Engineer

UTF-8

🆃↕ Unicode-Kodierung mit ein bis vier Bytes pro Skalarwert

Was es ist

UTF-8 ist die Kodierung, die Unicode-Text in Bytes umsetzt. Der Unicode-Standard definiert sie als die Kodierungsform, die jeden Unicode-Skalarwert, also jeden Codepunkt außer den für UTF-16 reservierten Surrogaten, als Folge von ein bis vier Bytes speichert. Was ein Benutzer als einzelnes Zeichen wahrnimmt, kann mehrere Skalarwerte enthalten und deshalb mehr Bytes benötigen. Sie ist neben UTF-16 und UTF-32 eine der drei Kodierungsformen des Standards und diejenige, die von den Webstandards verlangt wird. Als Bezeichnung auf einer HTML-Seite oder in einem HTTP-Header wird sie meist kleingeschrieben, utf-8, manchmal auch ohne Bindestrich als utf8. Der Encoding Standard akzeptiert beide und unterscheidet bei Bezeichnungen nicht zwischen Groß- und Kleinschreibung.

Ken Thompson entwarf UTF-8 im September 1992 für das Betriebssystem Plan 9 nach Kriterien, die Rob Pike vorgegeben hatte. Auf dem Weg durch die Standardisierung hieß die Kodierung zunächst FSS-UTF und dann UTF-2, bevor sie ihren heutigen Namen erhielt. RFC 3629, die Definition der IETF von 2003, begrenzt sie auf die Codepunkte U+0000 bis U+10FFFF, denselben Coderaum wie Unicode. Als der RFC geschrieben wurde, erlaubte die Beschreibung von UTF-8 in ISO/IEC 10646 noch Folgen von bis zu sechs Bytes. Nach dem RFC und dem Unicode-Standard benötigt kein Unicode-Skalarwert mehr als vier.

Die führenden Bits jedes Bytes sagen, um welche Art von Byte es sich handelt. Ein Byte mit dem höchsten Bit 0 ist ein eigenständiges Zeichen, eines der 128 ASCII-Zeichen. Ein Byte, das mit 110, 1110 oder 11110 beginnt, eröffnet eine Folge von zwei, drei oder vier Bytes. Jedes folgende Byte beginnt mit 10 und trägt sechs weitere Bits des Codepunkts. Das ganze Format passt in vier Zeilen, wobei x für die Bits des Codepunkts steht:

```
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
```

Die Zeile mit drei Bytes lässt die Surrogate U+D800 bis U+DFFF aus: Sie sind keine Zeichen, und jede Bytefolge, die eines kodieren würde, ist fehlerhaft.

Aus diesem Aufbau folgen vier Eigenschaften, die der Unicode-Standard einzeln benennt. UTF-8 ist für ASCII transparent: Die Bytes 00 bis 7F bedeuten dasselbe wie in ASCII und kommen nie innerhalb eines anderen Zeichens vor. Es ist selbstsynchronisierend: Ein Startbyte lässt sich leicht von einem Fortsetzungsbyte unterscheiden. UTF-8-Zeichenfolgen Byte für Byte zu sortieren ergibt dieselbe Reihenfolge wie eine Sortierung nach Codepunkten. Und es hat keine Bytereihenfolge; die Frage nach Big Endian oder Little Endian, die UTF-16 und UTF-32 beantworten müssen, stellt sich deshalb nicht.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist UTF-8?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Software, die in jedem Ihrer Absatzmärkte natürlich klingt — einschließlich Datumsangaben, Währungen und der Sortierung von Namen.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. Englisch-dänisches i18n-System aufgebaut — 12,027 Nachrichten je Sprachversion, auf Englisch und Dänisch

