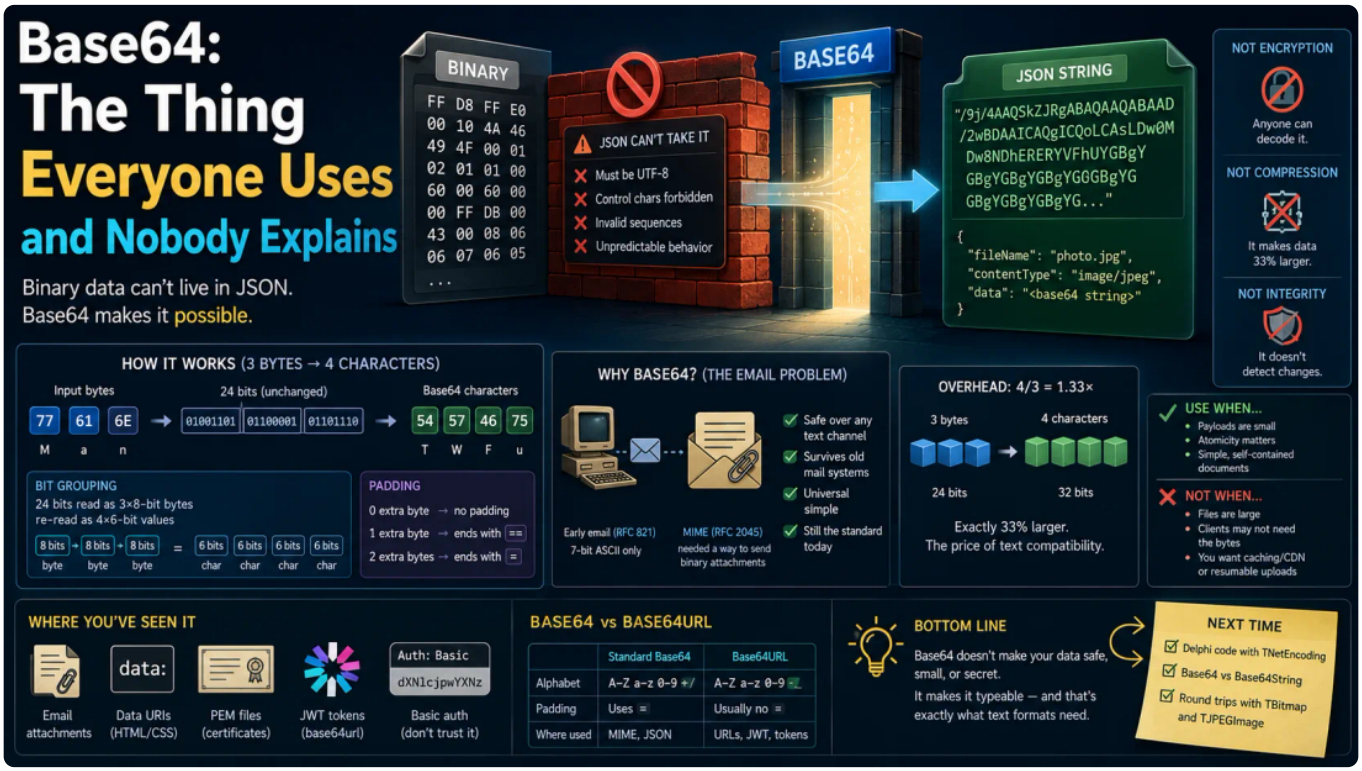
NEWS · JULY 23, 2026

Base64: Das, was alle benutzen und niemand erklärt

Teil fünf der JSON-Serie für Delphi-Entwickler — warum Binärdaten nicht in JSON passen, was Base64 wirklich mit Ihren Bytes macht, warum E-Mail dieses Problem zuerst löste und was die 33 Prozent tatsächlich kosten.

By Dr. Holger Flick



Diese Serie hat inzwischen festgehalten, woraus JSON besteht — sechs Wertetypen und eine rekursive Regel —, und wir haben einen ganzen Beitrag dem ersten gewidmet, was diese Liste nicht enthält: Datumsangaben.

