NEWS · JULY 24, 2026

From TBitmap to JSON and Back Again

Part six of the JSON series for Delphi developers — encoding and decoding base64 with TNetEncoding, the Base64 versus Base64String trap that breaks JSON payloads, and complete image round trips with TBitmap and TJPEGImage.

By Dr. Holger Flick

Last time we took base64 apart. We established that binary data can't go into JSON because RFC 8259 requires

UTF-8 and most bytes in an image aren't valid UTF-8; that base64 solves it by regrouping 24 bits into four 6-bit values mapped onto 64 safe characters; that this costs exactly 33%; and that it provides no security whatsoever.

That was the theory, and it was deliberately code-free. This post is the opposite: it's the working code, and because you understand the mechanism, every line of it should now look inevitable rather than magical.

We'll cover encoding and decoding with `TNetEncoding`, one genuine trap in the RTL that silently produces JSON your consumers may reject, and complete round trips with real images — `TBitmap` and `TJPEGImage` — using nothing outside the RTL and VCL. By the end you'll be able to put a picture in a JSON document and get the same picture back out, and know exactly which steps can lose data and which can't.

The encoder lives in System.NetEncoding

Everything you need is in `System.NetEncoding`, a unit that ships with Delphi and needs nothing installed.

The entry point is the `TNetEncoding` class, which exposes ready-made encoder instances as class properties. **You don't create or free these** — they're singletons managed by the RTL, which is why the calls read so cleanly:

```
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;
```

That `Man` ’`TWFu` is the exact example we traced bit by bit in the previous post — three bytes in, four characters out. Now it's two method calls.

The two methods you'll use nearly always are:

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

Note the asymmetry in the type names, because it's a deliberate and helpful design: you encode **bytes** to a **string**, and decode a **string** back to **bytes**. There's also an `Encode(const Input: string): string` overload, but reach for it carefully — it encodes the *text* of a string, which involves a character-encoding decision, and for binary payloads you want to be explicit about bytes. Stick to the two above and the types keep you honest.

The trap: Base64 versus Base64String

Now the thing that catches people, and it's a real bug rather than a style preference. `TNetEncoding` offers **three** base64 encoders, and picking the wrong one produces output that is valid base64 but problematic JSON.

```
class property Base64: TNetEncoding;         // MIME style – inserts line breaks
class property Base64String: TNetEncoding;   // one continuous line
class property Base64URL: TNetEncoding;      // URL-safe alphabet, no padding
```

The difference is visible in the RTL source. `TBase64Encoding.Create` passes `kCharsPerLine` and `kLineSeparator`, which are declared as:

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

So `TNetEncoding.Base64` **inserts a CRLF every 76 characters.** `TBase64StringEncoding.Create`, by contrast, passes `0` and `''` — no line length, no separator — producing one unbroken string.

That 76 is not arbitrary, and last post explains it: RFC 2045 requires that "the encoded output stream must be represented in lines of no more than 76 characters each." `TNetEncoding.Base64` is doing exactly the right thing **for email**, which is what it was named after. It is not the right thing for a JSON payload.

Same bytes, same algorithm — one of these is a problem in JSON

The top box is what goes wrong. Those CRLF characters are **control characters**, and as the previous post established, JSON strings cannot contain them raw — a conforming serializer must escape them as `\r\n`. So your payload is bigger, it's full of escape sequences, and — the part that actually bites — a strict consumer on the other end may reject the value or fail to decode it, because plenty of base64 decoders in other languages don't tolerate embedded whitespace.

> **The rule, stated once**
> **For JSON, always use** `TNetEncoding.Base64String`**.** Use `TNetEncoding.Base64` only when you are genuinely producing MIME content, such as an email body. Use `TNetEncoding.Base64URL` when the value goes into a URL path, a query parameter, or a JWT. This is worth grepping your codebase for. `TNetEncoding.Base64` is the name that sounds like the default, so it's what people reach for, and the resulting payloads often work fine against a tolerant decoder — right up until you integrate with one that isn't. The failure arrives late and looks like someone else's bug.

The URL variant is the same story in the other direction:

```
LToken := TNetEncoding.Base64URL.EncodeBytesToString(LBytes);
// uses '-' and '_' instead of '+' and '/', and emits no '=' padding
```

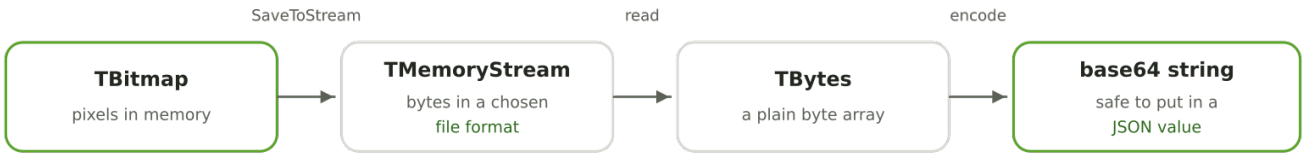
The RTL source confirms both parts: `TBase64URLEncoding.Create` passes its own alphabet tables plus `False` for padding. That matches RFC 4648's URL-safe alphabet and is why JWT tokens contain `-` and `_` and typically end without `=`.