Englisch-dänisches i18n-System aufgebaut Kontakt aufnehmen →

Warum es wichtig ist

Das Web hat sich darauf festgelegt. Der Encoding Standard von WHATWG, den Browser zum Lesen und Schreiben von Text implementieren, bezeichnet UTF-8 als bevorzugte Kodierung für den Austausch von Unicode. Er verlangt sie für neue Protokolle und Formate sowie für bestehende Formate, die in neuen Zusammenhängen eingesetzt werden. Autoren sollen UTF-8 verwenden und es mit der Bezeichnung utf-8 angeben. HTML folgt dem: Die Kodierung eines Dokuments muss UTF-8 sein, unabhängig davon, ob es sie deklariert, und Autorenwerkzeuge müssen UTF-8 standardmäßig für neue Dokumente verwenden. Von einem Produkt, das Seiten veröffentlicht, JSON sendet oder Formularübermittlungen annimmt, wird bereits erwartet, dass es sie beherrscht.

Der Aufbau erklärt, warum diese Festlegung Bestand hat. Weil ASCII-Bytes nie innerhalb eines anderen Zeichens vorkommen, funktioniert ein für ASCII geschriebenes Programm weiterhin mit UTF-8-Text: Ein Parser, der an Kommas, Schrägstrichen oder Zeilenumbrüchen trennt, findet sie in einem dänischen oder ukrainischen Satz genau dort, wo sie stehen, und nie mitten in einem å oder ľ. Weil ein Programm an einer beliebigen Position den Anfang eines kodierten Skalarwerts findet, indem es höchstens vier Bytes und meist weniger zurücksucht, kann die Dekodierung bei einer folgenden gültigen Folge fortgesetzt werden, statt den Rest der Datei zu verlieren. Das verspricht nicht genau ein Ersatzzeichen für jedes beschädigte Byte. Eine Suche nach einem vollständig kodierten Skalarwert trifft nie einen Teil eines anderen. Weil die binäre Reihenfolge der Codepunktreihenfolge entspricht, kann eine Datenbank oder ein Programm UTF-8-Bytes ohne Dekodierung sortieren. Diese Reihenfolge ist allerdings technisch: Sie stellt å vor æ und ø, während ein dänisches Wörterbuch das Alphabet mit æ, ø und å abschließt.

Der Preis ist die Größe. Der Unicode-Standard nennt UTF-8 hinreichend kompakt: Für ASCII-Syntax und westliche Sprachen braucht es deutlich weniger Platz als UTF-16; Texte in asiatischen Schriften wie Hindi, Thai, Chinesisch, Japanisch und Koreanisch werden dagegen erheblich größer. Im dänischen Grundalphabet benötigen Buchstaben ein Byte, außer æ, ø und å mit je zwei; andere akzentuierte Buchstaben können mehr benötigen. Ukrainisch braucht zwei Bytes für jeden kyrillischen Buchstaben und eines für ASCII-Leerzeichen, Ziffern und Satzzeichen, wobei andere Satzzeichen mehr brauchen können. Ein in Bytes bemessenes Feld fasst deshalb ungefähr halb so viele ukrainische wie englische Buchstaben.

Wie es in der Praxis aussieht

Die Bytes lassen sich leicht sichtbar machen. Node.js besitzt wie ein Browser den `TextEncoder` aus dem Encoding Standard, der eine Zeichenfolge als UTF-8 schreibt:

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

Unter Node.js 25 gibt das Programm jedes Zeichen und seine Bytes hexadezimal aus:

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

Der Buchstabe A gehört zu ASCII und behält sein einzelnes Byte. Das dänische å ist U+00E5 und benötigt acht Bits, eines mehr als die sieben, die ein einzelnes Byte bietet. Es verwendet deshalb das Zwei-Byte-Muster. Sein Elf-Bit-Wert 00011100101 wird in 00011 und 100101 aufgeteilt. Das Startbyte besteht aus 110, gefolgt von 00011, also C3; das Fortsetzungsbyte aus 10, gefolgt von 100101, also A5. Das ukrainische Ґ, U+0490, benötigt auf dieselbe Weise zwei Bytes, das Eurozeichen drei und der altitalische Buchstabe 𐌂 jenseits von U+FFFF vier. Das erste Byte jeder Folge kündigt ihre Länge an: C3 und D2 beginnen mit 110, E2 mit 1110 und F0 mit 11110.

