LSB                             MSB
86        26        97        00
0x00      0x01      0x02      0x03      <- memory address
```

As you can see from the above example, the little-endian system writes and reads a number in reverse.

> 💡 *Most modern computer systems use **little-endian** format to store the data in the memory. You can follow **this thread** to understand their comparisons.*

Character Set

A **character set** or simply **charset** is a table of unique numbers assigned to different characters like **letters**, **numbers** and other **symbols**. There are many characters set like ASCII, UTF, ISO, and others. For example, the value of character **A** in the ASCII character set is **65** (*decimal*).

Encoding Scheme

An **encoding scheme** or simply **encoding** is a **way** to represent a character in binary. An encoding must follow a specific character set. For example, UTF-8 encoding follows the **UTF character set**. It uses **8-bit binary numbers** to represent a character.

Since the value of character **A** in the **UTF** character set is decimal **65** *(or hexadecimal 41)*, UTF-8 encoding encodes character **A** to `01000001` which is the binary equivalent of decimal value **65**.

Code Point

A **code point** is a decimal value associated with a character in a character set. For example, the code point of character **A** in the **UTF** charset is **65**.

ASCII Encoding

ASCII (*abbreviated from **American Standard Code for Information Interchange***) is an **encoding** and **charset** developed by **ASA** in the **1960s**. This encoding was mainly developed for electronic communications in the United States. Hence it only encodes **English characters**, **numbers** and other **symbols** used generally in the **US** and in US-based digital systems.

ASCII character set contains a total of **128 characters** and each character has a unique value between **0** and **127**. Hence a **7-bit binary number** is sufficient to represent a character from ASCII charset since a 7-bit binary number can hold values from **0** to **127** (*total of 128 unique values* → ^{27}).

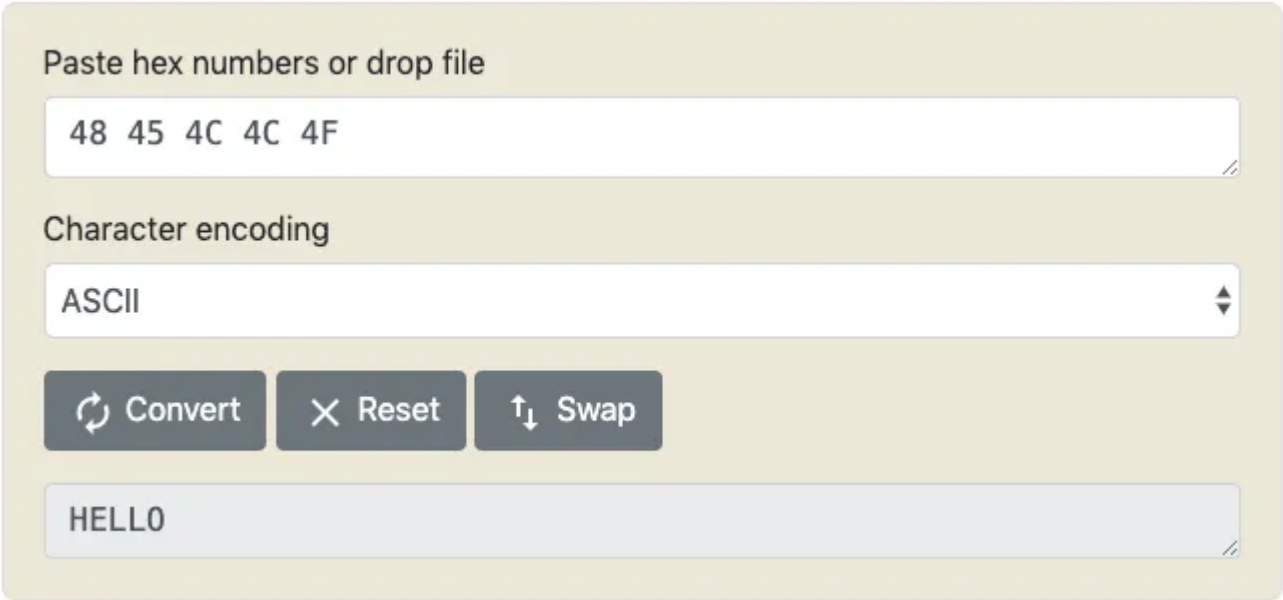
> 💡 *The **bit-width** or **bit-length** is the **length of the binary number** used by an encoding scheme to represent a character. Hence in ASCII, the bit-width is **7**.*

However, in a typical computer system, the **memory** is made of unit cells and each individual cell contains **8 bits** (*byte*). Hence, even though ASCII needs only **7 bits** to encode a character, it is stored as **8 bit** by keeping first bit **0** (*MSB*). Hence, the actual **bit-width** of ASCII is **8**.

> 💡 *Since the first bit of ASCII character is always **0**, it is also called as **dead bit** as it has no significance or use.*

You can follow **this table** to see the **code points** of characters in ASCII charset. From this table, the values of the characters in `HELLO` text in the hexadecimal representation are `48 45 4C 4C 4F`.

You can use **this online tool** to convert a sequence hexadecimal bytes to ASCII text. Using this tool, `48 45 4C 4C 4F` in ASCII is...



https://www.rapidtables.com/convert/number/hex-to-ascii.html

Examples

- **0**1000001 → 41_{16} → 65_{10} → A

- **0**1100001 → 61_{16} → 91_{10} → a
- **0**0100000 → 20_{16} → 32_{10} → (*space*)
- **0**0111001 → 39_{16} → 57_{10} → 9
- **0**1000000 → 40_{16} → 64_{10} → @

Pros and Cons

ASCII is one of the simplest encodings schemes ever developed. Since it uses only **one byte per character**, normally the text file size or network payload is small compared to other encodings.

ASCII is **fixed-width encoding** which means **the bit-width taken by a character is fixed** and it is **8** for ASCII encoding. This way, a program reading ASCII encoded text does not have to spend to much time guessing **bit-width** of a character. This makes ASCII text easier to read and write.

To this date, may web pages are still served using ASCII encoding. However, ASCII charset only contains English characters and this encoding can only be used to transport English language data.

Extended ASCII Encodings

Since ASCII uses only 7 bits to encode a character but uses 8 bits of memory to store a character, the **MSB** always remain unused. If an **encoding** utilizes this **dead bit**, we can add 128 new characters to the ASCII characters set, **without increasing the file size or network payload**.

Such encoding is called **extended ASCII encoding**. There are many encodings that supplement the ASCII character set. One of the most popular is ISO 8859–1 which adds 128 new characters to the charset.