Die Lösung dort war eine Konvention. Ein Datum wird zu einem String in einem vereinbarten Format, und beide Seiten müssen die Vereinbarung kennen.

Jetzt zum zweiten fehlenden Typ, und dieser ist schwerer. **Binärdaten.** Ein JPEG. Eine PDF-Rechnung. Ein signiertes Zertifikat. Ein ZIP-Archiv. Nichts davon ist ein String, eine Zahl, ein Boolean oder null, und keine noch so gute Konvention macht aus einem JPEG eine Zahl.

Die Antwort, die die ganze Branche verwendet, ist Base64 — und hier meine ehrliche Beobachtung aus Jahren Delphi-Code-Review: Die meisten Entwickler, die es korrekt einsetzen, könnten nicht sagen, was es tut. Sie kennen die Zauberformel — `TNetEncoding.Base64.EncodeBytesToString` —, sie wissen, dass ein langer String aus Buchstaben und Ziffern herauskommt, und sie wissen, dass es funktioniert. Fragt man, was mit den Bytes tatsächlich passiert, werden die Antworten vage, und ich habe mehr als einmal „es komprimiert das" und „es verschlüsselt das" gehört. Beides ist falsch, und das Zweite ist gefährlich.

Dieser Beitrag ist die Erklärung. Was Binärdaten sind, warum JSON sie wirklich nicht tragen kann, was Base64 auf Bit-Ebene mit Ihren Bytes macht, warum Ihr E-Mail-Programm das seit den frühen Neunzigern tut, und was es kostet. Diesmal überhaupt kein Delphi-Code — der kommt im nächsten Beitrag. Hier geht es darum, die Sache gut genug zu verstehen, dass der Code danach offensichtlich ist.

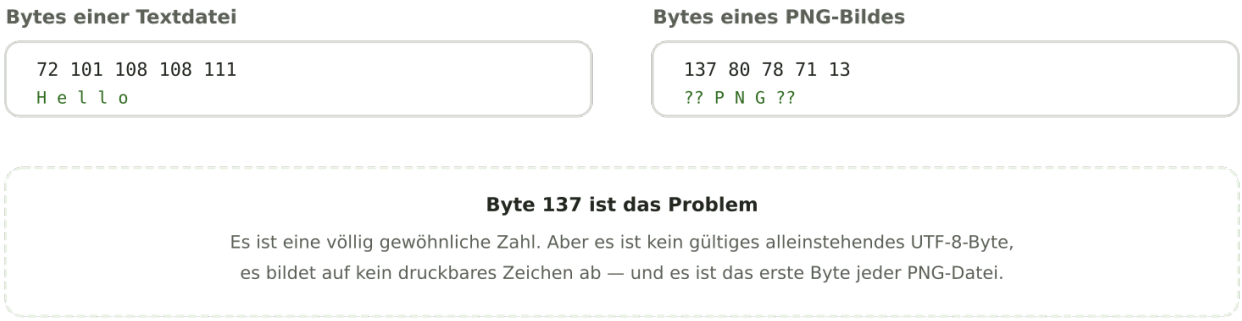
Was „binär" eigentlich heißt

Fangen wir tiefer an, als nötig scheint, denn genau hier beginnt die Verwirrung.

Jede Datei auf Ihrer Platte ist eine Folge von Bytes, und ein Byte ist eine Zahl von 0 bis 255. Das gilt für eine Textdatei genauso wie für ein JPEG. Der Unterschied liegt nicht in der Speicherung — er liegt in der **Vereinbarung darüber, was die Zahlen bedeuten**.

In einer Textdatei sollen die Bytes durch eine Zeichenkodierung gelesen werden: eine Nachschlagetabelle, die sagt „Byte 72 bedeutet den Buchstaben H". In einem JPEG bedeutet Byte 72 keinen Buchstaben. Es könnte Teil eines Farbwerts sein, ein Tabellenindex oder ein Fragment komprimierter Bilddaten. Es bedeutet das, was die JPEG-Spezifikation an dieser Position sagt.

„Binärdaten" sind also keine besondere Art von Bytes. **Es sind Bytes, deren Bedeutung nicht Text ist** — und, entscheidend, Bytes, die *jeden* Wert von 0 bis 255 annehmen, auch die, die keine Zeichenkodierung auf etwas Druckbares abbildet.



