TECH STYLE LTD

TECH-STYLE.CO

PRODUCT BROCHURE

json-compress — Product Brochure

Lossless JSON compression that is still JSON — 45–75% smaller, six implementations, one published format.

WHAT IS INSIDE

- JSON spends most of its bytes repeating itself
- Say each thing once
- What it saves, on documents that look like real ones
- The awkward cases are the whole job
- Small enough to hold in your head
- Pull it straight from the repository
- Compress it here. Decompress it anywhere.
- Specified, so it can be ported
- Anywhere the same keys keep going past
- What people ask before they adopt it

Document	json-compress brochure
Generated	2 September 2026
Online	https://www.tech-style.co/docs/json-compress-brochure

Lossless JSON compression that is still JSON. It writes each key name once instead of once per record, turns arrays of like-shaped objects into columns and rows, and de-duplicates the values that keep coming back — typically 45–75% smaller, with a byte-perfect round trip and nothing new for your stack to learn.

Six implementations of one published format, and they write the same bytes. Compress in a Node service, restore it in a Go worker, a PHP cron job, a .NET batch, a Rust service or a Python notebook — any of them, in any direction. Three are on their registries today; the other three build from the repository.

```
$ npm install @tech-style/json-compress
```

```
$ dotnet add package TechStyle.JsonCompress
```

```
$ go get github.com/eharain/JSON-Compress/go
```

JSON spends most of its bytes repeating itself

Send a thousand records over an API and you have sent a thousand copies of every key name. A status field with four possible values has been spelled out a thousand times. A currency field that is always "GBP" has been sent a thousand times. None of that is information — it is the same handful of strings, written out again and again because JSON has no way to say “as before”.

This started as an internal helper. We were moving large result sets between a database layer and a set of front ends, and the payloads were dominated by field names. The fix was small and obvious: send the field names once, then send the rows as arrays. It worked well enough that it kept getting copied from project to project, so we wrote it properly, specified the format, and published it.

Minifying does not help with any of this. Minifying removes whitespace — the small half of the problem. What is left is the repetition, and the repetition is most of the file.

Typical saving	45–75% on record-shaped JSON, against minified
On top of gzip	A further 13–20% where gzip is already on
Round trip	Byte for byte, key order included
Dependencies	None, in any runtime

> **It is still JSON.**
> No binary framing, no base64, no custom parser. The compressed document travels anywhere the original did — an HTTP body, a WebSocket frame, a Postgres jsonb column, localStorage, a queue message, a log line.

THREE TRANSFORMATIONS

Say each thing once

Nothing here is clever. It is the same three observations anybody makes staring at a large JSON payload, applied properly and made reversible.

A key catalogue

Every distinct key name is written once, at the top of the document, and replaced throughout by a short token. Keys are ranked by how often they are used, so the busiest key gets a one-character token. A key that appears ten thousand times costs its full name exactly once.

Columnar tables

An array of like-shaped objects becomes a column list and a matrix of rows. The key names leave the rows entirely. This is the single biggest win, and it is why record sets compress far better than deeply nested configuration.

Constant and dictionary columns

A column whose value never changes is lifted out of the rows and stored once. A column drawn from a small set of strings — statuses, regions, currencies — becomes a short list plus a row of small integers.

A string catalogue

Anything else that repeats often enough to pay for itself is written once and referenced. The decision is made by measuring, not guessing: a value only enters the catalogue when the reference costs fewer bytes than the copies would.

Before

```
[ { "id": 1, "name": "Ada", "role": "admin", "team": "core" }, { "id": 2, "name": "Bob",
"role": "admin", "team": "core" }, { "id": 3, "name": "Cy", "role": "user", "team":
"core" }, { "id": 4, "name": "Dee", "role": "user", "team": "core" } ]
```

After

```
{ "jc": 1, "k": ["id","name","role","team"], "s": ["admin","user"], "d": { // columns,
encodings, constants "~t": [0,1,2,3], "~e": [0,0,1,2], "~cv": ["core"], "~r":
[[1,"Ada",0],[2,"Bob",0], [3,"Cy",1], [4,"Dee",1]] } }
```

MEASURED, NOT CLAIMED

What it saves, on documents that look like real ones

Produced by the benchmark that ships with the package. Every case is round-tripped and checked against the original before its size is reported — a benchmark that quietly loses data is not a benchmark. The compressed sizes are the same in both languages, because both encoders write the same bytes; the gzip figures move by a point or two between them, because .NET and the browser do not tune deflate identically.