Another popular extended ASCII encoding is **Code page 437**, also called as **DOS Latin US**. You can see the new characters beyond code point **127** from **this table**. However, this encoding was mainly used in IBM PCs.

> 💡 *You can learn more about ASCII encoding from this **wiki article**.*

ISO/IEC 8859–1

Even though ASCII is one of the popular encodings schemes, it lacks support for characters in other languages. Hence, **ISO** and **IEC** jointly created a series of standards knowns as **ISO/IEC 8859** for **8-bit character encoding.**

ISO 8859–1 is an **extension** for ASCII encoding which introduces **128** new characters by utilizing the **dead bit** of the ASCII code point. This new addition of character contains **96** printable characters and **32** control characters.

This encoding was mainly created to support **Latin Alphabets** used in various languages in Europe, hence it is also called as **ISO Latin 1.** ISO 8859–1 **charset** contains all the characters used **English**, **Spanish**, **Swedish**, **Italian,** etc. languages however languages like **Dutch**, **German**, **French**, etc. have incomplete coverage.

> 💡 ***Read more*** *about the coverage information of ISO 8859–1 and how a language with incomplete coverage came up with a workaround.*

According to this **wiki article**, ISO 8859–1 was the default encoding of the document delivered via **HTTP** with the **MIME Type** beginning with `text/` . This was changed in **HTML5** to **UTF-8.** To this date, some **HTTP header values** are still encoded in ISO 8859–1 encoding.

> 💡 *You can follow* ***this article*** *for the character set of ISO 8859–1 encoding.*

Examples

- **0**1000001 → 41_{16} → 65_{10} → A
- **0**1100001 → 61_{16} → 91_{10} → a
- **1**0101001 → $A9_{16}$ → 169_{10} → ©
- **1**0111101 → BD_{16} → 189_{10} → ½
- **1**1110101 → $F5_{16}$ → 245_{10} → õ

From the above example, we can see that ISO 8859–1 utilized the **dead bit** of ASCII encoding to add other characters. In a nutshell, **code points** of characters between **0** and **127** in ISO 8859–1 represent the same characters in ASCII charset, which makes it **compatible with ASCII.**

Hence an ASCII encoded file/data can be read by ISO 8859–1 decoder. However, an ISO 8859–1 encoded file won't be successfully parsed by ASCII decoder if code

points of the characters fall beyond **127**.

Pros and Cons

Like ASCII, ISO 8859–1 is also a **fixed-width** encoding scheme with a **bit-width of 8**. ISO 8859–1 has **96** more characters from **Latin Character Set** which covers almost all European languages.

However, despite introducing new characters, it doesn't solve the problem for all the languages. Even, some of the European languages still don't have full support. ISO 8859–1 certainly can't be used for non-Latin languages.

Unicode Consortium and UTF encodings

If we think about **8-bit fixed-width** encoding, we have only **8 bits** to represent a character. Hence the maximum of 256 characters can be represented by such an encoding.

But in the world we live in, we have multiple languages and each language has multiple characters and symbols let alone the **Chinese** language has more than **5,000** characters. **8-bit fixed-width** encoding certainly isn't enough.

Since the world was becoming more interconnected with the invention of the **internet** and the **world-wide-web**, we needed a universally accepted character set and encoding that can represent all the characters ever existed on this planet with the appetite for new characters.

The **Unicode Consortium** is a non-profit organization that maintains the **Unicode** standard. Unicode maintains the **Unicode Character Set** simply called as **Unicode** (an abbreviation for ***Universally Coded Character Set***).

The **Unicode Consortium** also maintains the standard for **UTF** encodings. **UTF** (*an acronym for* ***Unicode Transformation Format***) is a set of encoding schemes based on Unicode **charset**.

◉ — — — — ◉

UCS Encodings

Before the formation of the **Unicode Consortium**, the earlier work started with **ISO/IEC 10646** standard which defined **16-bit** and **32-bit fixed-width** encoding schemes to support characters from basic languages used across the world. UCS is *an acronym for* **Universal Coded Character Set**.

They came up with **UCS-2** and **UCS-4** encoding schemes. However, these encoding schemes are now obsolete, and more suitable **UTF** encodings are now supported backward compatibility with these encodings.

According to this *Wikipedia* article...

> This ISO/IEC 10646 standard is maintained in conjunction with The Unicode Standard ("Unicode"), and they are code-for-code identical.

UCS-2

UCS-2 is **16-bit fixed-width** encoding (*2 bytes*), which means **16 bits** will be used to encode a character. Since a character takes **2 bytes** of memory, **65,536** characters (2^{16}) characters can be represented by UCS-2 encoding.

This encoding is identical to **UTF-16** encoding but due to restriction set by **Unicode** on the character set, UCS-2 can only represent **63,488** characters while **2,048** characters are restricted by the Unicode.

> 💡 *UCS-2 is now obsolete and it is replaced by UTF-16 encoding.*

UCS-4

UCS-4 is 32-bit fixed-width encoding (4 bytes), which means **32 bits** will be used to encode a character. Since a character takes **4 bytes** of memory, **4,294,967,296** characters (2^{32}) characters can be represented by UCS-4.

However, Unicode charset can only contain **1,112,064** characters. Since **UCS** and **Unicode** follow the same character set, **UCS-4** is restricted to encoded only **1,112,064** characters.

> 💡 *UCS-4 and UTF-32 encoding are similar in every which way.*

Even though **UCS** encoding schemes added support for additional characters but due to their nature of being **fixed-width** encodings, **UTF** encodings were finally

taken over.

🐯 Since **UCS-2** is now **obsolete** and identical to **UTF-16** while **UCS-4** is exactly identical to **UTF-32**, we are not going through their details. However, while learning UTF encoding, you can get the idea of how these encodings.

◉ — — — — ◉

UTF Encodings

So far, we learned about **fixed-width encodings**. A fixed-width encoding scheme uses a fixed length of bits to store the **code point** of a character. **ASCII** used **8 bits** to encode a character while **UCS-2** uses **16 bits**.

A **variable-width encoding** in opposed to fixed-width encoding uses **variable bits** to encode a character **when necessary**. For example, when the **code point** of a character can be represented in **8 bits**, this encoding will use 8 bits to store the code point. For example, code point $\mathbf{65_{10}}$ or character **A**.

But when the code point of character can not be represented in 8 bits, more than 8 bits are used. Since we can store data in multiples of 8 bits, **2 or more bytes** are used to encode a character width code point greater than **256**.

For example, the character **Ā** whose **code point** is $\mathbf{256_{10}}$ or $\mathbf{100_{16}}$ (*in Unicode*), can only be stored in more than **1 byte** of memory, because the minimum value that can be stored in 1 byte is **255**.

