Engineer

UTF-8

🔤 Unicode-kodning med én til fire byte pr. skalarværdi

Hvad det er

UTF-8 er den kodning, der gør Unicode-tekst til bytes. Unicode-standarden definerer den som den kodningsform, der gemmer hver Unicode-skalarværdi, det vil sige ethvert kodepunkt undtagen de surrogater, der er sat af til UTF-16, som en sekvens på én til fire bytes. Et tegn, som brugeren opfatter som ét, kan bestå af flere skalarværdier og derfor kræve flere bytes. Den er en af standardens tre kodningsformer ved siden af UTF-16 og UTF-32, og den, som nettets egne standarder kræver. Som betegnelse, i en HTML-side eller en HTTP-header, skrives den som regel med små bogstaver, utf-8, og nogle gange uden bindestreg, utf8; Encoding Standard godtager begge og skelner ikke mellem store og små bogstaver i betegnelserne.

UTF-8 blev udtænkt i september 1992 af Ken Thompson efter designkriterier, som Rob Pike havde opstillet, til styresystemet Plan 9, og den hed både FSS-UTF og UTF-2, før den fik sit nuværende navn. RFC 3629, IETF's definition fra 2003, begrænser den til kodepunkterne U+0000 til U+10FFFF, det samme koderum, som Unicode har. Da RFC'en blev skrevet, tillod beskrivelsen af UTF-8 i ISO/IEC 10646 stadig sekvenser på op til seks bytes; efter RFC'en og Unicode-standarden fylder ingen Unicode-skalarværdi mere end fire.

De forreste bit i hver byte fortæller, hvilken slags byte det er. En byte, hvis øverste bit er 0, er et tegn for sig selv, et af ASCII's 128 tegn. En byte, der begynder med 110, 1110 eller 11110, indleder en sekvens på to, tre eller fire bytes, og hver af de efterfølgende bytes begynder med 10 og bærer seks bit mere af kodepunktet. Hele formatet kan stå i fire rækker, hvor x står for kodepunktets bit:

```
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
```

Rækken med tre bytes udelader surrogaterne, U+D800 til U+DFFF: de er ikke tegn, og enhver bytesekvens, der ville kode et af dem, er ugyldig.

Fire egenskaber følger af den form, og Unicode-standarden nævner hver af dem ved navn. UTF-8 er gennemsigtig for ASCII: bytes fra 00 til 7F betyder det samme som i ASCII og optræder aldrig inde i et andet tegn. Den er selvsynkroniserende: en startbyte er let at skelne fra en fortsættelsesbyte. En binær sortering af UTF-8-strenge giver samme rækkefølge som en sortering efter kodepunkt. Og den har ingen byterækkefølge, så spørgsmålet om big-endian eller little-endian, som UTF-16 og UTF-32 må tage stilling til, opstår aldrig.

Brug begrebet i en beslutning

Brug »Hvad er UTF-8?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. Software, der læses naturligt på hvert marked, I sælger på — også datoer, valutaer og den måde, navne sorteres på.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. Har bygget et engelsk/dansk i18n-system — 12,027 beskeder pr. sprog, på engelsk og dansk

Har bygget et engelsk/dansk i18n-system Kontakt os →

Hvorfor det betyder noget

Nettet har lagt sig fast på den. WHATWG's Encoding Standard, den specifikation, browsere implementerer for at læse og skrive tekst, kalder UTF-8 den mest velegnede kodning til udveksling af Unicode og kræver den til nye protokoller og formater og til eksisterende formater, der tages i brug i nye sammenhænge. Den beder forfattere om at bruge UTF-8 og om at angive den med betegnelsen utf-8. HTML følger trop: et dokuments kodning skal være UTF-8, uanset om dokumentet angiver det eller ej, og redigeringsværktøjer skal som standard bruge UTF-8 til nye dokumenter. Et produkt, der udgiver sider, sender JSON eller tager imod formularer, forventes allerede at tale den.

Designet forklarer, hvorfor den beslutning har holdt. Fordi ASCII-bytes aldrig optræder inde i et andet tegn, virker et program skrevet til ASCII stadig på UTF-8-tekst: en parser, der deler ved kommaer, skråstreger eller linjeskift, finder dem i en dansk eller ukrainsk sætning præcis der, hvor de står, og aldrig midt i et å eller et ґ. Fordi et program, der lander et vilkårligt sted, finder starten på en kodet skalarværdi ved at søge højst fire bytes bagud, og som regel færre, kan afkodningen fortsætte ved en efterfølgende velformet sekvens i stedet for at miste resten af filen. Det lover ikke præcis ét erstatningstegn for hver beskadiget byte. En søgning efter én komplet kodet skalarværdi rammer aldrig en del af en anden. Fordi den binære rækkefølge er kodepunkternes rækkefølge, kan en database eller et program sortere UTF-8-bytes uden at afkode dem. Den rækkefølge er dog en teknisk rækkefølge: den sætter å før æ og ø, hvor en dansk ordbog slutter alfabetet med æ, ø og å.