Diese Website schreibt jede Seite in UTF-8 und gibt das dort an, wo HTML es verlangt. Das erste Element im Kopf jeder Seite ist `<meta charset="utf-8">`, mit der vom Encoding Standard vorgeschriebenen Bezeichnung utf-8 und deutlich innerhalb der ersten 1.024 Bytes, in denen HTML die Deklaration verlangt. Die Credits-Seite hat einen Abschnitt mit externen Prüfungen, von denen keine von uns stammt. Jede führt einen Live-Test aus, statt uns etwas zu glauben. Dazu gehört der W3C Internationalization Checker, der die Sprach- und Kodierungsdeklarationen liest, durch die jede Sprachfassung dieser Website für die richtigen Leser verständlich wird.

Wo es schiefgeht

Eine Bytegrenze schneidet ein Zeichen durch. Eine in Bytes bemessene Spalte, ein Nachrichtenfeld oder ein Puffer kann mitten in einer Folge enden. Der Name Ærø hat drei Zeichen und fünf Bytes. Eine Grenze von vier behält Æ, r und das erste Byte von 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 }
```

Das allein stehende Startbyte wird als U+FFFD, das Ersatzzeichen, dekodiert. Der gespeicherte Name endet deshalb mit einem Platzhalter an der Stelle des ø. Geschnitten werden sollte stattdessen an einer Grenze zwischen Unicode-Skalarwerten. `encodeInto` in der letzten Zeile erledigt das selbst: Bei vier Bytes Platz schreibt es nur die drei Bytes von Æ und r und meldet unter `read`, wie viel es von der Zeichenfolge übernommen hat, gezählt in UTF-16-Einheiten. Code, der bereits vorhandene Bytes abschneidet, kann über die mit 10 beginnenden Fortsetzungsbytes zum Startbyte zurückgehen und dort schneiden. So bleibt UTF-8 wohlgeformt. Ein ganzes vom Benutzer wahrgenommenes Zeichen zu erhalten erfordert zusätzlich, die Grenzen von Graphemclustern zu beachten.

Eine Byte Order Mark steht am Dateianfang. UTF-8 hat keine Bytereihenfolge und braucht deshalb keine Markierung. Ein Konverter, der UTF-16 mit einer Byte Order Mark nach UTF-8 umsetzt, macht daraus jedoch die drei Bytes EF BB BF. Der Unicode-Standard verlangt sie nicht, erklärt ihre Anwesenheit für unerheblich für die Konformität und empfiehlt sie nicht als UTF-8-Signatur. Leseprogramme behandeln sie unterschiedlich. Hier enthält bom.json diese drei Bytes, gefolgt von `{"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` behält die Markierung als Zeichen U+FEFF bei; deshalb weist `JSON.parse` den Text zurück. `TextDecoder` folgt dem Encoding Standard, entfernt die Markierung, und dieselben Bytes lassen sich fehlerfrei parsen. Im Browser reicht die Wirkung der Markierung weiter: Der Encoding Standard gibt ihr Vorrang vor allem anderen, auch vor dem HTTP-Header `Content-Type`. UTF-8 sollte ohne Markierung geschrieben werden. Dateien aus anderen Systemen sollten mit einem Decoder gelesen werden, der sie entfernt; alternativ ist ein führendes U+FEFF vor dem Parsen zu entfernen.

Eine Prüfung liest die Bytes vor ihrer Dekodierung. RFC 3629 beschreibt einen Parser, der seine Eingabe in kodierter Form prüft und die Bytes 2F 2E 2E 2F für den Pfadschritt "/../" zurückweist. Er lässt aber 2F C0 AE 2E 2F durch, wobei C0 AE eine ungültige Zwei-Byte-Schreibweise des Punkts ist, die der Parser anschließend als Punkt liest. Der RFC hält fest, dass ein 2001 unter Webservern weit verbreiteter Virus genau diesen Trick nutzte. Seit Unicode 3.1 sind solche nicht kürzestmöglichen Formen fehlerhaft; die Bytes C0, C1 sowie F5 bis FF kommen in gültigem UTF-8 nie vor. Ein