NEWS · JULY 24, 2026

Von TBitmap nach JSON und zurück

Teil sechs der JSON-Serie für Delphi-Entwickler — Base64 kodieren und dekodieren mit TNetEncoding, die Falle Base64 versus Base64String und vollständige Bild- Rundwege mit TBitmap und TJPEGImage.

By Dr. Holger Flick



Letztes Mal haben wir Base64 auseinandergenommen. Wir haben festgehalten, dass Binärdaten nicht in JSON passen, weil RFC 8259 UTF-8 verlangt und die meisten Bytes eines Bildes kein gültiges UTF-8 sind; dass Base64 das löst, indem es 24 Bit in vier 6-Bit-Werte umgruppiert, die auf 64 sichere Zeichen abgebildet werden; dass das genau 33 Prozent kostet; und dass es keinerlei Sicherheit bietet.

Das war die Theorie, und sie war bewusst codefrei. Dieser Beitrag ist das Gegenteil: der funktionierende Code — und weil Sie den Mechanismus verstehen, sollte jede Zeile davon jetzt zwangsläufig statt magisch wirken.

Wir behandeln Kodieren und Dekodieren mit `TNetEncoding`, eine echte Falle in der RTL, die stillschweigend JSON erzeugt, das Ihre Konsumenten ablehnen können, und vollständige Rundwege mit echten Bildern — `TBitmap` und `TJPEGImage` — mit nichts außerhalb von RTL und VCL. Am Ende können Sie ein Bild in ein JSON-Dokument legen und dasselbe Bild wieder herausholen, und Sie wissen genau, welche Schritte Daten verlieren können und welche nicht.

Der Kodierer wohnt in System.NetEncoding

Alles Nötige steckt in `System.NetEncoding`, einer Unit, die mit Delphi ausgeliefert wird und nichts zu installieren verlangt.

Der Einstiegspunkt ist die Klasse `TNetEncoding`, die fertige Kodierer-Instanzen als Klasseneigenschaften bereitstellt. **Sie erzeugen und geben diese nicht frei** — es sind Singletons, die die RTL verwaltet, weshalb sich die Aufrufe so sauber lesen:

```
uses
  System.NetEncoding;

var
  LBytes: TBytes;
  LText: string;
begin
  LBytes := TEncoding.UTF8.GetBytes('Man');
  LText := TNetEncoding.Base64String.EncodeBytesToString(LBytes);
  Writeln(LText);            // TWFu

  LBytes := TNetEncoding.Base64String.DecodeStringToBytes(LText);
  Writeln(TEncoding.UTF8.GetString(LBytes));   // Man
end;
```

Dieses `Man` ’`TWFu` ist genau das Beispiel, das wir im vorigen Beitrag Bit für Bit verfolgt haben — drei Bytes rein, vier Zeichen raus. Jetzt sind es zwei Methodenaufrufe.

Die beiden Methoden, die Sie fast immer verwenden werden, sind:

```
function EncodeBytesToString(const Input: array of Byte): string;
function DecodeStringToBytes(const Input: string): TBytes;
```

Beachten Sie die Asymmetrie in den Typnamen, denn sie ist bewusst und hilfreich: Sie kodieren **Bytes** zu einem **String** und dekodieren einen **String** zurück zu **Bytes**. Es gibt auch eine Überladung `Encode(const Input: string): string`, aber greifen Sie vorsichtig danach — sie kodiert den *Text* eines Strings, was eine Entscheidung über die Zeichenkodierung einschließt, und bei Binär-Payloads wollen Sie ausdrücklich mit Bytes arbeiten. Bleiben Sie bei den beiden oben, dann halten die Typen Sie ehrlich.