Dieselben Bytewerte, zwei verschiedene Vereinbarungen über die Bedeutung

Der gestrichelte Kasten ist der Kern des ganzen Beitrags. Byte 137 ist nicht exotisch oder defekt — es ist einfach eine Zahl. Aber es gibt keinen Weg, es direkt in einen JSON-String zu setzen, und der nächste Abschnitt erklärt genau, warum.

Warum JSON sie nicht nehmen kann

Drei getrennte Hindernisse stehen im Weg, und jedes allein würde genügen.

JSON ist Text, und Text heißt UTF-8. RFC 8259 ist unmissverständlich: „JSON text exchanged between systems that are not part of a closed ecosystem MUST be encoded using UTF-8." Das ist ein MUST, das stärkste Wort, das die IETF hat. Und UTF-8 ist kein Freibrief — es ist eine strukturierte Kodierung mit strengen Regeln darüber, welche Bytefolgen zulässig sind. Die meisten Bytefolgen, die in einem JPEG natürlich vorkommen, sind **kein gültiges UTF-8**. Nicht „ungewöhnlich" — tatsächlich ungültig, von einem konformen Parser abgelehnt.

Steuerzeichen sind rundheraus verboten. Selbst wenn man UTF-8 beiseitelässt: JSON-Strings dürfen keine rohen Steuerzeichen enthalten — der RFC verlangt Escaping für U+0000 bis U+001F. Binärdaten sind voll

davon. Insbesondere Byte 0 ist in einer Binärdatei völlig gewöhnlich und roh in einem JSON-String völlig unzulässig. (Für Delphi- und C-Entwickler schlimmer: Ein Nullbyte terminiert in vielen Sprachen einen String, sodass selbst Code, der es *tolerierte*, stillschweigend abschneiden würde.)

Ungültige Folgen erzeugen unvorhersehbares Verhalten. RFC 8259 sagt offen, dass bei JSON-Text mit

Bitfolgen, die keine Unicode-Zeichen kodieren können, „the behavior of software that receives JSON texts containing such values is unpredictable" ist. Andere Parser, andere Ergebnisse, keine Zusage. Das ist kein Bug, den man umgeht; das ist die Spezifikation, die sich weigert, den Fall zu definieren.

Drei unabhängige Gründe, warum die rohen Bytes nicht hineinpassen

Beachten Sie, dass das drei *unabhängige* Barrieren sind. Lösen Sie eine, bleiben die anderen zwei. Deshalb ist die Lösung kein Workaround und kein Escaping-Trick — sie muss ein echter Wechsel der Repräsentation sein.

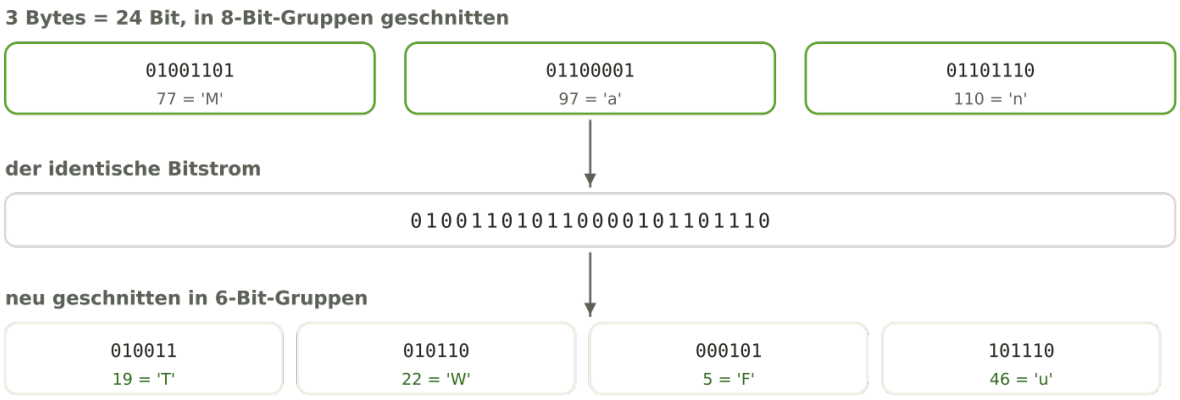
Was Base64 tatsächlich macht

Hier die Idee in einem Satz, und er lohnt zweimal gelesen zu werden: **Base64 schreibt Ihre Bytes mit nur 64 überall sicheren Zeichen neu, indem es die Bits neu gruppiert.**

