Engineer

UTF-8

🔤 Unicode-koding med én til fire byte per skalarverdi

Hva det er

UTF-8 er kodingen som gjør Unicode-tekst om til byte. Unicode-standarden definerer den som kodingsformen som lagrer hver Unicode-skalarverdi, det vil si ethvert kodepunkt unntatt surrogatene som er satt av til UTF-16, som en sekvens på én til fire byte. Et tegn som brukeren oppfatter som ett, kan inneholde flere skalarverdier og derfor trenge flere byte. Den er én av standardens tre kodingsformer, ved siden av UTF-16 og UTF-32, og den som nettets egne standarder krever. Som betegnelse, i en HTML-side eller en HTTP-header, skrives den som regel med små bokstaver, utf-8, og noen ganger uten bindestrek, utf8; Encoding Standard godtar begge og skiller ikke mellom store og små bokstaver i betegnelsene.

UTF-8 ble utformet i september 1992 av Ken Thompson, etter designkriterier Rob Pike hadde satt opp, for operativsystemet Plan 9, og den het både FSS-UTF og UTF-2 før den fikk navnet den har i dag. RFC 3629, IETFs definisjon fra 2003, begrenser den til kodepunktene U+0000 til U+10FFFF, det samme koderommet som Unicode har. Da RFC-en ble skrevet, tillot beskrivelsen av UTF-8 i ISO/IEC 10646 fortsatt sekvenser på opptil seks byte; etter RFC-en og Unicode-standarden tar ingen Unicode-skalarverdi mer enn fire.

De første bitene i hver byte forteller hva slags byte det er. En byte der den øverste biten er 0, er et tegn for seg selv, ett av de 128 tegnene i ASCII. En byte som begynner med 110, 1110 eller 11110, innleder en sekvens på to, tre eller fire byte, og hver byte som følger, begynner med 10 og bærer seks biter til av kodepunktet. Hele formatet får plass i fire rader, der x står for kodepunktets biter:

```
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
```

Raden med tre byte utelater surrogatene, U+D800 til U+DFFF: de er ikke tegn, og enhver bytesekvens som ville kode et av dem, er ugyldig.

Fire egenskaper følger av den formen, og Unicode-standarden gir hver av dem et navn. UTF-8 er gjennomsiktig for ASCII: bytene 00 til 7F betyr det samme som i ASCII og forekommer aldri inne i et annet tegn. Den er selvsynkroniserende: en startbyte er lett å skille fra en fortsettelsesbyte. En binær sortering av UTF-8-strenger gir samme rekkefølge som en sortering etter kodepunkt. Og den har ingen byterekkefølge, så spørsmålet om big-endian eller little-endian, som UTF-16 og UTF-32 må ta stilling til, oppstår aldri.

Bruk begrepet i en beslutning

Bruk «Hva er UTF-8?» i en konkret beslutning. Vi kan knytte begrepet til systemet dere bygger eller kjøper. Programvare som leses naturlig i alle markeder du selger i, også datoene, valutaene og måten navn sorteres på.

Se tjenestene våre Ta kontakt →

Undersøk antakelsene bak ordet

Bruk forklaringen og det tilknyttede arbeidet til å formulere spørsmål for prosjektet. Vi kan undersøke antakelsene sammen før et verktøy velges. Bygde et engelsk/dansk i18n-system — 12,027 meldinger per språkversjon, på engelsk og dansk

Bygde et engelsk/dansk i18n-system Ta kontakt →

Hvorfor det er viktig

Nettet har bestemt seg for den. WHATWGs Encoding Standard, spesifikasjonen nettlesere implementerer for å lese og skrive tekst, kaller UTF-8 den best egnede kodingen for utveksling av Unicode og krever den for nye protokoller og formater, og for eksisterende formater som tas i bruk i nye sammenhenger. Den ber forfattere bruke UTF-8 og angi den med betegnelsen utf-8. HTML følger etter: kodingen i et dokument må være UTF-8 enten dokumentet oppgir det eller ikke, og redigeringsverktøy skal bruke UTF-8 som standard for nye dokumenter. Et produkt som publiserer sider, sender JSON eller tar imot skjemaer, forventes allerede å snakke den.

Utformingen forklarer hvorfor det valget har holdt. Fordi ASCII-bytene aldri forekommer inne i et annet tegn, fungerer et program skrevet for ASCII fortsatt på UTF-8-tekst: en parser som deler ved komma, skråstrek eller linjeskift, finner dem i en norsk eller ukrainsk setning nøyaktig der de står, og aldri midt i en å eller en ґ. Fordi et program som havner et vilkårlig sted, finner starten på en kodet skalarverdi ved å søke høyst fire byte bakover, og som regel færre, kan dekodingen fortsette ved en etterfølgende velformet sekvens i stedet for å miste resten av filen. Det lover ikke nøyaktig ett erstatningstegn for hver skadet byte. Et søk etter én fullstendig kodet skalarverdi treffer aldri en del av en annen. Fordi den binære rekkefølgen er kodepunktenes rekkefølge, kan en database eller et program sortere UTF-8-byte uten å dekode dem. Den rekkefølgen er likevel en teknisk rekkefølge: den setter å foran æ og ø, mens en norsk ordbok avslutter alfabetet med æ, ø og å.