Die Falle: Base64 versus Base64String

Nun das, was viele erwischt, und es ist ein echter Bug statt einer Stilfrage. `TNetEncoding` bietet **drei** Base64-Kodierer an, und der falsche erzeugt Ausgabe, die gültiges Base64, aber problematisches JSON ist.

```
class property Base64: TNetEncoding;         // MIME-Stil – fügt Zeilenumbrüche ein
class property Base64String: TNetEncoding;   // eine durchgehende Zeile
class property Base64URL: TNetEncoding;      // URL-sicheres Alphabet, keine Auffüllung
```

Der Unterschied ist im RTL-Quelltext sichtbar. `TBase64Encoding.Create` übergibt `kCharsPerLine` und `kLine-Separator`, die so deklariert sind:

```
kCharsPerLine = 76;
kLineSeparator = #13#10;
```

`TNetEncoding.Base64` **fügt also alle 76 Zeichen ein CRLF ein.** `TBase64StringEncoding.Create` dagegen übergibt `0` und `''` — keine Zeilenlänge, kein Trenner — und erzeugt einen ununterbrochenen String.

Diese 76 ist nicht willkürlich, und der letzte Beitrag erklärt sie: RFC 2045 verlangt, dass „the encoded output stream must be represented in lines of no more than 76 characters each". `TNetEncoding.Base64` tut also genau das Richtige **für E-Mail**, wonach es benannt ist. Für ein JSON-Payload ist es nicht das Richtige.

TNetEncoding.Base64 — MIME-Stil

```
iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAADUlEQVR42mP8z8BQDwAEhQGAhKmMIQAA
<CR><LF>                                                    ← ein Umbruch, alle 76 Zeichen
AAAASUVORK5CYII=
```

TNetEncoding.Base64String — eine Zeile

```
iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAADUlEQVR42mP8z8BQDwAEhQGAhKmMIQAA
AAAASUVORK5CYII=
```

Dieselben Bytes, derselbe Algorithmus — eines davon ist in JSON ein Problem

Der obere Kasten ist das, was schiefgeht. Diese CRLF-Zeichen sind **Steuerzeichen**, und wie der vorige Beitrag festgehalten hat, dürfen JSON-Strings sie nicht roh enthalten — ein konformer Serializer muss sie als `\r\n` escapen. Ihr Payload wird also größer, es ist voller Escape-Sequenzen, und — der Teil, der wirklich beißt — ein strenger Konsument auf der anderen Seite kann den Wert ablehnen oder nicht dekodieren können, denn etliche Base64-Dekodierer in anderen Sprachen tolerieren eingebettete Leerzeichen nicht.

> **Die Regel, einmal gesagt**
> **Für JSON immer `TNetEncoding.Base64String` verwenden.** Nehmen Sie `TNetEncoding.Base64` nur, wenn Sie tatsächlich MIME-Inhalte erzeugen, etwa einen E-Mail-Body. Nehmen Sie `TNetEncoding.Base64URL`, wenn der Wert in einen URL-Pfad, einen Query-Parameter oder ein JWT geht. Danach lohnt sich ein `grep` durch Ihre Codebasis. `TNetEncoding.Base64` ist der Name, der nach Standard klingt, also greift man danach — und die entstehenden Payloads funktionieren oft prima gegen einen toleranten Dekodierer, bis zur Integration mit einem, der es nicht ist. Der Fehler kommt spät und sieht aus wie das Problem eines anderen.

Die URL-Variante ist dieselbe Geschichte in die andere Richtung:

```
LToken := TNetEncoding.Base64URL.EncodeBytesToString(LBytes);
// nutzt '-' und '_' statt '+' und '/' und erzeugt keine '='-Auffüllung
```

Der RTL-Quelltext bestätigt beides: `TBase64URLEncoding.Create` übergibt eigene Alphabettabellen sowie `False` für die Auffüllung. Das entspricht dem URL-sicheren Alphabet aus RFC 4648 und ist der Grund, warum JWT-Tokens `-` und `_` enthalten und typischerweise ohne `=` enden.