Nicht komprimieren. Nicht verschlüsseln. **Bits neu gruppieren.** Das ist der ganze Mechanismus, und sobald man ihn sieht, hört der 33-Prozent-Aufschlag auf, ein Rätsel zu sein, und wird zu Arithmetik.

Ein Byte hat 8 Bit. Es gibt 256 mögliche Werte, und die meisten sind als Text nicht sicher. Base64 stellt also eine andere Frage: Was, wenn wir den Bitstrom in **6-Bit**-Stücke schneiden statt in 8-Bit-Stücke? Sechs Bit ergeben 2v = 64 mögliche Werte — und 64 verschiedene, druckbare, universell sichere Zeichen sind leicht zu finden. RFC 4648 definiert das Alphabet: `A`–`Z`, `a`–`z`, `0`–`9`, `+` und `/`.

Die Neugruppierung arbeitet mit 3 Bytes auf einmal, denn 3 Bytes sind 24 Bit, und 24 ist sowohl durch 8 als auch durch 6 glatt teilbar:



Drei Bytes rein, vier Zeichen raus — dieselben 24 Bit, anders geschnitten

Folgen Sie der mittleren Zeile, und Sie sehen den Punkt: **Die Bits ändern sich nie.** Aus `Man` wird `TWFu`, und die Bitfolge in der Mitte ist in beiden Lesarten identisch. Drei Bytes gingen hinein, vier Zeichen kamen heraus. Verändert hat sich nur, wo die Schere angesetzt wurde.

Daher kommt auch der Aufschlag, ganz ohne Handwedeln: 4 Ausgabezeichen je 3 Eingabebytes sind genau 4/3, also **33 Prozent mehr**. RFC 2045 sagt es unumwunden — kodierte Daten seien „consistently only about 33 percent larger than the unencoded data".

Die Gleichheitszeichen am Ende

Nun das Detail, das jeder gesehen und kaum jemand erklärt bekommen hat: Warum enden Base64-Strings so oft auf `=` oder `==`?

Das Verfahren arbeitet in 3-Byte-Gruppen, aber echte Daten kommen nicht in bequemen Dreierschritten. Endet Ihre Eingabe mit 1 oder 2 übrigen Bytes, füllt der Kodierer die Bits zu einer vollen Gruppe auf und hängt dann `=`-Zeichen an, um festzuhalten, wie viel der letzten Gruppe echt war:

- Eingabelänge durch 3 teilbar ’ keine Auffüllung.
- **1 Byte** übrig ’ Ausgabe endet auf`==`.
- **2 Bytes** übrig ’ Ausgabe endet auf`=`.

Das `=` ist das 65. Zeichen im Alphabet des RFC und trägt **überhaupt keine Daten** — es ist eine Längenmarkierung, damit der Dekodierer weiß, wie viele Bytes er aus der letzten Gruppe erzeugen soll. RFC 4648 verlangt es standardmäßig: „Implementations MUST include appropriate pad characters at the end of encoded data unless the specification referring to this document explicitly states otherwise." Diese Ausnahmeklausel ist wichtig und taucht weiter unten wieder auf.

> **Base64 ist keine Sicherheit. Nicht im Geringsten.**
> Das ist das Missverständnis mit echten Konsequenzen, deshalb sage ich es deutlich. Base64 ist **keine Verschlüsselung**. Es gibt keinen Schlüssel, kein Geheimnis und nichts zu brechen — Dekodieren ist eine öffentliche, mechanische Operation, die jeder Programmierer, jede Browser-Konsole und jedes Kommandozeilenwerkzeug sofort durchführt. Ein Base64-String ist *genau so geheim* wie die Bytes darin. Es ist auch **keine Kompression**. Es macht Daten um 33 Prozent größer, nicht kleiner. Und es ist **kein Integritätsschutz** — keine Prüfsumme, keine Signatur, und es wird Ihnen nicht sagen, ob Bytes unterwegs

