Engineer

UTF-8

🔤 Unicode-kodning med en till fyra byte per skalärvärde

Vad det är

UTF-8 är kodningen som gör om Unicode-text till byte. Unicode-standarden definierar den som den kodningsform som lagrar varje Unicode-skalärvärde, det vill säga varje kodpunkt utom de surrogat som är reserverade för UTF-16, som en sekvens på en till fyra byte. Ett tecken som användaren uppfattar som ett kan innehålla flera skalärvärden och därför behöva fler byte. Den är en av standardens tre kodningsformer, vid sidan av UTF-16 och UTF-32, och den som webbens egna standarder kräver. Som beteckning, i en HTML-sida eller en HTTP-header, skrivs den oftast med gemener, utf-8, och ibland utan bindestreck, utf8; Encoding Standard godtar båda och skiljer inte mellan versaler och gemener i beteckningar.

UTF-8 togs fram i september 1992 av Ken Thompson, efter designkriterier som Rob Pike hade ställt upp, för operativsystemet Plan 9, och den gick under namnen FSS-UTF och UTF-2 innan den fick sitt nuvarande namn. RFC 3629, IETF:s definition från 2003, begränsar den till kodpunkterna U+0000 till U+10FFFF, samma kodrymd som Unicode har. När RFC:n skrevs tillät beskrivningen av UTF-8 i ISO/IEC 10646 fortfarande sekvenser på upp till sex byte; enligt RFC:n och Unicode-standarden tar inget Unicode-skalärvärde mer än fyra.

De första bitarna i varje byte talar om vilken sorts byte det är. En byte vars högsta bit är 0 är ett tecken i sig, ett av de 128 tecknen i ASCII. En byte som börjar med 110, 1110 eller 11110 inleder en sekvens på två, tre eller fyra byte, och varje byte som följer börjar med 10 och bär ytterligare sex bitar av kodpunkten. Hela formatet ryms på fyra rader, där x står för kodpunktens bitar:

```
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 utelämnar surrogaten, U+D800 till U+DFFF: de är inte tecken, och varje bytesekvens som skulle koda ett av dem är ogiltig.

Fyra egenskaper följer av den formen, och Unicode-standarden ger var och en av dem ett namn. UTF-8 är transparent för ASCII: byte från 00 till 7F betyder samma sak som i ASCII och förekommer aldrig inuti ett annat tecken. Den är självsynkroniserande: en startbyte är lätt att skilja från en fortsättningsbyte. En binär sortering av UTF-8-strängar ger samma ordning som en sortering efter kodpunkt. Och den saknar byteordning, så frågan om big-endian eller little-endian, som UTF-16 och UTF-32 måste besvara, uppstår aldrig.

Använd begreppet i ett beslut

Använd ”Vad är UTF-8?” i ett konkret beslut. Vi kan knyta begreppet till systemet ni bygger eller köper. Mjukvara som känns naturlig på varje marknad du säljer på, inklusive datum, valutor och hur namn sorteras.

Se våra tjänster Hör av er →

Granska antagandena bakom ordet

Använd förklaringen och det länkade arbetet för att formulera frågor till projektet. Vi kan undersöka antagandena tillsammans innan ett verktyg väljs. Byggde ett i18n-system för engelska och danska — 12,027 meddelanden per språkversion, på engelska och danska

Byggde ett i18n-system för engelska och danska Hör av er →

Varför det spelar roll

Webben har bestämt sig för den. WHATWG:s Encoding Standard, specifikationen som webbläsare implementerar för att läsa och skriva text, kallar UTF-8 den lämpligaste kodningen för utbyte av Unicode och kräver den för nya protokoll och format, och för befintliga format som tas i bruk i nya sammanhang. Den uppmanar författare att använda UTF-8 och att ange den med beteckningen utf-8. HTML följer efter: kodningen i ett dokument måste vara UTF-8 oavsett om dokumentet anger det eller inte, och redigeringsverktyg ska använda UTF-8 som standard för nya dokument. En produkt som publicerar sidor, skickar JSON eller tar emot formulär förväntas redan tala den.

Utformningen förklarar varför det valet har hållit. Eftersom ASCII-byten aldrig förekommer inuti ett annat tecken fungerar ett program skrivet för ASCII fortfarande på UTF-8-text: en parser som delar vid kommatecken, snedstreck eller radbrytningar hittar dem i en svensk eller ukrainsk mening precis där de står, och aldrig mitt i ett å eller ett ґ. Eftersom ett program som hamnar på en

godtycklig position hittar början på ett kodat skalärvärde genom att söka högst fyra byte bakåt, och oftast färre, kan avkodningen fortsätta vid en efterföljande korrekt sekvens i stället för att förlora resten av filen. Det lovar inte exakt ett ersättningstecken för varje skadad byte. En sökning efter ett fullständigt kodat skalärvärde träffar aldrig en del av ett annat. Eftersom den binära ordningen är kodpunkternas ordning kan en databas eller ett program sortera UTF-8-byte utan att avkoda dem. Den ordningen är ändå en teknisk ordning: den sätter ä före å, där en svensk ordbok avslutar alfabetet med å, ä och ö.

Storleken är priset. Unicode-standarden kallar UTF-8 rimligt kompakt: mycket mindre än UTF-16 för ASCII-syntax och västerländska språk, och betydligt större för asiatiska skriftsystem som hindi, thai, kinesiska, japanska och koreanska. I det svenska grundalfabetet tar bokstäver en byte utom å, ä och ö, som tar två; andra bokstäver med accent kan behöva fler. Ukrainska tar två byte för varje kyrillisk bokstav och en för ASCII-mellanslag, siffror och skiljetecken, men andra skiljetecken kan behöva fler. Därför rymmer ett fält som mäts i byte ungefär hälften så många ukrainska bokstäver som engelska.

Så ser det ut i praktiken

Bytena är lätta att ta fram. Node.js har, precis som en webbläsare, Encoding Standards `TextEncoder`, som skriver en sträng 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", "€", "𐌂"]) {
  console.log(text, hex(encoder.encode(text)).toUpperCase());
}
```