Sizes for six document shapes, minified and compressed, with and without gzip

DOCUMENT	MINIFIED	COMPRESSED	SAVED	COMPRESSED + GZIP	SAVED OVER GZIP ALONE
API record set — 1,000 users	206.2 KB	54.8 KB	73%	13.3 KB	20%
Nested API response — orders with lines	96.5 KB	39.9 KB	59%	9.4 KB	6%
Time series — 2,000 readings	166.7 KB	44.3 KB	73%	11.4 KB	19%
Structured logs — 1,500 lines	237.3 KB	57.3 KB	76%	21.2 KB	13%
GeoJSON — 400 features	58.9 KB	31.6 KB	46%	6.9 KB	5%
Small config file — little repetition	3.1 KB	2.2 KB	29%	0.5 KB	22%

The last row is in the table on purpose.
Most JSON meets gzip on its way out of a server, and gzip already removes a good deal of this redundancy. On record-shaped data there is a further 13–20% to take. On a small document with little repetition, the envelope costs more than it saves — so measure() and the stats command exist to tell you which case you are in before you change anything.

The awkward cases are the whole job

Getting a 70% saving on tidy data is easy. What decides whether a compression library is usable is what it does with the documents nobody thought about — and those are exactly the documents production systems produce.

Every one of these is a real hazard that a naive implementation gets wrong, and each is covered by a test that fails if it regresses.

- Key order survives, including inside packed tables. Where no single column order can satisfy every row, the array is left as objects rather than quietly reordered.
- A missing key is not a null one. {a:1} and {a:1,b:null} in the same array stay different.
- Keys that look like numbers keep their position. Most catalogue schemes shuffle them, because JavaScript objects sort integer-like keys to the front.
- A key literally named __proto__ comes back as data, not as a mangled prototype.
- Any string at all — control characters, astral-plane characters, and strings that collide with the format's own escape.
- Circular references throw rather than hanging the process.

2,000 generated documents

Every test run builds two thousand pseudo-random documents from a seeded source — ragged rows, awkward strings, hostile key names — and compares each round trip byte for byte against what JSON itself produces. A failure reports its seed, so it can be replayed exactly.

Every option combination

The four transformations can each be turned off. All sixteen combinations are exercised against the same documents, because an option that is only correct when the others are on is not an option.

Six implementations, every push

Node 18, 20 and 22; .NET 8 and 10; Python 3.9 to 3.13; PHP 8.1 to 8.4; Go 1.21 and current; Rust 1.70 and stable — on Linux and Windows. Each job runs the tests, the build, the benchmark and a check of what the published package would actually contain, and then runs every conformance case through the other languages as well.

THE SURFACE

Small enough to hold in your head

This is the JavaScript API. The other five mirror it call for call, under each language's own naming conventions.

```
import { compress, decompress, measure } from '@tech-style/json-compress'; // still
ordinary JSON const small = compress(records); // byte for byte, key order included
const back = decompress(small); // before you change anything const report = await
measure(records); // { minified: 211145, compressed: 51386, // percent: 75.7,
gzipMinified: 15850, // gzipCompressed: 12316 }
```

- core compress(value, options?) — to an envelope
- core decompress(envelope) — back again
- core stringify(value) / parse(text) — the string-level pair
- safe decompressIfNeeded(value) — for a boundary carrying both forms

- safe isCompressed(value) — a shape check
- size measure(value) — minified, compressed, gzipped
- size gzip / gunzip / gzipSize — no dependency
- json validate(text) / locate(text) — the exact line and column
- json minify(text) / beautify(text) — whitespace only
- table pack(records) / unpack(packed) — the columnar idea alone
- bind createCodec(options) — options fixed once

On the command line

```
# what would it save? changes nothing $ json-compress stats orders.json orders.json as
given 206.2 KB minified 206.2 KB compressed 54.8 KB saved 151.4 KB 73.4% smaller than
minified minified + gzip 16.6 KB compressed + gzip 13.3 KB saved 3.3 KB 20.0% smaller
than minified + gzip
```

Anywhere in a pipeline

```
$ curl -s https://api.example.com/orders \ | json-compress compress > orders.jc.json $
json-compress decompress orders.jc.json --pretty # and the everyday ones $ json-compress
minify input.json $ json-compress beautify input.json -i 4 $ json-compress validate
input.json
```

NO INSTALL, NO BUILD STEP

Pull it straight from the repository

The built bundles are committed to the public repository, so jsDelivr serves them over a CDN. One script tag and you have the whole library — no npm account, no bundler, no toolchain.