verändert wurden. Wo das tatsächlich wehtut: HTTP-Basic-Authentifizierung kodiert `benutzer:passwort` in Base64, was kodiert aussieht und deshalb für geschützt gehalten wird. Ist es nicht. Es ist Klartext mit Zwischenschritt — und genau deshalb ist Basic Auth über alles außer HTTPS ein Abfluss von Zugangsdaten. Brauchen Sie Geheimhaltung, verschlüsseln Sie; brauchen Sie Integrität, signieren Sie. Base64 löst ein *Transportproblem*, und nur dieses.

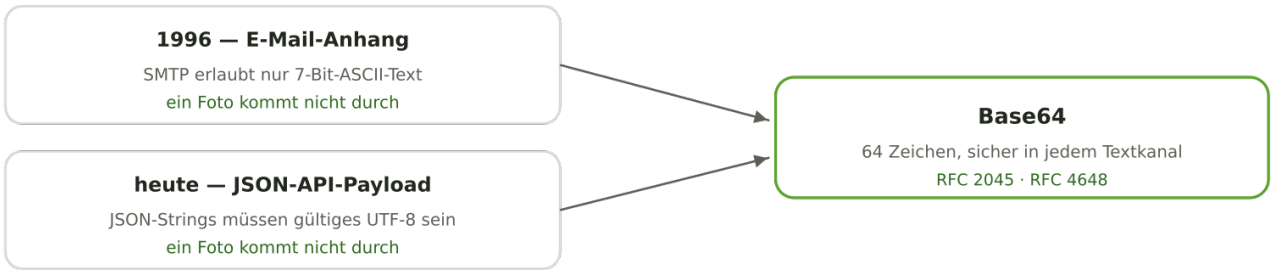
E-Mail hat das zuerst gelöst

Der Grund, warum Base64 sich so universell anfühlt, ist, dass es das ist — und es wurde es, indem es genau dieses Problem löste, mehr als ein Jahrzehnt bevor JSON existierte.

Frühe Internet-E-Mail war für reinen Text spezifiziert. RFC 821, der ursprüngliche SMTP-Standard von 1982, beschränkte Nachrichten auf 7-Bit-US-ASCII. Nicht 8-Bit-Bytes — **sieben Bit pro Zeichen**, weil die Mail-Infrastruktur der Zeit darauf gebaut war. Mailserver auf der Strecke konnten das achte Bit verstümmeln, Zeichen entfernen und Zeilenenden umschreiben, und sie taten es.

Ein Foto durch diese Infrastruktur zu schicken war unmöglich. Also definierte MIME — Multipurpose Internet Mail Extensions, 1996 — eine Reihe von Content-Transfer-Encodings, und Base64 war das für beliebige Binärdaten. RFC 2045 beschreibt das Entwurfsziel genau: eine Kodierung, „designed to represent arbitrary sequences of octets in a form that need not be humanly readable", die Daten „into material in the '7bit' range" verwandelt und sie damit „safe to carry over restricted transports" macht.

Diese Formulierung — *need not be humanly readable* — ist die ehrliche. Base64 sollte nie hübsch sein. Es sollte überleben.



Dasselbe Problem, zwanzig Jahre auseinander, mit derselben Antwort

Das Diagramm ist das Argument dafür, das hier richtig zu lernen statt eine Formel auswendig zu können. Es ist **dasselbe Problem**: ein Kanal, der nur Text akzeptiert, und Daten, die kein Text sind. Wer es einmal versteht, versteht auch, warum Data-URIs in HTML funktionieren, warum JWT-Tokens aussehen, wie sie aussehen, warum PEM-Zertifikatsdateien lesbares ASCII sind und warum `Authorization: Basic`-Header diese Form haben. Alles derselbe Trick.