UTF encoding provided multiple encoding schemes, both fixed-width and variable-width. Such encoding schemes are UTF-8, UTF-16, and UTF-32.

UTF-8

UTF-8 is an **8-bit variable-length** encoding scheme designed to be **compatible** with **ASCII** encoding. In this encoding, from **1** up to **4** bytes can be used to encode a character (*unlike ASCII which uses fixed 1 byte*). UTF-8 encoding uses the **UTF character set** for character code points.

In a **variable-length encoding** scheme, a character is encoded in **multiple** of **N** bits. For example, in UTF-8 encoding, a character can take **M x 8** bits of memory, where

N is **8** (*fixed*) and **M** can be **1** up to **4** (*variable*).

Here, **N** is also called as the **code unit.** The code unit is the building block of the **code point** (*coded character representation*). In UTF-8 encoding, the code unit is **8 bits** or **1 byte** because a character is encoded in N **bytes.**

The main idea behind UTF-8 was to encode all the characters that could possibly exist on the planet but at the same time support ASCII encoding. This means that an ASCII encoded character will look exactly similar in UTF-8.

> 💡 *This also means that UTF-8 is the superset of ASCII.*

```
01000001  ->  A (ASCII)
01000001  ->  A (UTF-8)
```

In the above example, you can see that the **code point** of the character **A** is **65** in both the encodings and they are represented in the same way. This is the same for all the characters in the ASCII charset. Hence a UTF-8 decoder can easily read an ASCII document.

The real fun begins when we have to encode a character that goes beyond the ASCII charset. We know that an ASCII character uses only **7 bits** of memory hence the maximum value of the code point is restricted to **127**. So how do we encode a character whose code point is larger than 127?

> 💡 *The Unicode Consortium maintains the character set used across UCS and UTF encodings. Each character has a fixed code point. You can follow **this table** to look up any character and see its code point in hexadecimal.*

The simple answer to that is, **add more code units.** If a character has a code point greater than 127, let's say 128, we would add one more code unit (*1 more byte*) and a UTF-8 decoder will consider **2 bytes** to generate the code point of the character (*which would yield more value than 127*).

But the real question is, **how the decoder will know if it needs to consider one byte, two bytes or more to decode a character?**

Let's consider a simple example. We need to store a plain text file that contains text **dõg** in UTF-8 encoding. From our earlier understanding, character **d** and **g** belong to the ASCII table but character **õ** does not.

If we look up the code point of character o in the UTF **charset table**, its code point is $\mathbf{245_{10}}$ or $\mathbf{F5_{16}}$. Hence it **must take more than one code unit** to encode a character, **but how many**?

The UTF-8 standard follows a very basic approach. If a character has a code point greater than **127**, it can take **2 or more** code units to encode the character. The UTF-8 decoder can look at the **starting bits** of the code unit and calculate how many bytes it follows needs to be considered.

- While reading a file, if a UTF-8 decoder encounters a **byte** with leading **0 bit**, it means the character is from ASCII charset and takes only **1 code unit** or **1 byte**. Hence UTF-8 has to read that byte only to generate the code point (*hence a character*). And then it moves to the next code point. For example, if the code unit is $\mathbf{01000001_2}$, then it is an ASCII character which is **A** with the code point $\mathbf{65_{10}}$ or $\mathbf{41_{16}}$.
- If a UTF-8 decoder encounters a **code unit** (*byte*) with leading **1 bit**, then it knows that this is not an ASCII character and this character can take **2 or more** code units.
- If a character takes 2 or more code units, each code unit **except the first one** has the leading **2 bits** set to $\mathbf{10_2}$ to signify that they are **continuation** bytes or **continuation** code units.
- The **leading code unit** use **leading bits** to tell the UTF-8 decoder how many **continuation** code units this character takes.

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

(Source: https://en.wikipedia.org/wiki/UTF-8)

From the above UTF-8 table, you can see that all ASCII characters take only one code unit to encode a character in UTF-8 and their leading bit is **0**.

However, if the code point of a character is between $\mathbf{80_{16}}$ and $\mathbf{7FF_{16}}$ then it needs **two code units** to encode the character. The first code unit has $\mathbf{110_2}$ leading bits to signify that it follows **1 continuation code unit**. The other **continuation** code unit starts with $\mathbf{10_2}$.

Similarly, if the code point of a character is between $\mathbf{800_{16}}$ and $\mathbf{FFFF_{16}}$, then it needs **three code units** to encode the character. The first code unit has $\mathbf{1110_2}$ leading bits to signify that it follows **2 continuation code units**. The rest **continuation** code units start with $\mathbf{10_2}$.

A UTF-8 decoder just has to look at the leading code unit to calculate how many bytes (*code units*) it needs to consider to decode a character. After that, it can calculate the code point value from the **bits used to encode the code point** (*mentioned in the **Bits for code point** column in the above table*).

Let's consider our old example. We need to encode the text **dõg** in UTF-8. As we know, **d** and **g** belong to the ASCII table, their code points are $\mathbf{100_{10}}$ and $\mathbf{103_{10}}$ respectively and in binary, they are $\mathbf{1100100_2}$ and $\mathbf{1100111_2}$.

For the character **õ**, we need to convert its code point value $\mathbf{245_{10}}$ to **11-bit** binary number which yields **00011 110101**$_2$. Then we can fit this binary representation in the byte sequence for UTF-8 encoding (*see the above table*).

> 💡 *If you are confused about why we use 11-bit binary number, please refer to the table above. Since code point $\mathbf{245_{10}}$ or $\mathbf{F5_{16}}$ belongs to 2nd row, we have 11 bits to encode the code point with 2 code units.*

Hence our final text in UTF-8 encoding will look like below.

```
01100100 11000011 10110101 01100111    <- binary
64       C3       B5       67          <- hex
-------- ---------------- --------
d        õ                g           <- characters
```

You can use a **Hex Editor** to store above hexadecimal numbers as file's content and open that file in UTF-8 text reader. You will see the result **dõg**.

You can use **this online tool** to do the same and see results yourself. Enter **64 C3 B5 67** in the hexadecimal text box to see the characters.

> 💡 *You can also use **this online tool** to generate UTF-8 byte sequence for a character without banging your head against the wall.*

Pros and Cons

UTF-8 is an awesome encoding scheme and there are many reasons for it. As it supports compatibility with ASCII, any ASCII encoded document is a valid UTF-8 document. However, the reverse is not true (*see the earlier example*).

Since UTF-8 is a variable-length encoding, it does need to waste memory like UCS-2 or UCS-4 to represent a character with fixed **16 bits** or **32 bits** which could have been easily encoded in **8 bits**.