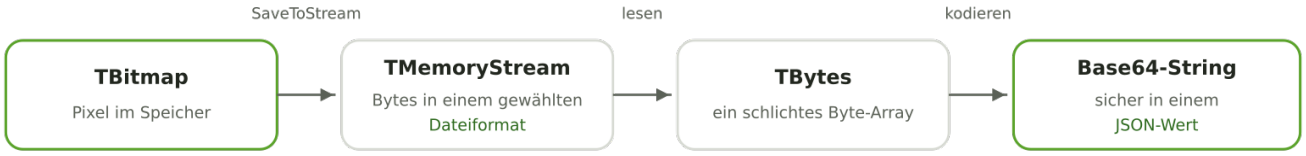
Bytes aus einem Bild herausholen

Nun zu Bildern, und hier gibt es einen gedanklichen Schritt, über den sonst korrekter Code stolpert.

Ein `TBitmap` im Speicher ist keine Datei. Es ist ein lebendes Objekt mit Pixeln, einer Palette, einem Gerätekontext. Es hat keine inhärente Byte-Repräsentation — die Frage „was sind die Bytes dieses Bitmaps?"

hat keine Antwort, bevor Sie ein **Dateiformat** wählen, in das es serialisiert wird. BMP? PNG? JPEG? Jedes erzeugt völlig andere Bytes für dasselbe Bild.

Die Kette hat also eine Stufe mehr, als man erwartet:



Vier Stufen, und jede ist eine echte Umwandlung

Den zweiten Kasten überspringt man gedanklich gern. In `SaveToStream` fällt die Formatentscheidung, und sie bestimmt, ob Ihr 500-KB-Foto als 2-MB-BMP oder als 400-KB-JPEG im Payload landet. Die 33 Prozent von Base64 gelten dann auf das, was dabei herauskam — **die Formatwahl wiegt also weit schwerer als der Kodierungsaufschlag**.

Hier die Hilfsfunktion, und sie ist kurz:

```
uses
  System.Classes, System.NetEncoding, Vcl.Graphics;

function GraphicToBase64(AGraphic: TGraphic): string;
var
  LStream: TMemoryStream;
  LBytes: TBytes;
begin
  LStream := TMemoryStream.Create;
  try
    AGraphic.SaveToStream(LStream);       // das Format bestimmt die Klasse von AGraphic
    SetLength(LBytes, LStream.Size);
    LStream.Position := 0;
    LStream.ReadBuffer(LBytes, LStream.Size);
    Result := TNetEncoding.Base64String.EncodeBytesToString(LBytes);
  finally
    LStream.Free;
  end;
end;
```

`TGraphic` statt `TBitmap` entgegenzunehmen ist Absicht — `TBitmap`, `TJPEGImage` und `TPNGImage` stammen alle davon ab, diese eine Funktion bedient also jedes Format, und die *Klasse des übergebenen Objekts* wählt die Kodierung. Das ist die Formatentscheidung, sichtbar gemacht durch Ihre Typwahl.

Die Zeile `LStream.Position := 0` ist nicht optional. Nach `SaveToStream` steht die Streamposition am Ende; das Zurücksetzen zu vergessen liest nichts und liefert einen leeren String. Ein Klassiker, und er scheitert leise.

Ein Bild ins JSON legen

Mit der Hilfsfunktion nutzt der Aufbau des Payloads das Objektmodell aus Teil drei:

```
uses
  System.JSON, System.NetEncoding, System.Classes, Vcl.Graphics, Vcl.Imaging.jpeg;

var
  LJpeg: TJPEGImage;
  LBitmap: TBitmap;
  LRoot: TJSONObject;
begin
  LBitmap := TBitmap.Create;
  LJpeg := TJPEGImage.Create;
  LRoot := TJSONObject.Create;
  try
    LBitmap.LoadFromFile('photo.bmp');

    LJpeg.Assign(LBitmap);              // umwandeln: Bitmap-Pixel -> JPEG
    LJpeg.CompressionQuality := 80;     // 1..100; niedriger = kleiner und verlustreicher

    LRoot.AddPair('filename', 'photo.jpg');
    LRoot.AddPair('content_type', 'image/jpeg');
    LRoot.AddPair('data', GraphicToBase64(LJpeg));

    Writeln(LRoot.ToJSON);
    // {"filename":"photo.jpg","content_type":"image/jpeg","data":"/9j/4AAQSkZJRg..."}
  finally
    LRoot.Free;
    LJpeg.Free;
    LBitmap.Free;
  end;
end;
```