Die Variante, der Sie begegnen werden: base64url

Eine Feinheit, die man besser kennt, bevor sie zubeißt. Standard-Base64 verwendet `+` und `/`, und beide haben in URLs besondere Bedeutung — `/` ist ein Pfadtrenner, und `+` wird oft als kodiertes Leerzeichen gelesen. RFC 4648 definiert daher eine **URL-sichere Variante**, base64url, die sie durch `-` und `_` ersetzt. Sie lässt außerdem üblicherweise die `=`-Auffüllung weg und nutzt dafür die oben zitierte Ausnahmeklausel. Das ist nicht kosmetisch: Ein base64url-String, den man einem Standard-Base64-Dekodierer vorsetzt, scheitert oder liefert Müll, und umgekehrt genauso. Wenn Sie mit JWT-Tokens arbeiten, benutzen Sie bereits

> base64url — die Spezifikation schreibt es vor —, und deshalb enthalten JWTs `-` und `_` und enden meist ohne `=`. Delphi bietet beides an, und der nächste Beitrag zeigt genau, wo.

Was die 33 Prozent wirklich kosten

Zeit für die ehrliche Ingenieursdiskussion, denn „Base64 fügt 33 Prozent hinzu" wird nachgeplappert, als sei damit etwas entschieden, und das ist es nicht.

Der Aufschlag ist real und er summiert sich. Aus einem 5-MB-Foto werden etwa 6,7 MB Base64. Aber die kodierte Größe ist nicht der ganze Preis. Dieser String muss vom Sender **im Speicher aufgebaut**, während der Zusammensetzung des JSON-Dokuments **im Speicher gehalten**, übertragen und dann vom Empfänger **geparst und dekodiert** werden — und eine naive Implementierung hält womöglich mehrere Kopien gleichzeitig.

Bei einer großen Datei kann der Spitzenspeicher ein Vielfaches des Originals betragen, und das ist in der Produktion ein weit häufigeres Problem als die Bandbreite.

Dazu kommen die CPU-Kosten für Kodieren und Dekodieren, die bei den meisten Lasten allerdings gegenüber der Netzwerkzeit wirklich vernachlässigbar sind.

Wann ist es also richtig, Binärdaten in JSON einzubetten?

Es passt gut, wenn das Payload klein ist (Vorschaubilder, Icons, Signaturen, kleine Zertifikate), wenn Atomarität zählt — Metadaten und Bytes sollen als eine unteilbare Einheit ankommen, die nicht halb scheitern kann — oder wenn ein einziges, in sich geschlossenes Dokument schlicht mehr wert ist als die Effizienz, wie bei einem Webhook oder einer Message-Queue-Nachricht.

Es passt schlecht, wenn die Dateien groß sind, wenn Clients die Metadaten womöglich ohne die Bytes wollen, wenn Sie das Binärmaterial getrennt cachen oder über ein CDN ausliefern möchten oder wenn Sie fortsetzbare Uploads brauchen.

> **Die Alternative, die bei großen Dateien meist gewinnt**
>
> Für alles Größere ist der bessere Weg fast immer, **die Bytes gar nicht ins JSON zu legen**. Laden Sie die Datei getrennt hoch — ein einfaches `POST` der rohen Bytes, ein `multipart/form-data`-Request oder eine vorsignierte URL direkt in den Objektspeicher — und legen Sie stattdessen eine **URL ins JSON**.
>
> Kein 33-Prozent-Aufschlag, keine Speicherspitze, das Binärmaterial ist cachebar, und Clients holen es nur, wenn sie es brauchen. Meine Faustregel, ausdrücklich als Faustregel und nicht als Gesetz: unter ein paar hundert Kilobyte ist Einbetten meist in Ordnung und die Einfachheit den Preis wert; im Megabyte-Bereich greifen Sie zur URL. Aber die Schwelle hängt wirklich von Ihrem Verkehr und Ihren Randbedingungen ab — ein internes Dienstchen mit wenig Last, das 5-MB-Payloads einbettet, kann völlig gesund sein, während ein stark frequentierter Endpunkt schon bei 200 KB Probleme bekommt. Messen Sie Ihren eigenen Fall, statt meiner Zahl zu vertrauen.

Die Formate, die dieses Problem nicht haben

Es wäre unredlich, Base64 als einzig mögliche Antwort darzustellen, denn das ist es nicht — mehrere Formate tragen Binärdaten nativ, ganz ohne Kodierungsschritt.