Getting bytes out of an image

Now to images, and there's a conceptual step here that trips up otherwise-correct code.

A `TBitmap` in memory is not a file. It's a live object holding pixels, a palette, a device context. It has no inherent byte representation — the question "what are this bitmap's bytes?" has no answer until you choose a **file format** to serialize it into. BMP? PNG? JPEG? Each produces completely different bytes for the identical picture.

So the pipeline has one more stage than people expect:



Four stages, and each one is a real conversion

The second box is the one people skip mentally. `SaveToStream` is where the format decision happens, and it's the step that determines whether your 500 KB photo becomes a 2 MB BMP or a 400 KB JPEG in the payload. Base64's 33% then applies to whatever came out — so **the format choice matters far more than the encoding overhead**.

Here's the helper, and it's short:

```
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);       // format is decided by AGraphic's class
    SetLength(LBytes, LStream.Size);
    LStream.Position := 0;
    LStream.ReadBuffer(LBytes, LStream.Size);
    Result := TNetEncoding.Base64String.EncodeBytesToString(LBytes);
  finally
    LStream.Free;
  end;
end;
```

Taking `TGraphic` rather than `TBitmap` is deliberate — `TBitmap`, `TJPEGImage`, and `TPNGImage` all descend from it, so this one function handles every format, and the *class of the object you pass in* selects the encoding. That's the format decision, made explicit by your choice of type.

The `LStream.Position := 0` line is not optional. After `SaveToStream` the stream position sits at the end; forgetting to rewind reads nothing and yields an empty string. It's a classic, and it fails quietly.

Putting an image into JSON

With the helper in place, building the payload uses the object model from part three:

```
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);              // convert: bitmap pixels -> JPEG
    LJpeg.CompressionQuality := 80;     // 1..100; lower = smaller and lossier

    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;
```

Two lines deserve attention. `LJpeg.Assign(LBitmap)` is where the real size reduction happens — this is JPEG compression, and it will typically shrink a photograph by an order of magnitude compared to the raw bitmap. Base64's 33% then applies to that much smaller number. **Compressing before encoding is far more effective than worrying about base64's overhead**, and the ordering is what makes embedded images practical at all.

And `content_type` is not decoration. Base64 is format-blind — the decoder gets bytes back and has no idea whether they're a JPEG, a PNG, or a PDF. Sending the MIME type alongside is what lets the receiver know what to construct. Skipping it means the other end has to guess, and guessing is how you get sniffing bugs.

Reading it back

The reverse trip decodes to bytes, wraps them in a stream, and loads:

```
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('Expected a JSON object');
  try
    LRoot := TJSONObject(LValue);
    LData := LRoot.GetValue<string>('data', '');
```