A script tag

```
<script src="https://cdn.jsdelivr.net/gh/eharain/JSON-Compress@1.0.2/javascript/dist/
json-compress.min.js"></script> <script> const small = jsonCompress.compress(myData);
const back = jsonCompress.decompress(small); </script>
```

Or an ES module

```
<script type="module"> import { compress, measure } from 'https://cdn.jsdelivr.net/gh/
eharain/JSON-Compress@1.0.2/javascript/dist/index.mjs'; const small = compress(myData);
</script>
```

Our own tool page uses exactly this.
The online JSON minifier loads the library from that same URL and does all of its work in your browser — which is why it can promise that nothing you paste is ever uploaded.

SIX LANGUAGES, ONE FORMAT

Compress it here. Decompress it anywhere.

Most compression libraries are a private arrangement between a library and itself: whatever compressed the document has to be present to read it back. json-compress is a written specification with six implementations against it — and they do not merely interoperate, they emit the same bytes. Any one of them can be the encoder and any other the decoder, in any direction, with no shim in between.

JavaScript & TypeScript

Node 18+, Deno, Bun and every modern browser. Typed, and served from a CDN with no install at all.

.NET

net8.0, net10.0 and netstandard2.0 for .NET Framework and Unity. No dependencies on .NET 8 and later.

Python

Python 3.9 and later, pure standard library — nothing to build, no C extension, and it runs on PyPy. The PyPI release is in preparation; install from the repository meanwhile.

PHP

PHP 8.1 and later, ext-json and nothing else. Rows straight out of PDO go in without conversion. The Packagist release is in preparation; install from the VCS repository meanwhile.

Go

Go 1.21 and later, standard library only. Struct tags are honoured exactly as encoding/json honours them.

Rust

Rust 1.70 and later. Zero dependencies by default; serde and gzip are opt-in features. The crates.io release is in preparation; build from the git repository meanwhile.

Every direction, checked on every push

Cross-language interoperability matrix: every implementation restores documents compressed by every other

			PYTHON	PHP	GO	RUST
JavaScript / TypeScript	+	+	+	+	+	+
.NET	+	+	+	+	+	+
Python	+	+	+	+	+	+
PHP	+	+	+	+	+	+
Go	+	+	+	+	+	+
Rust	+	+	+	+	+	+

The same command line, six times over

```
# a Python service writes the payload $ json-compress compress orders.json >
orders.jc.json # a Go worker, a PHP cron job, a .NET batch or a Rust # service reads it
back — same command, same bytes $ json-compress decompress orders.jc.json >
restored.json $ diff orders.json restored.json # no output
```

How that is kept true

A corpus of 144 documents — the awkward cases from the specification, plus 120 generated from a seeded source every test suite reproduces independently — is checked into the repository once, each paired with the envelope the reference encoder wrote for it.

Every implementation asserts both directions against that one shared file on every CI run: it must restore all 144 envelopes to the original value, and it must write those same 144 envelopes itself. The specification only requires the first. Doing the second as well is what turns compatible into identical.

> **Written from the specification, not translated from the code.**
> Each port was implemented against SPEC.md rather than transliterated from the JavaScript — which is how the specification got proved complete: six times over, against six different sets of language constraints.

Specified, so it can be ported

The wire format is written down in full — the envelope, the two token alphabets, the escape rule, the table node, the encoder's decision points and the decoder's obligations. It is complete enough to implement from without reading the JavaScript, and it carries the conformance rules a port has to meet.

That matters because a compression format is only useful if both ends agree. A document compressed by a Node service has to read correctly in a .NET worker, a PHP job, a Go batch or a Python notebook — otherwise it is a private trick, not a format.

Six implementations are released, and each was written from the document rather than translated from the JavaScript. That is what proves the specification complete: six independent readings of it, in six languages with six different sets of constraints, arriving at byte-identical output.

Implementation status by language

LANGUAGE	REGISTRY	PACKAGE	STATUS
JavaScript / TypeScript	npm	@tech-style/json-compress	Released
.NET	NuGet	TechStyle.JsonCompress	Released
Python	PyPI	json-compress	Release in preparation
PHP	Packagist	tech-style/json-compress	Release in preparation
Go	pkg.go.dev	eharain/JSON-Compress/go	Released
Rust	crates.io	json-compress	Release in preparation
Java	Maven Central	—	Planned

WHERE IT EARNS ITS KEEP

Anywhere the same keys keep going past

Responses and payloads

Less to serialise, less to send, less to parse at the other end. Largest effect on paginated record sets, which is most of what an API returns.

Browser storage

localStorage quotas are small and unforgiving. This is a straight multiplier on how much fits, and the browser build is 7 KB over the wire.

Sockets and streams

WebSocket and SSE traffic often has no per-message compression at all, so the structural saving is the only saving available.

JSON columns

Postgres jsonb, MySQL json: smaller rows, smaller indexes, smaller backups, and the column still holds valid JSON.

Logs and telemetry

Structured log lines are almost entirely repeated keys and repeated levels, services and regions. The best case the format has.

Queues and jobs

Message brokers impose hard per-message size limits. This is often the difference between one message and a chunking scheme you have to write and maintain.

QUESTIONS

What people ask before they adopt it

Short answers. The specification and the READMEs carry the long ones.

Is the compressed output still valid JSON?

Yes. The result is an ordinary JSON document — an object with a version marker, a key catalogue, a string catalogue and the encoded document. Any JSON parser reads it, any JSON column stores it, any HTTP client sends it. Nothing in the pipeline needs to know what it is looking at, which is the whole reason the format exists.

Can I compress in one language and decompress in another?

Yes, in any direction, across all six implementations — JavaScript, .NET, Python, PHP, Go and Rust. They do not merely interoperate: given the same document they emit the same bytes, and every implementation proves it on every CI run against a shared 144-document corpus. Compress in a Python job, restore in a Go worker; compress in the browser, restore in .NET.

How much smaller does JSON actually get?

Typically 45–75% on record-shaped data — API record sets, logs, time series, exports. A thousand-user API response measured 206 KB minified and 55 KB compressed, a 73% saving. Small documents with little repetition save little or nothing, and the built-in measure() tells you which of the two you have before you commit to it.

Does it still help if I already gzip?

Usually, by a further 5–20% on record-shaped data, because the two remove different redundancy: gzip works on a sliding byte window, this works on the structure of the document. On a small config file it can end up marginally larger than gzip alone, and the size report says so rather than hiding it.

How is it different from MessagePack, CBOR, BSON or Protocol Buffers?

Those are binary formats. They are excellent, and they cost you the ability to read a payload in a log, store it in a JSON column, query it with existing tools or hand it to a service that expects JSON. Protocol Buffers additionally want a schema and a code generation step. json-compress gives up some ratio in exchange for the output remaining JSON, with no schema, no code generation and no new content type.

How is it different from gzip or Brotli?

It is not a competitor to them — it composes with them. gzip and Brotli are byte-oriented and produce opaque binary; this is structure-oriented and produces JSON. Run both and you keep transport compression exactly as it is, while removing the repetition gzip has to keep re-describing.

Is it really lossless? What about key order?

Genuinely lossless. Key order survives, including inside packed tables. A missing key stays distinct from a null one. Keys that look like numbers keep their position, a key literally named __proto__ comes back as data, and strings that collide with the format's own escape character round-trip unharmed. Every one of those is a named test case in all six suites.

What does it cost, and can I use it commercially?

It is free and MIT licensed, commercial use included, with no attribution requirement in your product. There is no paid tier, no telemetry and no network access anywhere in any implementation. The browser tool does all its work on your own machine — nothing is uploaded.

CONTACT

Talk to us

Tech Style Ltd is a UK technology company building enterprise software, procurement intelligence and AI platforms. Tell us what you are trying to do and we will tell you honestly whether we are the right people for it.

This document online	www.tech-style.co/product-json-compress
Website	www.tech-style.co
Enquiries	hello@tech-style.co

Products	www.tech-style.co/products
Partner programme	www.tech-style.co/partners

This document was generated on request.
It is built from the live content of tech-style.co at the moment you downloaded it, so it can never quote a price, a version or a status the site has since moved on from. Request a fresh copy any time at www.tech-style.co/downloads.