UTF-8 is **self-synchronizing**. This means if a program reding UTF-8 text file jumps at a random code unit, it knows when the next leading code unit of a character begins. It just has to look at the **initial bits** of the code unit.

> 💡 *Self-synchronization is critical to any good character encoding. If a text reader stumbles upon a random byte, it does not have to start reading the document from the beginning to find the next or previous valid character from that byte.*

UTF-8 encoding also doesn't have to worry about the endianness of the system. This is because the UTF-8 code unit is just **8 bits** long. In the end, **all UTF encodings are read as a stream (*sequence*) of code units**. Hence in UTF-8, **only 8 bits are read at a time to generate a numeric value** (*this will be easy to understand in UTF-16 section*).

Due to these facts, most web pages served on the internet are UTF-8 encoded[0]. Also, most of the text documents shared over the internet are UTF-8 encoded. So UTF-8 is quickly becoming the **de facto** standard for encoding.

A web page or a document on the internet can send information about its encoding in `Content-Type` header which generally is like `Content-Type: <MIME Type>; charset=<encoding>` . Hence for an HTML page encoded in UTF-8 will send `Content-Type: text/html; charset=UTF-8` header with the response.

> 💡 ***HTML5** uses UTF-8 encoding as the default encoding of a HTML document unless mentioned in the **charset** META tag, read more.*

Internet Assigned Numbers Authority (IANA) prefers **UTF-8** or **utf-8** as the identifier for the UTF-8 encoded document. So you should watch out for `Content-Type` header or **meta-information** of a file because if a file is encoded in one scheme but informs about the other can cause a problem.

Despite these benefits of UTF-8, some might argue that UTF-8 is not very efficient when it comes to represent characters outside the ASCII range, because those characters needed to be encoded with more than 1 byte.

Hence, some of the languages like **Java** and **JavaScript** use **UTF-16** as their default character encoding scheme. UTF-16 uses **2 bytes per code unit** but it brings a lot of mess with it. Let's get into that.

UTF-16

UTF-16 is **16-bit variable length** encoding scheme and it uses the UTF character set for character code points. This means that a UTF-16 encoded character will have a **16-bit code unit**.

As we know that a UTF-8 encoded character can be represented in 1 to 4 code units, a UTF-16 character can be represented in **1 or 2** code units. Hence a UTF-16 character can take **16** or **32** bits of memory based on its **code point**.

Before jumping into the UTF-16 encoding specifications, let's understand how we can make UTF-16 work.

Since we have 16-bit code unit, in theory, we can encode 2^{16} characters from code point **0** to **65,535**. But what if we have a character with the code point greater than 65,535? In that case, **we can add another code unit**.

With the extra code unit, we can encode a total of 2^{32} characters which is more than **4M**. But then the question is, how a UTF-16 decoder will know that it needs to consider 2 code units to decode a character?

UTF-8 solved this problem by setting **initial bits** of the **first** code unit and **continuation** code units to some specific bit values which a UTF-8 decoder can use to deduct how many code units a character can take.

We can do the same with the UTF-16 code unit but then we have to sacrifice some bits in a code unit for this feature. We can set some initial bits of a code unit to some meaningful value that a UTF-16 decoder can understand.

Also to give **self-synchronizing** power to code units, a code unit must be able to tell whether it is the **initial code unit** or a **continuation code unit** and not a character of just one code unit.

So Unicode decided to sacrifice the initial **6 bits** of the code unit leaving only **10 bits** to encode the code point of a character per code unit. If a character needs **2** code units, **20 bits** of the memory (*out of 32 bits or 4 bytes*) contains the actual **code point information** of the character.

So what are these initial bits and how **these bits make a dent** in the UTF character set? Let's follow the below example.

```
1101 10xx xxxx xxxx    1101 11xx xxxx xxxx
FIRST CODE UNIT----    SECOND CODE UNIT---
```

From the UTF-16 standard, the first code unit should start with $\mathbf{110110}_2$ and the second code unit should start with $\mathbf{110111}_2$. This will help a UTF-16 decoder to understand which one if the first code unit and which one is the second. This makes UTF-16 **self-synchronizing**.

Now what we have **10 bits** per code unit to play with, what is the range we can play within? In the end, **how many characters can be encoded in two code units** of UTF-16 encoding?

> *Don't worry, we will talk about characters encoded in just one code unit.*

If you take a look at the above code unit templates, we have a range from **1101 1000 0000 0000**$_2$ to **1101 1111 1111 1111**$_2$. That is equivalent to $\mathbf{D800}_{16}$ to $\mathbf{DFFF}_{16}$.

> 💡 *The first code unit has range from* $\mathbf{D800}_{16}$ ***to*** $\mathbf{6FFF}_{16}$ *and the second code unit has the range from* $\mathbf{DC00}_{16}$ *to* $\mathbf{DFFF}_{16}$*. We can get these values by turning all* ***code points bits : on*** *and* ***off****.*

Since UTF-16 has to be **self-synchronizing**, code points between $\mathbf{D800_{16}}$ and $\mathbf{DFFF_{16}}$ must not represent a character in UTF-16. Since all UTF encodings follow the same UTF character set, **these code points are restricted by UTF** and they are not and won't be assigned to any characters[0].

Code points between $\mathbf{D800_{16}}$ and $\mathbf{DFFF_{16}}$ do not represent any characters hence they are called **surrogate code points** or **together** they are also called as **surrogate pairs**[0].

The first surrogate code point (*from first code unit*) also called as **high surrogate** and second code point (*from second code unit*) also called as **low surrogate**. Making the total of **2048** code points, **1024** per surrogate.

> 🤦 *Surrogate code points **sacrifised their lives** so that we could encode more characters with **two code units**. Think about that!*

So the big question, can we encode a character with a single code unit of UTF-16? The answer is **YES**. UTF-16 is a 16-bit **variable-length** encoding scheme. So does that mean, we can encode 2^{16} characters with a single code unit?

The answer is **NO**. In theory, we could encode 2^{16} characters with code point $\mathbf{0000_{16}}$ (0_{10}) to $\mathbf{FFFF_{16}}$ (65535_{10}) but code points between $\mathbf{D800_{16}}$ and $\mathbf{DFFF_{16}}$ do not represent any characters as they are reserved.

Hence it is **safe to encode** characters from $\mathbf{0000_{16}}$ to $\mathbf{D7FF_{16}}$ and $\mathbf{E000_{16}}$ to $\mathbf{FFFF_{16}}$ leaving which accounts to **63,488** (*65536–2048*) characters. This is just for the characters that can be encoded in just one code unit of UTF-16.