Størrelsen er prisen. Unicode-standarden kalder UTF-8 rimelig kompakt: meget mindre end UTF-16 for ASCII-syntaks og vestlige sprog og betydeligt større for asiatiske skriftsystemer som hindi, thai, kinesisk, japansk og koreansk. I det danske grundalfabet koster bogstaver én byte bortset fra æ, ø og å, der fylder to; andre bogstaver med accent kan kræve flere. Ukrainsk fylder to bytes for hvert kyrillisk bogstav og én for ASCII-mellemrum, cifre og tegnsætning, men anden tegnsætning kan kræve flere. Derfor rummer et felt, der måles i bytes, omtrent halvt så mange ukrainske bogstaver som engelske.

Sådan ser det ud i praksis

De enkelte bytes er lette at få frem. Node.js har, ligesom en browser, Encoding Standards `TextEncoder`, der skriver en streng som 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", "€", "𐍈"]) {
  console.log(text, hex(encoder.encode(text)).toUpperCase());
}
```

Kørt i Node.js 25 skriver koden hvert tegn og dets bytes i hexadecimal:

```
A 41
å C3 A5
ґ D2 90
```

```
€ E2 82 AC
𐌏 F0 90 8C 8F
```

Bogstavet A er ASCII og beholder sin ene byte. Det danske å er U+00E5, der kræver otte bit, én mere end de syv, en enkelt byte giver plads til, så det får mønsteret med to bytes. Dets værdi på elleve bit, 00011100101, deles i 00011 og 100101; startbyten bliver 110 efterfulgt af 00011, altså C3, og fortsættelsesbyten bliver 10 efterfulgt af 100101, altså A5. Det ukrainske ԍ, U+0490, fylder to bytes på samme måde, eurotegnet tre og det gammelitaliske bogstav 𐌏, der ligger over U+FFFF, fire. Den første byte i hver sekvens melder sekvensens længde: C3 og D2 begynder med 110, E2 med 1110 og F0 med 11110.

Dette site skriver alle sider i UTF-8 og siger det, hvor HTML beder om det. Det første element i head på hver side er `<meta charset="utf-8">`, med den betegnelse, utf-8, som Encoding Standard foreskriver, og et godt stykke inden for de første 1.024 bytes, hvor HTML kræver, at angivelsen står. Sitets kolofon har et afsnit med tjek udefra, og ingen af dem er vores: hvert af dem kører en live-test i stedet for at tage vores ord for noget. Et af dem er W3C's internationaliseringstjek, der læser de sprog- og tegnsætsangivelser, som gør hver sprogudgave af sitet læselig for den rette læser.

Hvor det går galt

En bytegrænse skærer et tegn over. En kolonne, et beskedfelt eller en buffer, der måles i bytes, kan slutte midt i en sekvens. Navnet Ærø er tre tegn og fem bytes, og en grænse på fire beholder Æ, r og den første byte af 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 }
```

Den enlige startbyte afkodes som U+FFFD, erstatningstegnet, så det gemte navn ender med en pladsholder dér, hvor ø'et stod. Skær i stedet ved en grænse mellem Unicode-skalarværdier. `encodeInto` på sidste linje gør det af sig selv: med fire bytes til rådighed skriver den kun de tre bytes for Æ og r og melder i `read`, hvor meget af strengen den tog, talt i UTF-16-enheder. Kode, der skærer i bytes, den allerede har, kan gå baglæns forbi fortsættelsesbytes, altså dem, der begynder med 10, til startbyten og skære dér. Det holder UTF-8-sekvensen velformet; at bevare et helt tegn, som brugeren opfatter det, kræver også, at grænserne mellem grafemklynger respekteres.