Kört i Node.js 25 skriver koden ut varje tecken och dess byte i hexadecimal form:

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

Bokstaven A är ASCII och behåller sin enda byte. Bokstaven å är U+00E5, som behöver åtta bitar, en mer än de sju som en ensam byte har plats för, så den får mönstret med två byte. Dess värde på elva bitar, 00011100101, delas i 00011 och 100101; startbyten blir 110 följt av 00011, alltså C3, och fortsättningsbyten blir 10 följt av 100101, alltså A5. Det ukrainska ґ, U+0490, tar två byte på samma sätt, eurotecknet tre och den fornitaliska bokstaven 𐌢, som ligger över U+FFFF, fyra. Den första byten i varje sekvens talar om hur lång sekvensen är: C3 och D2 börjar med 110, E2 med 1110 och F0 med 11110.

Den här webbplatsen skriver alla sidor i UTF-8 och säger det där HTML ber om det. Det första elementet i head på varje sida är `<meta charset="utf-8">`, med beteckningen utf-8, som Encoding Standard föreskriver, och gott och väl inom de första 1024 byten, där HTML kräver att deklarationen ska stå. Webbplatsens kolofon har ett avsnitt med kontroller utifrån, och ingen av dem är vår: var och en kör ett test i realtid i stället för att ta vårt ord för något. En av dem är W3C Internationalization Checker, som läser de deklarationer av språk och teckenkodning som gör varje språkversion av webbplatsen läsbar för rätt läsare.

Var det går fel

En bytegräns skär ett tecken i två delar. En kolumn, ett meddelandefält eller en buffert som mäts i byte kan sluta mitt i en sekvens. Det danska ortnamnet Ærø är tre tecken och fem byte, och en gräns på fyra behåller Æ, r och den första 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 ensamma startbyten avkodas som U+FFFD, ersättningstecknet, så det lagrade namnet slutar med en platshållare där ø stod. Klipp i stället vid en gräns mellan Unicode-skalärvärden. `encodeInto` på sista raden gör det på egen hand: med fyra byte till förfogande skriver den bara de tre byten för Æ och r, och rapporterar i `read` hur mycket av strängen den tog, räknat i UTF-16-enheter. Kod som klipper byte den redan har kan gå bakåt förbi fortsättningsbyten, de

som börjar med 10, till startbyten och klippa där. Det håller UTF-8-sekvensen korrekt; för att bevara ett helt tecken som användaren uppfattar det måste också gränserna mellan grafemkluster respekteras.

En byteordningsmarkering står först i filen. UTF-8 saknar byteordning och behöver därför ingen markering, men en konverterare som gör om UTF-16 med en byteordningsmarkering till UTF-8 gör markeringen till de tre byten EF BB BF. Unicode-standarden kräver dem inte, säger att de inte påverkar om en implementation följer standarden, och rekommenderar dem inte som signatur för UTF-8. Program som läser filer behandlar dem olika. Här innehåller filen bom.json de tre byten följda 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` behåller markeringen som tecknet U+FEFF, och `JSON.parse` avvisar texten på grund av den. `TextDecoder` följer Encoding Standard och tar bort markeringen, och då går samma byte att tolka utan problem. I en webbläsare väger markeringen ännu tyngre: Encoding Standard sätter den över allt annat, också över HTTP-headern `Content-Type`. Skriv UTF-8 utan markeringen, och där filer kommer från andra system, avkoda dem med en avkodare som tar bort den, eller ta bort ett inledande U+FEFF före tolkningen.