Zwei Zeilen verdienen Aufmerksamkeit. `LJpeg.Assign(LBitmap)` ist der Ort, an dem die echte Größenreduktion passiert — das ist JPEG-Kompression, und sie schrumpft ein Foto gegenüber dem rohen Bitmap typischerweise um eine Größenordnung. Die 33 Prozent von Base64 gelten dann auf diese viel kleinere Zahl. **Vor dem**

Kodieren zu komprimieren ist weit wirksamer, als sich um den Base64-Aufschlag zu sorgen, und diese

Reihenfolge macht eingebettete Bilder überhaupt erst praktikabel.

Und `content_type` ist keine Deko. Base64 ist formatblind — der Dekodierer bekommt Bytes zurück und hat keine Ahnung, ob es ein JPEG, ein PNG oder ein PDF ist. Den MIME-Typ mitzuschicken ist das, was den

Empfänger wissen lässt, was er konstruieren soll. Ihn wegzulassen zwingt die Gegenseite zum Raten, und Raten ist der Weg zu Sniffing-Bugs.

Wieder einlesen

Der Rückweg dekodiert zu Bytes, hüllt sie in einen Stream und lädt:

```
uses
  System.JSON, System.NetEncoding, System.Classes, Vcl.Graphics, Vcl.Imaging.jpeg;

procedure LoadImageFromJson(const AJson: string; ATarget: TPicture);
var
  LValue: TJSONValue;
  LRoot: TJSONObject;
  LData, LContentType: string;
  LBytes: TBytes;
  LStream: TBytesStream;
  LJpeg: TJPEGImage;
begin
  LValue := TJSONObject.ParseJSONValue(AJson);
  if not (LValue is TJSONObject) then
    raise Exception.Create('JSON-Objekt erwartet');
  try
```

```
    LRoot := TJSONObject(LValue);
    LData := LRoot.GetValue<string>('data', '');
    LContentType := LRoot.GetValue<string>('content_type', '');

    if LData = '' then
      raise Exception.Create('Keine Bilddaten im Payload');
    if LContentType <> 'image/jpeg' then
      raise Exception.CreateFmt('Nicht unterstützter Content-Type: %s', [LContentType]);

    LBytes := TNetEncoding.Base64String.DecodeStringToBytes(LData);

    LStream := TBytesStream.Create(LBytes);
    try
      LJpeg := TJPEGImage.Create;
      try
        LJpeg.LoadFromStream(LStream);
        ATarget.Assign(LJpeg);          // ATarget übernimmt eine Kopie
      finally
        LJpeg.Free;
      end;
    finally
      LStream.Free;
    end;
  finally
    LValue.Free;
  end;
end;
```

`TBytesStream` ist der elegante Teil — es hüllt ein bestehendes `TBytes` als Stream, ohne zu kopieren, und ist damit genau der Adapter, den man zwischen `DecodeStringToBytes` und `LoadFromStream` braucht.

Die `content_type`-Prüfung leistet echte Arbeit. `TJPEGImage.LoadFromStream` löst bei PNG-Bytes eine Exception aus, und obwohl das behandelbar ist, gibt die Prüfung des deklarierten Typs vorab einen klaren Fehler statt eines Dekodiererabsturzes. Beachten Sie, dass `GetValue<string>` mit Standardwert den Fall des fehlenden Schlüssels abdeckt — die Prüfungen betreffen also den *Inhalt*, nicht die Existenz des Felds: das Muster aus

Teil drei, angewandt.