Since we have a total of **20 bits** to play with when it comes to characters than can be encoded in 2 code units of UTF-16, we can encode 2^{20} more characters, which is **1,048,576** characters.

So in total, we can encode **1,048,576** + **63,488** which amounts to **1,112,064** characters (*more than **1Million** characters*). This is the limit of the UTF character set. Since UTF-16 can encode these many characters, other UTF encodings can not represent characters beyond these.

UTF charset Code Points

As we know that a code point is a decimal value assigned to a character, we (*Unicode*) must not assign an invalid code point to an actual character. So far, the invalid code points are **surrogate code points**.

With **just a single** UTF-16 code unit, we can encode **63,488** characters ranging from $\mathbf{0000_{16}}$ to $\mathbf{D7FF_{16}}$ and $\mathbf{E000_{16}}$ to $\mathbf{FFFF_{16}}$. The last code point is **65,535.** These are called **BMP** characters (*explained later*).

With **two code units** of UTF-16, we can encode **1,048,576** characters. Since we can not again start from **0** value (*code point*) because these come after **BMP** characters, we need to offset them by **65,536.** These characters are called **Supplementary** characters (*explained later*).

Hence the first **supplementary** character has a code point value of $\mathbf{65536_{10}}$ which is equivalent to $\mathbf{10000_{16}}$. Since we can encode **1,048,576** characters with two code units of UTF-16, the last code point is $\mathbf{1114111_{10}}$ which is equivalent to $\mathbf{10FFFF_{16}}$.

So let's break down things in a simple tabular form.

UTF-16 CU	Code Point	
1	$\mathbf{0000_{16}}$ – $\mathbf{D7FF_{16}}$	valid
1	$\mathbf{D800_{16}}$ – $\mathbf{DFFF_{16}}$	invalid(surrogate)
1	$\mathbf{E000_{16}}$ – $\mathbf{FFFF_{16}}$	valid
2	$\mathbf{10000_{16}}$ – $\mathbf{10FFFF_{16}}$	valid
	$\mathbf{110000_{16}}$ – $\mathbf{FFFFFF_{16}}$	unassigned

With this knowledge, let's see how we can encode some characters in UTF-16. Let's pick a simple **ASCII** character **A** (*code point: 41_{16}*), a character from **Hindi** (*Indian*) language **आ** (*pronounced as **Aa**, code point: $\underline{906_{16}}$*) and an emoticon 😊 (called as ***Happy face**, code point: $\underline{1F60A_{16}}$*).

As we can see from the above table, both **A** and **आ** can be encoded in **just one code unit** of UTF-16 since their values are less than $\mathbf{FFFF_{16}}$.

When we have to encode a character in just **one code unit**, we just have to convert the code point of the character in a **16-bit binary number.** For the characters **A**, $\mathbf{00000000\ 01000001_2}$ is the UTF-16 representation.

Similarly, for the character **आ**, we just need to convert its code point 906_{16} to a 16-bit binary number which is $00001001\ 00000110_2$.

Normally, we represent code units of a character in hexadecimal numbers. Hence for character **A**, the UTF-16 representation is 0041_{16} and similarly, for the character, UTF-16 representation **आ** is 0906_{16}.

For the character 😊, things are a little different. Its code point is $1F60A_{16}$. If we look at the UTF-16 table mentioned above, it has to be encoded in **2 code units** of UTF-16. So how do we begin?

First, we need to subtract 10000_{16} from the code point. The reason for that is, every character encoded in 2 code units of UTF-16 has come after the BMP characters whose last code point is $FFFF_{16}$.

Hence to get the actual **value of bits used for encoding** (*which is 20 in 2 code units*), we need to subtract 10000_{16} from the code point and use the final number to generate these 20 bits.

> 💡 *To help you understand better, the first character represented with 2 code units of UTF-16 will have all its **20 bits** set to **0**. So the value of the bits used to encode the code point of this character is 0. But still, its code point is 10000_{16} according to **Unicode character set**. This is because the value yielded by these 20 bits is added to 10000_{16} to generate the final code point.*

As seen earlier, these 2 code units look like below.

```
1101 10xx xxxx xxxx    1101 11xx xxxx xxxx
FIRST CODE UNIT----    SECOND CODE UNIT---
```

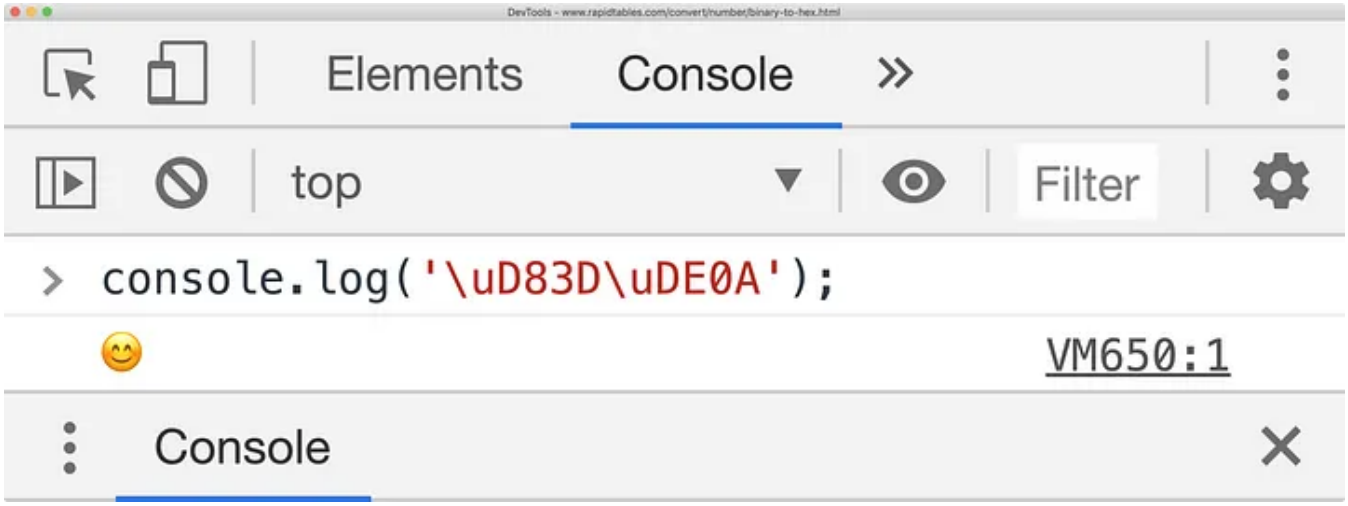
We just need to fill these 20 bits (x) with the value received from the character code point. The code point of the character 😊 is $1F60A_{16}$. But first, we need to subtract 10000_{16} from it. We get $F60A_{16}$.

Now we need to convert $F60A_{16}$ to a 20-bit binary number and fill the 20 bits in the above code unit template. $F60A_{16}$ in binary is $0000111101\ 1000001010_2$. Now we can fill these 20 placeholder bits.

Below are the final code units.

```
1101 1000 0011 1101    1101 1110 0000 1010
0xD83D                 0xDE0A
```

The quick way to check if these code units are valid and in fact if these surrogate pairs can represent the encoded character, open a Browser DevTool and enter `console.log('\uD83D\uDE0A');` in the console.



(Chrome Developer Tools)

You can also use this **online tool** to generate UTF-16 code points.

Unicode Character Planes

A plane is a **continuous group** of 2^{16} or **65,536** code points. Since UTF-16 has restricted code points to a maximum of $\mathbf{10FFFF_{16}}$, we have a total of **17** characters planes in Unicode standard starting from 0 to 16.

Since $\mathbf{2^{16}}$ **characters** can be defined by the single code unit of UTF-16 (*including surrogate code points*), it forms the first (*0th*) plane. This plane contains almost all the characters in **basic languages** around the world. This is why this plane is called the **Basic Multilingual Plane** or **BMP.**

Next, we have code points defined by two code units of UTF-16. These contain $\mathbf{2^{20}}$ **characters** as we have 20 bits to encode the code point value. These are divided into **16 planes** (2^4 x 2^{16}). These are called **supplementary planes.**

> 💡 *For more information on these planes, read this Wikipedia document.*

Comparison with UCS-2

UCS-2 is a 16-bit fixed-width encoding. That means **only one 16-bit code unit** is used to represent a code point. In theory, UCS-2 can represent 2^{16} distinct characters but there is a twist.

> 💡 *BTW, we are using the term **code unit** in this fixed-width encoding to understand the relation between UTF-16. In reality, there is no such thing as **code unit** in any fixed with encoding.*

Since UCS follows the Unicode character set, the encoding of the characters in UCS-2 is identical to the encoding of the characters in UTF-16 which are represented in just one code unit.

> 💡 *Since UCS follows the Unicode character set, it can not encode a valid character with the code points reserved for the surrogates.*

So in nutshell, UCS-2 contains the characters of **Basic Multilingual Plane**. This is the reason, some older documents, and software used UCS-2 encoding. But UCS-2 encoding has become **obsolete** and UTF-16 is preferred.

Endianness and BOM

As we discussed before, A UTF encoded documents on a low level contain the sequence of code units. For **UTF-8**, the code unit is **8 bits** long while for **UTF-16**, it is **16 bits** long. These code units make up the characters.

A UTF-8 or UTF-16 decoder reads the code units sequentially, one code unit at a time to generate the characters.

Each code unit represents a **numeric value** that a UTF-8 or UTF-16 decoder can take a look at and decide whether it is sufficient to represent a character or it follows other code units that should be considered as well.

When it comes to UTF-8, things are simple. Since each code unit is 8 bits long, converting that 8-bit binary number to a **numeric value** is quick and easy. This is not the case with UTF-16 though.

UTF-16 code unit is a 16-bit (*2 bytes*) binary number that represents a code point value. To generate the numeric value from multiple bytes, in general, is tricky and different systems behave differently.

This behavior depends on the **endianness** of the system. From our earlier discussion about endianness, there are two ways we can write a UTF-16 code unit value. Either in **Big-endian** format or **Little-endian** format.

In Big-endian format, the MSB is stored first and LSB is stored last. So far, we are writing the UTF-16 code unit value in the Big-endian format. To write UTF-16 code unit value in Little-endian, we need to swap bytes.

Let's talk about character **आ**. From the earlier example, it can be represented in just one code unit of UTF-16 and its encoding in hexadecimal representation looks like $\mathbf{0906_{16}}$.

$\mathbf{0906_{16}}$ is a 16-bit number with **09 being the MSB** and **06 being the LSB**. Hence in **Big-endian** architecture, it will be stored as **09 06**. However, in a **Little-endian** architecture, it will be stored as **06 09**.

Hence it becomes our responsibility to encode characters by taking endianness of the system in mind so that the system can read a UTF-16 document correctly.

But how can we tell beforehand whether a user's machine is compatible with the encoded document or not? And since the endianness of a system can affect how a document is decoded, how do we share it publically?

This is where **BOM** comes into the picture. A **Byte Order Mark** (BOM) is a byte sequence that is added at the beginning of a text file or text data.

Unicode recommends character with code point $\mathbf{FEFF_{16}}$ to act as a BOM for UTF-16 and UTF-32 encodings. This character should before the first character of the document. However, this character won't be considered by the decoder in the output.

This character (**U+FEFF**) is a **zero-width non-breaking space** (*ZWNBSP*) character and it's **invisible**. Hence even if a decoder fails to recognize the BOM, it won't produce any visible output.

This character is represented in a single code unit of UTF-16 and in hexadecimal representation it looks like **FE** (*MSB*) and **FF** (*LSB*).

Hence when characters are encoded in **Big-endian** format, we need to add **FEFF** as the BOM at the beginning of the file and when characters are encoded in **Little-**

endian format, we need to add **FFFE** (*reverse*) as the BOM at the beginning of the file.

Unicode recommends adding the BOM to UTF-16 encoded document. However, if the BOM is missing then Big-endian format is assumed.

IANA prefers **UTF-16** as the identifier to signify a UTF-16 encoded document. However, **UTF-16BE** is used for document encoded in **Big-endian** format and **UTF-16LE** is used for **Little-endian** format.

When **UTF-16BE** or **UTF-16LE** name is used, then BOM is not recommended to be prepended to a file. Even in this case, if the BOM is added, then it will be considered as **ZWNBSP** character and it won't be ignored.

> 💡 *UTF-16, UTF-16BE and UTF-16LE names are case insensitive.*

Pros and Cons

UTF-16 is efficient because it has only 2 code units and since most used characters fall in the BMP set, they can be represented in just one code unit. However, it comes with lots of issues.

The biggest disadvantage of UTF-16 is that it is not ASCII compatible. Since ASCII characters are encoded with a single code unit (*16-bit number*), they can not be decoded properly by an ASCII decoder.

UTF-16 consumes unnecessary space for ASCII characters. Compared to the UTF-8 encoded document which contains only ASCII characters, the size of the same document encoded in UTF-16 is two times bigger.

UTF-16 also gets affected by the endianness of the system. If the BOM is missing and an appropriate encoding identifier is not used (*like UTF-16LE*), a UTF-16 encoded document may not get decoded properly.

Due to the nature of UTF-16 encoding, it has introduced surrogate code points that can not represent the valid characters. Also, it has limited the Unicode character set to $\mathbf{10FFFF}_{16}$ (*last code point*).

Despite these facts, some of the programming languages like **JavaScript**, **Java**, etc. and systems like **Windows** prefer UTF-16 encoding.

UTF-32

UTF-32 is 32-bit **fixed-width** encoding scheme which means a single 32-bit code-unit will be used to encode a character.

In contrast with UTF-8 and UTF-16, since we are not dealing with multiple code units, encoding in UTF-32 is fairly easy. We just need to **convert the code point of a character in a 32-bit binary number**.

Let's encode some characters in UTF-32. We will use previously used characters for simplicity.

For character **A** with code point $\mathbf{41_{16}}$, its binary representation is $\mathbf{1000001_2}$. Since we have to store the code point in 32-bits, we need to **pad-left** the binary number with **0** bits. This will yield $\mathbf{00000000\ 00000000\ 00000000\ 01000001_2}$ or $\mathbf{00\ 00\ 00\ 41_{16}}$.

For character **आ** with code point $\mathbf{906_{16}}$, its 32-bit binary representation is $\mathbf{00000000\ 00000000\ 00001001\ 00000110_2}$ or $\mathbf{00\ 00\ 09\ 06_{16}}$. Similarly, for the character 😊 with code point $\mathbf{1F60A_{16}}$, its binary representation is $\mathbf{00000000\ 00000001\ 11110110\ 00001010_2}$ or $\mathbf{00\ 01\ F6\ 0A_{16}}$.

> 💡 *You can use **this online tool** to see the UTF-32 encoded byte sequence. You just need to copy and paste character in the **utf8** text box.*

Comparison with UCS-4

UCS-4 is 32-bit fixed-width encoding scheme and since UCS uses the UTF character set, it is identical to UTF-32 in every which way.

Endianness and BOM

A UTF-32 decoder considers **4 sequential bytes** to generate the numeric code point value hence it will get affected by the endianness of a system.

Hence, Unicode recommends adding the **Byte Order Mark** (*BOM*) before the beginning of a UTF-32 encoded file. As we saw in UTF-16, we used the **ZWNBSP** character with code point $\mathbf{FFFE_{16}}$.

We need to add the same character in UTF-32 but encoded in **4 bytes**. Hence when we have encoded character in **Big-endian** format, we use **00 00 FF FE** byte sequence as the BOM or **FE FF 00 00** in case of **Little-endian** format.

If this BOM is missing in the document, the **Big-endian** format will be considered by the UTF-32 decoder by default.

IANA prefers **UTF-32** as the identifier to signify a UTF-32 encoded document. However, **UTF-32BE** is used for document encoded in **Big-endian** format and **UTF-32LE** is used for **Little-endian** format.

When **UTF-32BE** or **UTF-32LE** name is used, then BOM is not recommended to be prepended to a file. If still a BOM is added, then it will be considered as **ZWNBSP** character and it won't be ignored.

> 💡 *UTF-32, UTF-32BE and UTF-32LE names are case insensitive.*

Pros and Cons

A great reason to encode text data in UTF-32 would be to increase the efficiency of the UTF-32 decoder. Since all possible UTF-32 characters can be encoded in just one code unit, searching and sorting operations are fast.

However, the biggest disadvantage of using UTF-32 encoding is **memory consumption**. Since all characters, even in BMP will consume 32-bit memory, it is not efficient when it comes to **disk space** or **network bandwidth**.

As we know that, the last valid code point in UTF charset is $\mathbf{10FFFF_{16}}$ which is only **6 bytes** long, hence a 32-bit encoded character will always have **1 byte** empty and of no use.

U+ convention

When we have to define a Unicode character with its code point value in hexadecimal number, Unicode recommends using the **U+** prefix follows by **4 or 6 hexadecimal code point digits**.

For character **A** with code point $\mathbf{41_{16}}$, it will be represented in Unicode notation as **U+0041**. For character **आ** with code point $\mathbf{906_{16}}$, its Unicode notation will be **U+0906**. Similarly, for the character 😊 with code point $\mathbf{1F60A_{16}}$, its Unicode notation will be **U+1F60A**.

Links

Below are some of the useful links for Unicode character conversions.

Binary, Hex and Decimal conversions

1. Hexadecimal to Binary conversion
2. Decimal to Hexadecimal conversion
3. Binary to Hexadecimal conversion

Unicode Standard

1. Unicode and Unicode Consortium
2. UTF-8 Wikipedia
3. UTF-16 Wikipedia
4. UTF-32 Wikipedia
5. UTF Frequently Asked Questions (FAQs)
6. BMP and Unicode Planes

UTF Encoders and Decoders

1. Character → Code Point
2. Code Point → Character
3. Character → UTF-8, UTF-16, UTF-32
4. UTF-8 Code Units (*hex*) → character
5. UTF-16 Code Units(*hex*) → Character
6. UTF-32 Code Unit(*hex*) → Character
7. Character → UTF-16 (*play with BOM and Endianness*)
8. Character → UTF-32 (*play with BOM and Endianness*)

Character and Code Point Search

1. ASCII table
2. ISO 8859–1 table
3. Unicode table

I have created an NPM package with the name **utf-info** to display extensive encoding information of a character in UTF-8, UTF-16, and UTF-32.

(thatisuday.com / GitHub / Twitter/ StackOverflow / Instagram)

Please clap 👏 if you found this article useful.

Unicode | Encoding | Utf 8 | Character Encoding | Programming

Follow

Written by Uday Hiwarale

8.4K Followers · Editor for JsPoint

Software Software Engineer at SAP ✦ github.com/thatisuday ✦ twitter.com/thatisuda

More from Uday Hiwarale and JsPoint

Uday Hiwarale in JsPoint

How the browser renders a web page?—DOM, CSSOM and Rendering

In this article, we will deep dive into DOM and CSSOM to understand how the browser renders a webpage. Brower blocks some rendering of a...

· 25 min read · Aug 3, 2019

5.5K 36

Uday Hiwarale in JsPoint

How does JavaScript and JavaScript engine work in the browser and node?

JavaScript's call stack, event loop, task queues, and various other pieces that make JavaScript as we know it.

✦ · 16 min read · Apr 23, 2018

2.8K 18

Uday Hiwarale in JsPoint

A simple guide to load C/C++ code into Node.js JavaScript Applications

In this article, we are going to get ourselves familiar with the mechanism and tools to load the C/C++ code dynamically in JavaScript...

· 23 min read · Dec 5, 2019

1.1K 11

Uday Hiwarale in JsPoint

A quick introduction to “Promises” and “Async/Await” (with new features)

In this lesson, we are going to learn about ES6 Promises implementation in TypeScript and async/await syntax.

· 10 min read · Aug 1, 2020

170 1

See all from Uday Hiwarale

See all from JsPoint

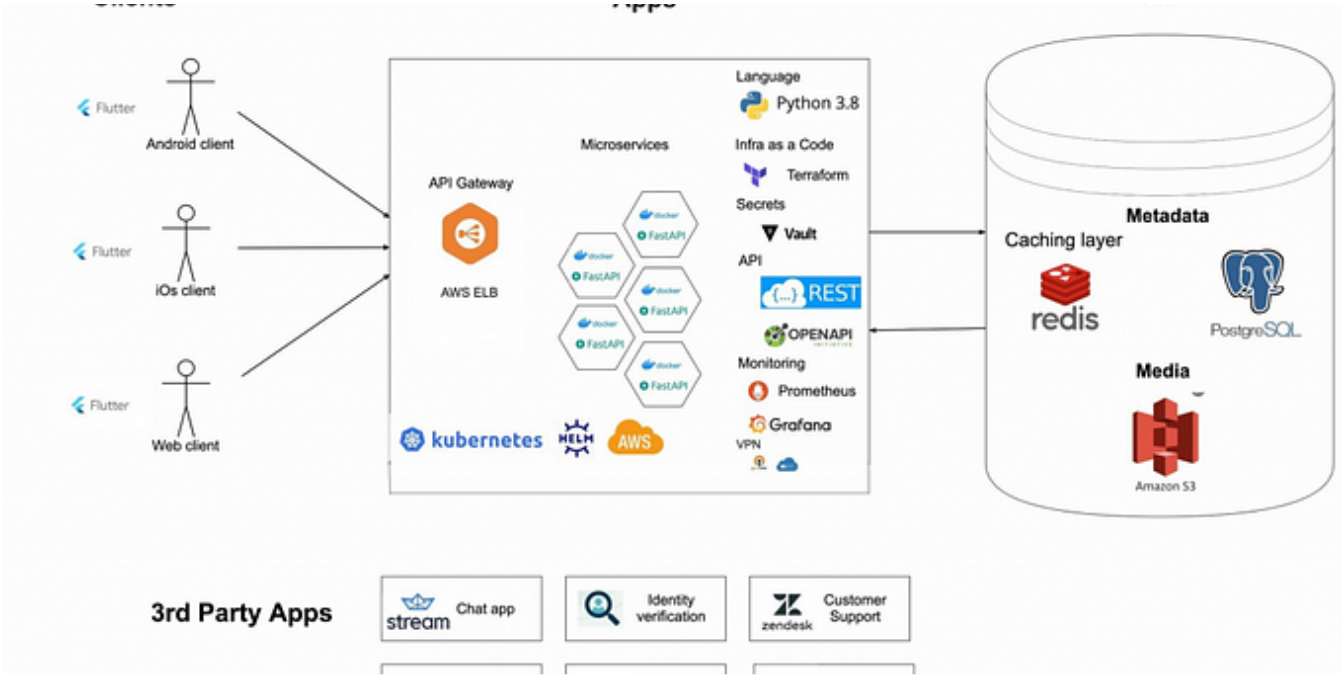
Recommended from Medium

Simsangcheol

Memory paging

Memory paging is a way for a computer to manage its memory more efficiently. Imagine your computer's memory as a big storage room filled...

2 min read · Apr 20



Dmitry Kruglov in Better Programming

The Architecture of a Modern Startup

Hype wave, pragmatic evidence vs the need to move fast

16 min read · Nov 7, 2022

6K 52

Lists

Staff Picks

454 stories · 298 saves

Stories to Help You Level-Up at Work

19 stories · 224 saves

Self-Improvement 101

20 stories · 601 saves

Productivity 101

20 stories · 558 saves

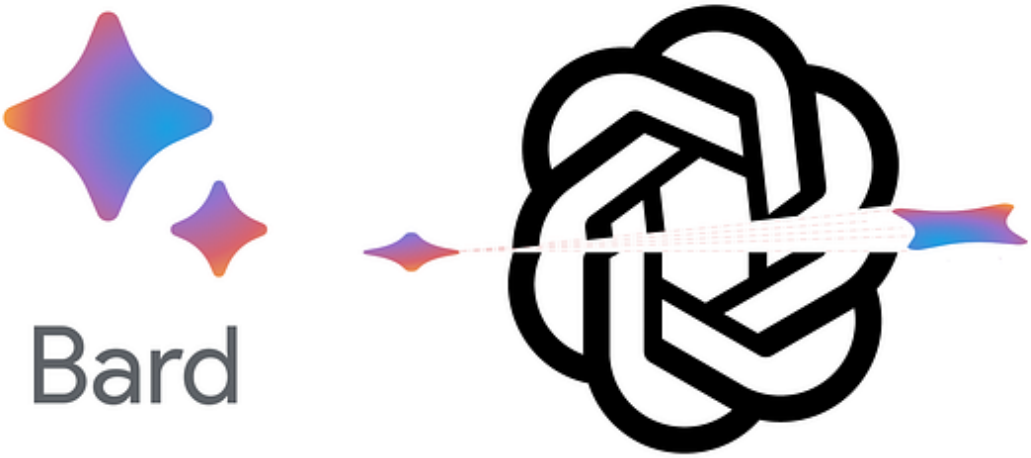
Ebo Jackson in Level Up Coding

Comparing Concurrency and Parallelism Techniques in Python: Threading, Multiprocessing, Concurrent.f

Introduction

5 min read · Jun 13

70



AL Anany

The ChatGPT Hype Is Over—Now Watch How Google Will Kill ChatGPT.

It never happens instantly. The business game is longer than you know.

· 6 min read · Sep 1

9.3K 293

Unbecoming

10 Seconds That Ended My 20 Year Marriage

It's August in Northern Virginia, hot and humid. I still haven't showered from my morning trail run. I'm wearing my stay-at-home mom...

· 4 min read · Feb 16, 2022

64K 943

Josh Weinstein in Python in Plain English

A Simple Guide to Shared Memory in Python

Python is an incredible programming language. Python is easy to learn, and you can do just about anything with it. However, it has some...

5 min read · Mar 28

1

See more recommendations