Et byte order mark står forrest i filen. UTF-8 har ingen byterækkefølge og behøver derfor intet mærke, men en konverter, der laver UTF-16 med et byte order mark om til UTF-8, gør mærket til de tre bytes EF BB BF. Unicode-standarden kræver dem ikke, siger, at de ikke har betydning for, om en implementering overholder standarden, og anbefaler dem ikke som signatur for UTF-8. Programmer, der læser filer, behandler dem forskelligt. Her rummer filen bom.json de tre bytes efterfulgt af `{"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' }
```

Nodes `readFileSync` beholder mærket som tegnet U+FEFF, og `JSON.parse` afviser teksten på grund af det. `TextDecoder` følger Encoding Standard og fjerner mærket, og så kan de samme bytes parses uden problemer. I en browser vejer mærket endnu tungere: Encoding Standard sætter det over alt andet, også over HTTP-headeren `Content-Type`. Skriv UTF-8 uden mærket, og hvor filer kommer fra andre systemer, så afkod dem med en dekoder, der fjerner det, eller fjern et indledende U+FEFF før parsningen.

Et tjek læser bytes, før de er afkodet. RFC 3629 beskriver en parser, der tjekker sit input i kodet form og afviser bytes 2F 2E 2E 2F, stitrinnet »/../«, men lukker 2F C0 AE 2E 2F igennem, hvor C0 AE er en ugyldig stavemåde af punktum på to bytes, som parseren alligevel læser som et punktum. RFC'en noterer, at netop det trick blev brugt af en udbredt virus, der angreb webservere i 2001. Siden Unicode 3.1 er den slags former, der ikke er de kortest mulige, ugyldige, og bytes C0, C1 og F5 til FF forekommer aldrig i gyldig UTF-8. En dekoder, der overholder standarden, `TextDecoder` iblandt, gør de fem bytes til en skråstreg, to U+FFFD og »./«, så der ikke er noget »/../« tilbage, der kan smutte forbi. Unicode-standarden er lige så streng med det, der følger efter en fejl: en dekoder, der sluger de gyldige bytes efter en ugyldig startbyte, overholder ikke standarden og kan udnyttes til angreb, og `TextDecoder` gør bytes C2 41 42 til U+FFFD efterfulgt af AB. Afkod først med platformens egen dekoder, tjek derefter den afkodede tekst, og skriv aldrig selv en UTF-8-dekoder, eller et bytemønster, der skal gøre det ud for en.

Erstatning skjuler skaden. Som standard erstatter en dekoder hver ugyldig sekvens med U+FFFD og fortsætter. Det er rigtigt, når en side skal vises, og forkert, når data skal importeres: teksten ser næsten normal ud, den gemte kopi har mistet de oprindelige bytes, og afsenderen får aldrig at vide, hvor filen var beskadiget. Encoding Standard nævner to fejltilstande for afkodning, »replacement« og »fatal«, og for identifikatorer og bytesekvenser inde i et format eller en protokol lader den andre standarder vælge en dekoder, der fejler i stedet for at erstatte. I JavaScript kaster `new TextDecoder("utf-8", { fatal: true })` en `TypeError` for det afklippede navn ovenfor i stedet for at returnere det. Brug den fatale tilstand overalt, hvor tekst gemmes, signeres eller sammenlignes, afvis inputtet med en besked, afsenderen kan handle på, og behold erstatningen til visning.

Angivelsen kommer for sent, mangler eller er forkert. HTML kræver, at det element, der angiver kodningen, er komplet inden for dokumentets første 1.024 bytes. Det kræver en angivelse, selv når alle tegn er ASCII, fordi browseren skal bruge en kodning til det, folk skriver i formularer, og til de URL'er, scripts bygger. En betegnelse, der ser harmløs ud, kan også være den forkerte: i Encoding Standard er »ascii«, »latin1« og »iso-8859-1« alle betegnelser for windows-1252, så en side, der er angivet som ASCII, læses som windows-1252. Sæt `<meta charset="utf-8">` først i head, send `Content-Type: text/html; charset=utf-8` med hver side, og behandl enhver anden betegnelse begge steder som en defekt.

Læs mere

- The Unicode Standard, Chapter 3: Conformance — definitionen af UTF-8, tabellen over gyldige bytesekvenser, byte order mark, og hvordan en konverter skal behandle ugyldigt input.
- The Unicode Standard, Chapter 2: General Structure — hvorfor UTF-8 er gennemsigtig for ASCII, selvsynkroniserende og fri for byterækkefølge, og hvordan størrelsen er i forhold til UTF-16.
- RFC 3629: UTF-8, a transformation format of ISO 10646 — IETF's definition, historien fra Plan 9 og de sikkerhedshensyn, der ligger bag angrebet med overlange kodninger af en sti.
- Encoding Standard — WHATWG's specifikation, som browsere følger: hvorfor UTF-8 er påkrævet, hvordan afkodning behandler byte order mark og fejl, og grænsefladerne `TextEncoder` og `TextDecoder`.
- HTML: Specifying the document's character encoding — hvad en angivelse skal sige, hvor den skal stå, og hvorfor en side, der kun er skrevet i ASCII, har brug for en.

Ordliste

Vælg ud fra jeres faktiske vilkår

Beskriv arbejdsgangen, de eksisterende grænseflader og de brugere, der har brug for ændringen. Vi kan definere en testbar leverance og en overdragelse, der gør teamet i stand til at vedligeholde løsningen.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er UTF-8?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

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

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/utf-8/