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

    if LData = '' then
      raise Exception.Create('No image data in payload');
    if LContentType <> 'image/jpeg' then
      raise Exception.CreateFmt('Unsupported 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 takes a copy
      finally
        LJpeg.Free;
      end;
    finally
      LStream.Free;
    end;
  finally
    LValue.Free;
  end;
end;
```

`TBytesStream` is the neat part — it wraps an existing `TBytes` as a stream without copying it, which is exactly the adapter needed between `DecodeStringToBytes` and `LoadFromStream`.

The `content_type` check is doing real work. `TJPEGImage.LoadFromStream` on PNG bytes raises an exception, and while that's recoverable, checking the declared type first gives a clear error instead of a decoder failure. Note that `GetValue<string>` with a default handles the missing-key case, so the checks are about *content*, not about whether the field existed — the pattern from part three, applied.

> **Validate before you decode anything you didn't create**
>
> Decoding base64 from an untrusted source is an **allocation whose size a stranger controls**. A malicious client can send a few hundred megabytes of base64 in a field your code expects to hold a thumbnail, and `DecodeStringToBytes` will faithfully try to allocate all of it — before any of your image logic runs. Image decoders themselves have a long history of vulnerabilities when fed malformed input, which is a second reason not to hand them arbitrary bytes. A cheap and effective guard is to check the encoded length first, since base64 length is directly proportional to byte count: ``pascal const CMaxImageBytes = 2 * 1024 * 1024; // 2 MB decoded begin // 4 base64 chars per 3 bytes, so encoded length H bytes * 4/3 if Length(LData) > (CMaxImageBytes div 3) * 4 + 4 then raise Exception.Create('Image payload too large'); LBytes := TNetEncoding.Base64String.DecodeStringToBytes(LData); end; `` Reject on the *encoded* string before allocating the decoded one. Also enforce your limit server-side on the whole request body — a check in application code helps nothing if the HTTP layer already buffered a gigabyte.

What the round trip preserves — and what it doesn't

This distinction matters, and conflating the two leads to people blaming base64 for losses it didn't cause.

Base64 is exactly lossless. Encode any byte sequence and decode it, and you get the identical bytes — every time, with no exceptions. It's a pure change of representation, as the bit-regrouping diagram in the previous post showed. If you base64 a JPEG file's bytes and decode them, you have that JPEG file, byte for byte.

The format conversion is where loss happens. `LJpeg.Assign(LBitmap)` performs JPEG compression, and JPEG is lossy — it discards visual information to save space, and `CompressionQuality` controls how much. Round-trip a bitmap through JPEG and the pixels coming back are *not* the pixels that went in. Do it repeatedly — decode, re-encode, decode again — and quality degrades with each generation.

Two different steps, only one of which can lose data

The practical consequence: **if you need the original file preserved exactly, don't reconstruct it — carry it.** Load the file's bytes straight from disk with `TFile.ReadAllBytes` and base64 those, rather than loading into a `TBitmap` and re-saving. That path never touches a codec, so what arrives is bit-identical to what left. Reconstructing through image classes is right when you *intend* to transform — resize a thumbnail, recompress for bandwidth — and wrong when you meant to transmit a file.

Data URIs — the same trick, one layer up

One place you'll meet all of this outside JSON is worth a short detour, because it's the same encoding in a different wrapper and it makes debugging much easier.

A data URI embeds a whole file into a URL string:

```
...
```

That's the MIME type, the word `base64`, and then exactly the string our helper produces. Building one is string concatenation:

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

Paste that into any browser's address bar and the image appears. It's the fastest way to confirm your encoding is correct without writing a consumer — and it's why an API that returns base64 images sometimes returns them pre-wrapped as data URIs, ready to drop into an HTML `src` attribute.

> **FireMonkey, and other stacks**
> The code above uses VCL, but the shape is identical elsewhere. In FireMonkey, `FMX.Graphics.TBitmap` has its own `LoadFromStream`/`SaveToStream`, so the same helper works with the FMX unit substituted — `TNetEncoding` is in `System.NetEncoding` and is entirely framework-neutral. Outside Delphi it's the same three steps everywhere: JavaScript has `btoa`/`atob` plus `FileReader.readAsDataURL`, Python has `base64.b64encode`, .NET has `Convert.ToBase64String`, and Go has `encoding/base64`. Because the RFC pins the algorithm exactly, a string produced by any of them decodes identically in all the others — which is

> worth remembering when you're debugging an integration and tempted to suspect the encoder. It's almost never the encoder. It's usually line breaks, or the URL-safe alphabet, or padding.

Takeaways

The code in this post is short, and that's the point — once the concept from last time is clear, the implementation is a handful of calls with a few sharp edges to know about.

- **Use `TNetEncoding.Base64String` for JSON.** `TNetEncoding.Base64` inserts a CRLF every 76 characters because it implements the MIME rule from RFC 2045. That's right for email and wrong for a JSON value. `Base64URL` is for URLs and JWTs.
- **A `TBitmap` has no bytes until you pick a format.** `SaveToStream` is where BMP versus JPEG versus PNG is decided, and that choice affects payload size far more than base64's 33% does.
- **Compress before you encode.** `TJPEGImage.Assign` plus `CompressionQuality` typically saves an order of magnitude; base64 overhead then applies to the smaller number.
- **Always send the content type.** Base64 carries no format information, and the receiver shouldn't have to guess.
- **Validate the encoded length before decoding untrusted input.** Decoding is an allocation sized by whoever sent the payload, and image decoders are a poor place to send unvetted bytes.
- **Base64 is lossless; format conversion is not.** If a file must arrive byte-identical, read and encode the file's own bytes rather than rebuilding it through an image class.

> Base64 gives back exactly what you gave it. Everything you lose between a `TBitmap` and a JSON payload, you lost somewhere else — usually at the moment you chose a file format.

That closes out this series. We started with six value types and one recursive rule, added the conventions for dates that JSON leaves to you, worked through the three RTL frameworks and the REST.Json lineage that confuses everyone, learned what serialization really costs, and finished by getting binary data across a text-only wire. The format is genuinely small — that was always its appeal — and the difficulty was never in the grammar. It was in the things the grammar deliberately left for you to decide.

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 —