> **Validieren Sie, bevor Sie etwas dekodieren, das Sie nicht selbst erzeugt haben**
> Base64 aus nicht vertrauenswürdiger Quelle zu dekodieren ist eine **Speicheranforderung, deren Größe ein Fremder bestimmt**. Ein böswilliger Client kann ein paar hundert Megabyte Base64 in ein Feld schicken, in dem Ihr Code ein Vorschaubild erwartet, und `DecodeStringToBytes` wird treu versuchen, alles zu allozieren — bevor irgendeine Ihrer Bildlogiken läuft. Bilddekodierer selbst haben eine lange Geschichte von Schwachstellen bei fehlerhaften Eingaben, was ein zweiter Grund ist, ihnen keine beliebigen Bytes zu übergeben. Eine billige und wirksame Absicherung ist, zuerst die kodierte Länge zu prüfen, denn die Base64-Länge ist direkt proportional zur Byteanzahl: ``pascal const CMaxImageBytes = 2 * 1024 * 1024; // 2 MB dekodiert begin // 4 Base64-Zeichen je 3 Bytes, kodierte Länge ≈ Bytes * 4/3 if Length(LData) > (CMaxImageBytes div 3) * 4 + 4 then raise Exception.Create('Bild-Payload zu groß'); LBytes := TNetEncoding.Base64String.DecodeStringToBytes(LData); end; `` Lehnen Sie auf dem *kodierten* String ab, bevor Sie den dekodierten allozieren. Erzwingen Sie Ihr Limit außerdem serverseitig für den gesamten Request-Body — eine Prüfung im Anwendungscode hilft nichts, wenn die HTTP-Schicht bereits ein Gigabyte gepuffert hat.

Was der Rundweg bewahrt — und was nicht

Diese Unterscheidung ist wichtig, und beides zu vermengen führt dazu, dass Base64 für Verluste verantwortlich gemacht wird, die es nicht verursacht hat.

Base64 ist exakt verlustfrei. Kodieren Sie eine beliebige Bytefolge und dekodieren Sie sie, bekommen Sie identische Bytes — jedes Mal, ohne Ausnahme. Es ist ein reiner Wechsel der Repräsentation, wie das Bit-Umgruppierungsdiagramm im vorigen Beitrag gezeigt hat. Wenn Sie die Bytes einer JPEG-Datei in Base64 wandeln und wieder zurück, haben Sie diese JPEG-Datei, Byte für Byte.

Die Formatumwandlung ist der Ort, an dem Verlust entsteht. `LJpeg.Assign(LBitmap)` führt JPEG-Kompression durch, und JPEG ist verlustbehaftet — es verwirft Bildinformation, um Platz zu sparen, und `CompressionQuality` steuert, wie viel. Schicken Sie ein Bitmap durch JPEG und zurück, sind die zurückkommenden Pixel *nicht* die hineingegangenen. Wiederholen Sie es — dekodieren, neu kodieren, wieder dekodieren —, verschlechtert sich die Qualität mit jeder Generation.



Zwei verschiedene Schritte, von denen nur einer Daten verlieren kann

Die praktische Konsequenz: **Wenn die Originaldatei exakt erhalten bleiben muss, rekonstruieren Sie sie nicht — transportieren Sie sie.** Laden Sie die Bytes der Datei direkt von der Platte mit `TFile.ReadAllBytes` und kodieren Sie diese, statt in ein `TBitmap` zu laden und neu zu speichern. Dieser Weg berührt nie einen Codec, sodass das Ankommende bitidentisch zum Abgeschickten ist. Die Rekonstruktion über Bildklassen ist richtig, wenn Sie *transformieren wollen* — ein Vorschaubild verkleinern, für Bandbreite neu komprimieren — und falsch, wenn Sie eine Datei übertragen wollten.

Data-URIs — derselbe Trick, eine Ebene höher

Ein Ort, an dem Ihnen all das außerhalb von JSON begegnet, lohnt einen kurzen Abstecher, denn es ist dieselbe Kodierung in anderer Verpackung und macht das Debuggen deutlich leichter.

Ein Data-URI bettet eine ganze Datei in einen URL-String ein:

```
...
```

Das ist der MIME-Typ, das Wort `base64` und dann exakt der String, den unsere Hilfsfunktion erzeugt. Einen zu bauen ist String-Verkettung:

```
LDataUri := '' + GraphicToBase64(LJpeg);
```

Fügen Sie das in die Adresszeile eines beliebigen Browsers ein, und das Bild erscheint. Das ist der schnellste Weg, Ihre Kodierung zu bestätigen, ohne einen Konsumenten zu schreiben — und der Grund, warum eine API, die Base64-Bilder liefert, sie manchmal gleich als Data-URI verpackt zurückgibt, fertig für ein HTML-`src`-Attribut.