Størrelsen er prisen. Unicode-standarden kaller UTF-8 rimelig kompakt: mye mindre enn UTF-16 for ASCII-syntaks og vestlige språk, og betydelig større for asiatiske skriftsystemer som hindi, thai, kinesisk, japansk og koreansk. I det norske grunnalfabetet tar bokstaver én byte bortsett fra æ, ø og å, som tar to; andre bokstaver med aksent kan trenge flere. Ukrainsk tar to byte for hver kyrillisk bokstav og én for ASCII-mellomrom, sifre og tegnsetting, men annen tegnsetting kan trenge flere. Derfor rommer et felt som måles i byte omtrent halvparten så mange ukrainske bokstaver som engelske.

Slik ser det ut i praksis

Bytene er lette å få fram. Node.js har, i likhet med en nettleser, Encoding Standards `TextEncoder`, som 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());
}
```

Kjørt i Node.js 25 skriver koden ut hvert tegn og bytene i heksadesimal form:

```
A 41
å C3 A5
ґ D2 90
```

```
€ E2 82 AC
𐌀 F0 90 8C 82
```

Bokstaven A er ASCII og beholder sin ene byte. Den norske å er U+00E5, som trenger åtte biter, én mer enn de sju én enkelt byte gir plass til, så den får mønsteret med to byte. Verdien på elleve biter, 00011100101, deles i 00011 og 100101; startbyten blir 110 fulgt av 00011, altså C3, og fortsettelsesbyten blir 10 fulgt av 100101, altså A5. Den ukrainske ґ, U+0490, tar to byte på samme måte, eurotegnet tre og den gammelitaliske bokstaven 𐌀, som ligger over U+FFFF, fire. Den første byten i hver sekvens forteller hvor lang sekvensen er: C3 og D2 begynner med 110, E2 med 1110 og F0 med 11110.

Dette nettstedet skriver alle sider i UTF-8 og sier fra der HTML ber om det. Det første elementet i head på hver side er `<meta charset="utf-8">`, med betegnelsen utf-8, som Encoding Standard foreskriver, og godt innenfor de første 1024 bytene, der HTML krever at deklarasjonen skal stå. Nettstedets kolofon har en del med kontroller utenfra, og ingen av dem er våre: hver av dem kjører en direkte test i stedet for å ta oss på ordet. En av dem er W3C Internationalization Checker, som leser språk- og tegnkodingsdeklarasjonene som gjør hver språkversjon av nettstedet lesbar for den rette leseren.

Hvor det går galt

En bytegrense kutter et tegn i to. En kolonne, et meldingsfelt eller en buffer som måles i byte, kan slutte midt i en sekvens. Navnet Ærø er tre tegn og fem byte, og en grense på fire beholder Æ, r og den første byten av 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 enslige startbyten dekodes som U+FFFD, erstatningstegnet, så det lagrede navnet slutter med en plassholder der ø-en stod. Kutt heller på en grense mellom Unicode-skalarverdier. `encodeInto` på siste linje gjør det av seg selv: med fire

byte til rådighet skriver den bare de tre bytene for Æ og r, og melder i `read` hvor mye av strengen den tok, talt i UTF-16-enheter. Kode som kutter i byte den allerede har, kan gå bakover forbi fortsettelsesbytene, de som begynner med 10, til startbyten og kutte der. Det holder UTF-8-sekvensen velformet; å bevare et helt tegn slik brukeren oppfatter det, krever også at grensene mellom grafemklynger respekteres.

Et byte order mark står først i filen. UTF-8 har ingen byterekkefølge og trenger derfor ikke noe merke, men en konverterer som gjør UTF-16 med et byte order mark om til UTF-8, gjør merket til de tre bytene EF BB BF. Unicode-standarden krever dem ikke, sier at de ikke har noe å si for om en implementasjon er i samsvar med standarden, og anbefaler dem ikke som signatur for UTF-8. Programmer som leser filer, behandler dem ulikt. Her inneholder filen bom.json de tre bytene fulgt av `{"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 merket som tegnet U+FEFF, og `JSON.parse` avviser teksten på grunn av det. `TextDecoder` følger Encoding Standard og fjerner merket, og da lar de samme bytene seg parse uten problemer. I en nettleser veier merket enda tyngre: Encoding Standard setter det over alt annet, også over HTTP-headeren `Content-Type` . Skriv UTF-8 uten merket, og der filer kommer fra andre systemer, dekod dem med en dekoder som fjerner det, eller fjern en innledende U+FEFF før parsingen.

En kontroll leser bytene før de er dekodet. RFC 3629 beskriver en parser som kontrollerer inndataene i kodet form og avviser bytene 2F 2E 2E 2F, stisteget «/../», men slipper gjennom 2F C0 AE 2E 2F, der C0 AE er en ulovlig skrivemåte for punktum på to byte, som parseren likevel leser som et punktum. RFC-en forteller at akkurat dette trikset ble brukt av et utbredt virus som angrep webservere i 2001. Siden Unicode 3.1 er slike former, som ikke er de kortest mulige, ugyldige, og bytene C0, C1 og F5 til FF forekommer aldri i gyldig UTF-8. En dekoder som følger standarden, `TextDecoder` blant dem, gjør de fem bytene om til en skråstrek, to U+FFFD og «./», så det ikke er noen «/../» igjen som kan smette forbi. Unicode-standarden er like streng med det som kommer etter en feil: en dekoder som sluker de gyldige bytene etter en ugyldig startbyte, følger ikke standarden og kan utnyttes til angrep, og `TextDecoder` gjør bytene C2 41 42 om til U+FFFD fulgt av AB. Dekod først med plattformens egen dekoder, kontroller deretter den dekodede teksten, og skriv aldri en UTF-8-dekoder, eller et bytemønster som skal gjøre samme nytten, for hånd.

Erstatning skjuler skaden. Som standard erstatter en dekoder hver ugyldig sekvens med U+FFFD og fortsetter. Det er riktig når en side skal vises, og feil når data skal importeres: teksten ser nesten normal ut, den lagrede kopien har mistet de opprinnelige bytene, og avsenderen får aldri vite hvor filen var skadet. Encoding Standard nevner to feilmoduser for dekoding, «replacement» og «fatal», og for identifikatorer og bytesekvenser inne i et format eller en protokoll lar den andre standarder velge en dekoder som feiler i stedet for å erstatte. I JavaScript kaster `new TextDecoder("utf-8", { fatal: true })` en `TypeError` for det avkuttede navnet ovenfor i stedet for å returnere det. Bruk den fatale modusen overalt der tekst lagres, signeres eller sammenlignes, avvis inndataene med en melding avsenderen kan gjøre noe med, og behold erstatningen til visning.

Deklarasjonen kommer for sent, mangler eller er feil. HTML krever at elementet som deklarerer kodingen, er fullstendig innenfor de første 1024 bytene i dokumentet. Det krever en deklarasjon selv når alle tegnene er ASCII, fordi nettleseren trenger en koding for det folk skriver inn i skjemaer, og for URL-er som skript bygger. En betegnelse som ser harmløs ut, kan også være feil: i Encoding Standard er «ascii», «latin1» og «iso-8859-1» alle betegnelser på windows-1252, så en side som er deklarert som ASCII, leses som windows-1252. Sett `<meta charset="utf-8">` først i head, send `Content-Type: text/html; charset=utf-8` med hver side, og behandle enhver annen betegnelse på noen av stedene som en feil.

Les mer

- The Unicode Standard, Chapter 3: Conformance — definisjonen av UTF-8, tabellen over gyldige bytesekvenser, byte order mark, og hvordan en konverterer skal behandle ugyldige inndata.
- The Unicode Standard, Chapter 2: General Structure — hvorfor UTF-8 er gjennomsiktig for ASCII, selvsynkroniserende og fri for byterekkefølge, og hvordan størrelsen står seg mot UTF-16.
- RFC 3629: UTF-8, a transformation format of ISO 10646 — IETFs definisjon, historien fra Plan 9 og sikkerhetshensynene bak angrepet med overlange kodinger av en sti.
- Encoding Standard — WHATWGs spesifikasjon, som nettlesere følger: hvorfor UTF-8 er påkrevd, hvordan dekoding behandler byte order mark og feil, og grensesnittene `TextEncoder` og `TextDecoder`.
- HTML: Specifying the document's character encoding — hva en deklarasjon må si, hvor den må stå, og hvorfor en side som bare er skrevet i ASCII, trenger en.

Ordliste

Velg ut fra deres faktiske rammer

Beskriv arbeidsflyten, eksisterende grensesnitt og brukerne som trenger endringen. Vi kan definere en testbar leveranse og en overlevering som gjør teamet i stand til å vedlikeholde løsningen.

Se hva det koster Ta kontakt →

Gjør begrepene praktiske

Spør hvordan «Hva er UTF-8?» gjelder i organisasjonen deres. Et konkret eksempel fra dagens system er nok til å starte samtalen.

Se tjenestene våre

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

Ta med deg denne siden

Visittkort

Laget med kjærlighet av D.

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

Opphavsrett © 2025 – nå · Engineer ApS · Alle rettigheter forbeholdt.

Skrevet ut fra https://engineer.company/nb/glossary/utf-8/