En kontroll läser byten innan de har avkodats. RFC 3629 beskriver en parser som kontrollerar sin indata i kodad form och avvisar byten 2F 2E 2E 2F, sökvägssteget ”/../”, men släpper igenom 2F C0 AE 2E 2F, där C0 AE är en otillåten stavning av punkt på två byte, som parsern ändå läser som en punkt. RFC:n berättar att just det tricket användes av ett utbrett virus som angrep

webbservrar år 2001. Sedan Unicode 3.1 är sådana former, som inte är de kortaste möjliga, ogiltiga, och byten C0, C1 och F5 till FF förekommer aldrig i giltig UTF-8. En avkodare som följer standarden, `TextDecoder` bland dem, gör om de fem byten till ett snedstreck, två U+FFFD och ”./”, så det finns inget ”/../” kvar som kan slinka igenom. Unicode-standarden är lika sträng med det som kommer efter ett fel: en avkodare som sväljer de giltiga byten efter en ogiltig startbyte följer inte standarden och kan utnyttjas för angrepp, och `TextDecoder` gör om byten C2 41 42 till U+FFFD följt av AB. Avkoda först med plattformens egen avkodare, kontrollera sedan den avkodade texten, och skriv aldrig en UTF-8-avkodare, eller ett bytemönster som ska göra samma nytta, för hand.

Ersättningen döljer skadan. Som standard ersätter en avkodare varje ogiltig sekvens med U+FFFD och fortsätter. Det är rätt när en sida ska visas och fel när data ska importeras: texten ser nästan normal ut, den sparade kopian har förlorat de ursprungliga byten, och avsändaren får aldrig veta var filen var skadad. Encoding Standard namnger två fellägen för avkodning, ”replacement” och ”fatal”, och för identifierare och bytesekvenser inuti ett format eller ett protokoll låter den andra standarder välja en avkodare som misslyckas i stället för att ersätta. I JavaScript kastar `new TextDecoder("utf-8", { fatal: true })` ett `TypeError` för det avklippta namnet ovan i stället för att returnera det. Använd det fatala läget överallt där text sparas, signeras eller jämförs, avvisa indata med ett meddelande som avsändaren kan agera på, och behåll ersättningen för visning.

Deklarationen kommer för sent, saknas eller är fel. HTML kräver att elementet som deklarerar kodningen står i sin helhet inom dokumentets första 1024 byte. Det kräver en deklaration även när alla tecken är ASCII, eftersom webbläsaren behöver en kodning för det folk skriver i formulär och för URL:er som skript bygger. En beteckning som ser harmlös ut kan också vara fel: i Encoding Standard är ”ascii”, ”latin1” och ”iso-8859-1” alla beteckningar för windows-1252, så en sida som deklareras som ASCII läses som windows-1252. Sätt `<meta charset="utf-8">` först i head, skicka `Content-Type: text/html; charset=utf-8` med varje sida, och behandla varje annan beteckning på något av ställena som en bugg.

Läs mer

- The Unicode Standard, Chapter 3: Conformance — definitionen av UTF-8, tabellen över giltiga bytesekvenser, byteordningsmarkeringen och hur en konverterare ska hantera ogiltig indata.
- The Unicode Standard, Chapter 2: General Structure — varför UTF-8 är transparent för ASCII, självsynkroniserande och fri från byteordning, och hur storleken står sig mot UTF-16.

- RFC 3629: UTF-8, a transformation format of ISO 10646 — IETF:s definition, historien från Plan 9 och säkerhetsövervägandena bakom angreppet med överlånga kodningar av en sökväg.
- Encoding Standard — WHATWG:s specifikation, som webbläsare följer: varför UTF-8 krävs, hur avkodningen hanterar byteordningsmarkering och fel, och gränssnitten `TextEncoder` och `TextDecoder`.
- HTML: Specifying the document's character encoding — vad en deklaration måste säga, var den måste stå och varför en sida som bara är skriven i ASCII behöver en.

Ordlista

Välj utifrån era faktiska förutsättningar

Beskriv arbetsflödet, befintliga gränssnitt och användarna som behöver förändringen. Vi kan definiera en testbar leverans och en överlämning som gör att teamet kan underhålla lösningen.

Se vad det kostar Hör av er →

Gör begreppen praktiska

Fråga hur ”Vad är UTF-8?” gäller i er organisation. Ett konkret exempel från ert nuvarande system räcker för att börja samtalet.

Se våra tjänster

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

Ta med dig den här sidan

Visitkort

Omsorgsfullt skapad med kärlek av D.

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

Upphovsrätt © 2025 – nu · Engineer ApS · Alla rättigheter förbehållna.

Utskrivet från https://engineer.company/sv/glossary/utf-8/