> **FireMonkey und andere Stacks**
>
> Der Code oben nutzt die VCL, aber die Form ist anderswo identisch. In FireMonkey hat `FMX.Graphics.TBitmap` eigene `LoadFromStream`/`SaveToStream`, dieselbe Hilfsfunktion funktioniert also mit ausgetauschter FMX-Unit — `TNetEncoding` liegt in `System.NetEncoding` und ist völlig framework-neutral.- Außerhalb von Delphi sind es überall dieselben drei Schritte: JavaScript hat `btoa`/`atob` plus `FileReader.readAsDataURL`, Python hat `base64.b64encode`, .NET hat `Convert.ToBase64String`, und Go hat `encoding/base64`. Weil der RFC den Algorithmus exakt festlegt, dekodiert ein von einem beliebigen davon erzeugter String in allen anderen identisch — was man sich merken sollte, wenn man eine Integration debuggt und versucht ist, den Kodierer zu verdächtigen. Es ist fast nie der Kodierer. Es sind meist Zeilenumbrüche, das URL-sichere Alphabet oder die Auffüllung.

Fazit

Der Code in diesem Beitrag ist kurz, und das ist der Punkt — sobald das Konzept vom letzten Mal klar ist, ist die Umsetzung eine Handvoll Aufrufe mit ein paar scharfen Kanten, die man kennen sollte.

- **Für JSON `TNetEncoding.Base64String` verwenden.** `TNetEncoding.Base64` fügt alle 76 Zeichen ein CRLF ein, weil es die MIME-Regel aus RFC 2045 umsetzt. Das ist richtig für E-Mail und falsch für einen JSON-Wert. `Base64URL` ist für URLs und JWTs.
- **Ein `TBitmap` hat keine Bytes, bevor Sie ein Format wählen.** In `SaveToStream` fällt die Entscheidung BMP gegen JPEG gegen PNG, und diese Wahl beeinflusst die Payload-Größe weit stärker als die 33 Prozent von Base64.
- **Komprimieren Sie vor dem Kodieren.** `TJPEGImage.Assign` plus `CompressionQuality` spart typischerweise eine Größenordnung; der Base64-Aufschlag gilt dann auf die kleinere Zahl.
- **Schicken Sie immer den Content-Type mit.** Base64 trägt keine Formatinformation, und der Empfänger sollte nicht raten müssen.
- **Prüfen Sie die kodierte Länge, bevor Sie nicht vertrauenswürdige Eingaben dekodieren.** Dekodieren ist eine Allokation, deren Größe der Absender bestimmt, und Bilddekodierer sind ein schlechter Ort für ungeprüfte Bytes.
- **Base64 ist verlustfrei; Formatumwandlung ist es nicht.** Muss eine Datei bitidentisch ankommen, lesen und kodieren Sie die Bytes der Datei selbst, statt sie über eine Bildklasse neu aufzubauen.

> Base64 gibt exakt zurück, was Sie ihm gegeben haben. Alles, was Sie zwischen einem `TBitmap` und einem JSON-Payload verlieren, haben Sie woanders verloren — meist in dem Moment, in dem Sie ein Dateiformat gewählt haben.

Damit schließt diese Serie. Wir haben mit sechs Wertetypen und einer rekursiven Regel begonnen, die Konventionen für Datumsangaben ergänzt, die JSON Ihnen überlässt, die drei RTL-Frameworks und die REST.Json-Linie durchgearbeitet, die alle verwirrt, gelernt, was Serialisierung wirklich kostet, und zum Schluss Binärdaten über eine reine Textleitung gebracht. Das Format ist wirklich klein — das war immer sein Reiz —, und die Schwierigkeit lag nie in der Grammatik. Sie lag in dem, was die Grammatik bewusst Ihnen zu entscheiden überlässt.

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 —