konformer Decoder, darunter `TextDecoder`, macht aus diesen fünf Bytes einen Schrägstrich, zwei U+FFFD und "./". Damit bleibt kein "/../", das die Prüfung umgehen könnte. Ebenso streng ist der Unicode-Standard beim Verhalten nach einem Fehler: Ein Decoder, der gültige Bytes nach einem fehlerhaften Startbyte verschluckt, ist nicht konform und für Sicherheitsangriffe anfällig. `TextDecoder` macht aus den Bytes C2 41 42 ein U+FFFD, gefolgt von AB. Zuerst sollte mit dem plattformeigenen Decoder dekodiert und danach der dekodierte Text geprüft werden. Ein UTF-8-Decoder oder ein ihn ersetzendes Bytemuster sollte nie von Hand geschrieben werden.

Ersetzung verbirgt den Schaden. Standardmäßig ersetzt ein Decoder jede fehlerhafte Folge durch U+FFFD und fährt fort. Für die Anzeige einer Seite ist das richtig, für den Datenimport falsch: Der Text liest sich beinahe normal, die gespeicherte Kopie hat die ursprünglichen Bytes verloren, und der Absender erfährt nie, wo die Datei beschädigt war. Der Encoding Standard benennt zwei Fehlermodi für die Dekodierung, „replacement" und „fatal". Für Kennungen und Bytefolgen innerhalb eines Formats oder Protokolls dürfen andere Standards einen Decoder wählen, der abbricht, statt Zeichen zu ersetzen. In JavaScript wirft `new TextDecoder("utf-8", { fatal: true })` beim oben abgeschnittenen Namen einen `TypeError`, statt ihn zurückzugeben. Wo Text gespeichert, signiert oder verglichen wird, sollte der fatale Modus verwendet, die Eingabe mit einer für den Absender hilfreichen Meldung zurückgewiesen und die Ersetzung auf die Anzeige beschränkt werden.

Die Deklaration kommt spät, fehlt oder ist falsch. HTML verlangt, dass das Element mit der Kodierungsdeklaration vollständig innerhalb der ersten 1.024 Bytes des Dokuments steht. Auch bei reinem ASCII-Text ist eine Deklaration erforderlich, denn der Browser braucht eine Kodierung für Formulareingaben und für URLs, die Skripte bilden. Auch eine harmlos wirkende Bezeichnung kann falsch sein: Im Encoding Standard sind “ascii”, “latin1” und “iso-8859-1” Bezeichnungen für windows-1252. Eine als ASCII deklarierte Seite wird deshalb als windows-1252 gelesen. `<meta charset="utf-8">` gehört an den Anfang des Kopfes, `Content-Type: text/html; charset=utf-8` zu jeder Seite; jede andere Bezeichnung an einer der beiden Stellen ist als Fehler zu behandeln.

Weiterführende Lektüre

- The Unicode Standard, Kapitel 3: Conformance — die Definition von UTF-8, die Tabelle wohlgeformter Bytefolgen, die Byte Order Mark und die Behandlung fehlerhafter Eingaben durch Konverter.
- The Unicode Standard, Kapitel 2: General Structure — warum UTF-8 für ASCII transparent, selbstsynchronisierend und frei von Bytereihenfolge ist und wie sein Platzbedarf im Vergleich zu UTF-16 ausfällt.
- RFC 3629: UTF-8, a transformation format of ISO 10646 — die IETF-Definition, ihre Geschichte seit Plan 9 und die Sicherheitsüberlegungen zum Angriff mit überlanger Pfadkodierung.
- Encoding Standard — die WHATWG-Spezifikation für Browser: vorgeschriebenes UTF-8, Byte Order Marks und Fehler beim Dekodieren sowie die Schnittstellen `TextEncoder` und `TextDecoder`.
- HTML: Specifying the document’s character encoding — Inhalt und Position der Deklaration und warum eine ausschließlich mit ASCII geschriebene Seite eine benötigt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist UTF-8?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/utf-8/