BSON, das Binärformat von MongoDB, hat einen erstklassigen Binärtyp. MessagePack ist im Kern „JSON, aber binär" und trägt rohe Bytes direkt. CBOR ist ein IETF-Standard mit derselben Eigenschaft. Protocol Buffers hat einen `bytes`-Typ und wird in großen Systemen breit eingesetzt, besonders mit gRPC. Alle vermeiden die 33 Prozent vollständig, und alle sind echte, gut konstruierte, aktiv genutzte Technologien.

Aber hier die praktische Realität, und deshalb behandelt dieser Beitrag Base64 als die Antwort: **In einer öffentlichen Web-API werden Sie ihnen selten begegnen.** Ihre natürliche Heimat ist das Innere eines Systems — Dienst-zu-Dienst-Verkehr, Speicher-Engines, mobile Sync-Protokolle —, wo beide Seiten von demselben Team kontrolliert werden und sich auf ein Binärformat einigen können. Sobald eine API öffentlich ist, muss sie aus einem Browser, aus `curl`, aus einer Sprache konsumierbar sein, deren Ökosystem vielleicht keine Bibliothek für Ihr Format hat — von einem Entwickler, der die Antwort beim Debuggen mit bloßem Auge lesen muss. JSONs Lesbarkeit und Universalität gewinnen dieses Argument klar, und sie gewinnen es *trotz* der 33 Prozent, was zeigt, wie viel diese Eigenschaften wert sind.

Der Kompromiss ist also real und die Alternativen sind gut. Es ist nur so, dass für die konkrete Aufgabe „eine öffentliche API, die jeder aufrufen kann" JSON mit Base64 zum De-facto-Standard geworden ist — nicht weil es elegant ist, sondern weil es überall funktioniert und nichts installiert werden muss. So entstehen Standards meistens.

Fazit

Base64 ist eine kleine Idee mit einem geheimnisvollen Namen, und genau zu wissen, was sie tut, räumt eine ganze Kategorie von Verwirrung ab.

- **Binärdaten sind einfach Bytes, deren Bedeutung nicht Text ist** — einschließlich der vielen Bytewerte, die keine Zeichenkodierung auf etwas Druckbares abbildet.
- **JSON kann rohe Bytes wirklich nicht tragen.** RFC 8259 sagt, JSON-Text MUSS UTF-8 sein, Steuerzeichen müssen escaped werden, und ungültige Folgen machen das Parserverhalten „unpredictable". Drei unabhängige Barrieren, eine notwendige Lösung.
- **Base64 gruppiert Bits neu, mehr nicht.** Vierundzwanzig Bit, gelesen als drei 8-Bit-Bytes, neu gelesen als vier 6-Bit-Werte, jeder auf eines von 64 sicheren Zeichen abgebildet. Dieses Verhältnis 4 zu 3 *ist* der 33-Prozent-Aufschlag. Die `=` am Ende sind Auffüllung, keine Daten.
- **Es ist weder Verschlüsselung noch Kompression noch Integritätsschutz.** Ein Base64-String ist exakt so geheim wie sein Inhalt — weshalb Basic Auth über einfaches HTTP Zugangsdaten preisgibt.
- **E-Mail hat das 1996 gelöst, und die Begründung überträgt sich unverändert.** Dasselbe Problem, dieselbe Antwort — und deshalb sehen Data-URIs, PEM-Dateien und JWTs so aus, wie sie aussehen.
- **Kleine Payloads einbetten, große verlinken.** Unter ein paar hundert KB ist Base64 in JSON meist in Ordnung. Im Megabyte-Bereich gehört eine URL ins JSON und die Bytes gehen getrennt.

> Base64 macht Ihre Daten nicht sicher, nicht klein und nicht geheim. Es macht sie *tippbar* — und das ist genau die Eigenschaft, die ein Textformat braucht.

Nächstes Mal schreiben wir den Code. Kodieren und Dekodieren mit `TNetEncoding`, der Unterschied zwischen `Base64` und `Base64String`, der JSON-Payloads still zerlegt, und vollständige Rundwege mit echten Bildern — `TBitmap` und `TJPEGImage` — mit nichts als der RTL.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —