Engineer

UTF-8

T↕ Unicode encoding with one to four bytes per scalar value

What it is

UTF-8 is the encoding that turns Unicode text into bytes. The Unicode Standard defines it as the encoding form that stores each Unicode scalar value, which is any code point except the surrogates set aside for UTF-16, as a sequence of one to four bytes. A user-perceived character can contain several scalar values and therefore need more bytes. It is one of the standard's three encoding forms, beside UTF-16 and UTF-32, and the one the web's own standards require. As a label, in an HTML page or an HTTP header, it is usually written in lower case, utf-8, and sometimes without the hyphen, utf8; the Encoding Standard accepts both and ignores case in labels.

Ken Thompson designed UTF-8 in September 1992 for the Plan 9 operating system, working to criteria that Rob Pike had laid down, and on its way through standardisation the encoding was called FSS-UTF and then UTF-2 before it got its present name. RFC 3629, the IETF's definition from 2003, limits it to the code points U+0000 to U+10FFFF, the same codespace Unicode has. When the RFC was written, the description of UTF-8 in ISO/IEC 10646 still allowed sequences of up to six bytes; under the RFC and the Unicode Standard, no Unicode scalar value takes more than four.

The leading bits of every byte say what kind of byte it is. A byte whose top bit is 0 is a character on its own, one of the 128 characters of ASCII. A byte that begins with 110, 1110 or 11110 starts a sequence of two, three or four bytes, and each byte that follows begins with 10 and carries six more bits of the code point. The whole format fits in four rows, with x standing for the bits of the code point:

```
U+0000..U+007F     0xxxxxxx
U+0080..U+07FF     110xxxxx 10xxxxxx
U+0800..U+FFFF     1110xxxx 10xxxxxx 10xxxxxx
```

```
U+10000..U+10FFFF  11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The three-byte row leaves out the surrogates, U+D800 to U+DFFF: they are not characters, and any byte sequence that would encode one is ill-formed.

Four properties follow from that shape, and the Unicode Standard names each of them. UTF-8 is transparent to ASCII: the bytes 00 to 7F mean what they mean in ASCII and never occur inside another character. It is self-synchronising: a lead byte is easy to tell from a continuation byte. Sorting UTF-8 strings byte by byte puts them in the same order as sorting them by code point. And it has no byte order, so the question of big-endian or little-endian order that UTF-16 and UTF-32 have to answer never arises.

Use the concept in a decision

Put “What is UTF-8?” to work in a real decision. We can help relate the concept to the system you are building or buying. Software that reads naturally in every market you sell in, including the dates, the currencies and the way names sort.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built an English/Danish i18n system — 12,027 messages a locale, in English and Danish

Built an English/Danish i18n system Get in touch →

Why it matters

The web has settled on it. The WHATWG Encoding Standard, the specification browsers implement for reading and writing text, identifies UTF-8 as the preferred way to exchange Unicode text and requires it for new protocols and formats, and for existing formats deployed in new contexts. It tells authors to use UTF-8 and to identify it by the label utf-8. HTML follows suit: the encoding of a document must be UTF-8 whether or not the document declares it, and authoring tools must default to UTF-8 for new documents. A product that publishes pages, sends JSON or accepts form posts is already expected to speak it.

The design explains why that settlement held. Because the ASCII bytes never appear inside another character, a program written for ASCII keeps working on UTF-8 text: a parser that splits on commas, slashes or line breaks finds them in a Danish or Ukrainian sentence exactly where they are, and never in the middle of an å or a ľ. Because a program dropped at any position finds the start of an

encoded scalar value by searching at most four bytes backward, and usually fewer, decoding can resume at a following well-formed sequence instead of losing the rest of the file. That does not promise exactly one replacement character for every damaged byte. A search for one complete encoded scalar value never matches part of another. Because the binary order is the code point order, a database or a program can sort UTF-8 bytes without decoding them. That order is a technical one, though: it puts å before æ and ø, where a Danish dictionary ends the alphabet with æ, ø and å.

Size is the trade. The Unicode Standard calls UTF-8 reasonably compact: for ASCII syntax and Western languages it takes much less space than UTF-16, while text in Asian scripts such as Hindi, Thai, Chinese, Japanese and Korean comes out significantly bigger. In the basic Danish alphabet, letters take one byte apart from æ, ø and å, which take two; other accented letters can need more. Ukrainian takes two bytes for every Cyrillic letter and one for ASCII spaces, digits and punctuation, but other punctuation can need more. A field measured in bytes therefore holds about half as many Ukrainian letters as English ones.

What it looks like in practice

The bytes are easy to see. Node.js, like a browser, has the Encoding Standard's `TextEncoder`, which writes a string as UTF-8:

```
const hex = (bytes) =>
  [...bytes].map((b) => b.toString(16).padStart(2, "0")).join(" ");
const encoder = new TextEncoder();
for (const text of ["A", "å", "Ґ", "€", "𐌂"]) {
  console.log(text, hex(encoder.encode(text)).toUpperCase());
}
```

Run in Node.js 25, it prints each character and its bytes in hexadecimal:

```
A 41
å C3 A5
Ґ D2 90
€ E2 82 AC
𐌂 F0 90 8C 82
```

The letter A is ASCII and keeps its single byte. The Danish å is U+00E5, which needs eight bits, one more than the seven a single byte offers, so it takes the two-byte pattern. Its eleven-bit value, 00011100101, is split into 00011 and 100101; the lead byte becomes 110 followed by 00011, which is C3, and the continuation byte becomes 10 followed by 100101, which is A5. The Ukrainian ґ, U+0490, takes two bytes the same way, the euro sign three, and the Old Italic letter 𐌂, beyond U+FFFF, four. The first byte of each sequence announces its length: C3 and D2 begin with 110, E2 with 1110 and F0 with 11110.

This site writes every page in UTF-8 and says so where HTML asks for it. The first element in the head of every page is `<meta charset="utf-8">`, with the label utf-8 that the Encoding Standard prescribes, and well inside the first 1,024 bytes, where HTML requires the declaration to sit. The credits page has a section of outside checks, none of them ours, each running a live test instead of taking our word for anything. One of them is the W3C Internationalization Checker, which reads the language and encoding declarations that make every language version of this site legible to the right reader.

Where it goes wrong

A byte limit cuts a character in half. A column, a message field or a buffer measured in bytes can end in the middle of a sequence. The name Ærø is three characters and five bytes, and a limit of four keeps Æ, r and the first byte of ø:

```
const bytes = new TextEncoder().encode("Ærø");
const cut = new TextDecoder().decode(bytes.slice(0, 4));
console.log(bytes.length, [...cut].map((c) => c.codePointAt(0).toString(16)));
console.log(new TextEncoder().encodeInto("Ærø", new Uint8Array(4)));
```

```
5 [ 'c6', '72', 'fffd' ]
{ read: 2, written: 3 }
```

The orphaned lead byte decodes as U+FFFD, the replacement character, so the stored name ends in a placeholder where the ø was. Cut on a Unicode scalar-value boundary instead. `encodeInto`, on the last line, does that by itself: given four bytes of room, it writes only the three bytes of Æ and r, and reports in `read` how much of the string it took, counted in UTF-16 units. Code that cuts bytes it already holds can step back over the continuation bytes, the ones that

begin with 10, to the lead byte, and cut there. That keeps the UTF-8 well-formed; preserving a whole user-perceived character also requires respecting grapheme-cluster boundaries.

A byte order mark starts the file. UTF-8 has no byte order, so it needs no mark, but a converter that turns UTF-16 with a byte order mark into UTF-8 turns the mark into the three bytes EF BB BF. The Unicode Standard does not require them, says their presence does not affect conformance, and does not recommend them as a signature for UTF-8. Readers disagree about them. Here the file bom.json holds those three bytes followed by `{"name":"Ærø"}` :

```
import { readFileSync } from "node:fs";

const text = readFileSync("bom.json", "utf8");
console.log(text.codePointAt(0).toString(16));
try {
  JSON.parse(text);
} catch (error) {
  console.log(error.name);
}
console.log(JSON.parse(new TextDecoder().decode(readFileSync("bom.json"))));
```

```
feff
SyntaxError
{ name: 'Ærø' }
```

Node’s `readFileSync` keeps the mark as the character U+FEFF, and `JSON.parse` rejects the text because of it. `TextDecoder` follows the Encoding Standard, removes the mark, and the same bytes parse cleanly. In a browser the mark goes further: the Encoding Standard ranks it above anything else, the HTTP `Content-Type` header included. Write UTF-8 without the mark, and where files arrive from other systems, decode them with a decoder that strips it, or strip a leading U+FEFF before parsing.

A check reads the bytes before they are decoded. RFC 3629 describes a parser that checks its input in encoded form and refuses the bytes 2F 2E 2E 2F, the path step “/../”, yet lets through 2F C0 AE 2E 2F, in which C0 AE is an illegal two-byte spelling of the full stop that the parser goes on to read as one. The RFC records that a virus which spread widely among web servers in 2001 used exactly this trick. Since Unicode 3.1 such non-shortest forms are ill-formed, and the bytes C0,

C1 and F5 to FF never occur in valid UTF-8. A conforming decoder, `TextDecoder` among them, turns those five bytes into a slash, two U+FFFD and "./", so no "/../" is left to slip past. The Unicode Standard is just as strict about what follows an error: a decoder that swallows the valid bytes after a bad lead byte is non-conformant and open to security exploits, and `TextDecoder` turns the bytes C2 41 42 into U+FFFD followed by AB. Decode first with the platform's own decoder, then check the decoded text, and never write a UTF-8 decoder, or a byte pattern that stands in for one, by hand.

Replacement hides the damage. By default a decoder replaces every bad sequence with U+FFFD and carries on. That is right for showing a page and wrong for importing data: the text reads almost normally, the stored copy has lost the original bytes, and the sender never learns where the file was damaged. The Encoding Standard names two error modes for decoding, "replacement" and "fatal", and for identifiers and byte sequences inside a format or protocol it lets other standards pick a decoder that fails instead of replacing. In JavaScript, `new TextDecoder("utf-8", { fatal: true })` throws a `TypeError` for the cut name above instead of returning it. Use the fatal mode wherever text is stored, signed or compared, reject the input with a message the sender can act on, and keep replacement for display.

The declaration is late, missing or wrong. HTML requires the element that declares the encoding to be complete within the first 1,024 bytes of the document. It requires a declaration even when every character is ASCII, because the browser needs an encoding for what people type into forms and for URLs that scripts build. A label that looks harmless can also be the wrong one: in the Encoding Standard, "ascii", "latin1" and "iso-8859-1" are all labels of windows-1252, so a page declared as ASCII is read as windows-1252. Put `<meta charset="utf-8">` first in the head, send `Content-Type: text/html; charset=utf-8` with every page, and treat any other label in either place as a defect.

Further reading

- The Unicode Standard, Chapter 3: Conformance — the definition of UTF-8, the table of well-formed byte sequences, the byte order mark, and how a converter must treat ill-formed input.
- The Unicode Standard, Chapter 2: General Structure — why UTF-8 is transparent to ASCII, self-synchronising and free of byte order, and how its size compares with UTF-16.

- RFC 3629: UTF-8, a transformation format of ISO 10646 — the IETF definition, its history from Plan 9, and the security considerations behind the overlong path attack.
- Encoding Standard — the WHATWG specification browsers follow: why UTF-8 is required, how decoding treats a byte order mark and errors, and the `TextEncoder` and `TextDecoder` interfaces.
- HTML: Specifying the document's character encoding — what a declaration must say, where it must appear, and why a page written only in ASCII needs one.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how "What is UTF-8?" applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/utf-8/