Engineer

JSON

the text format most data travels in

What it is

JSON is a plain-text notation for nested data, written with braces, brackets, double quotes and commas so that a program in almost any language can read back exactly what another program wrote. The letters stand for JavaScript Object Notation, because the syntax was taken from the way JavaScript writes literal values, yet a JSON document belongs to no language and needs no JavaScript to read. Files usually end in `.json`, and over HTTP the content is labelled with the media type `application/json`.

The whole format fits on one page. Four primitive types (strings, numbers, the literals `true` and `false`, and `null`) and two structured ones make up the vocabulary: an object, which is an unordered set of name and value pairs, and an array, which is an ordered list. Anything can nest inside anything else. Dates, binary data, comments and references between parts of a document are all left to conventions outside the format.

Douglas Crockford first registered it with the IETF in RFC 4627, in July 2006. The current definition is RFC 8259 from December 2017, which agrees with Ecma's standard ECMA-404 on the grammar and adds the rules that matter when two systems exchange it. Text that leaves a closed ecosystem must be UTF-8, and a sender must not add a byte order mark in front of it.

XML is older and more expressive, with attributes, namespaces and a long tradition of schema languages. YAML was revised in version 1.2 to be a strict superset of JSON, adding the comments and lighter punctuation people want in files they edit by hand. Binary encodings such as Protocol Buffers and CBOR trade readability for size and speed.

Use the concept in a decision

Put “What is JSON?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

Picking JSON is really picking a contract between two systems: the shape of what crosses the wire, and sometimes the shape of what lands in storage.

On the wire it is the default because everything already speaks it. A browser parses it without a library, every mainstream language has a mature parser, and an engineer reading a log can see what a request carried. The costs lie elsewhere. Every record repeats its field names, numbers travel as decimal text that has to be parsed back, and the format itself says nothing about which fields are required or what type each one holds. A binary, schema-first encoding such as Protocol Buffers is smaller and faster to parse, and its contract is checked by a compiler, but a person needs a tool to read a single message. For a public or browser-facing API, JSON is the sensible choice. Between internal services where volume or latency dominates, the trade can go the other way.

A schema usually supplies the missing contract. JSON Schema declares which properties an object must have, what types they hold and whether extra properties are allowed, and OpenAPI builds on it to describe whole HTTP APIs. A schema is a separate document, and it stays accurate only while something forces it to match the code.

In storage the question is whether a structure belongs in a JSON column or in real columns. PostgreSQL offers two types. `json` keeps an exact copy of the input text and reparses it every time a function touches it, while `jsonb` stores a decomposed binary form that is slightly slower to write, much faster to query and can be indexed; its documentation recommends `jsonb` for most applications. A JSON column suits data whose shape varies from row to row or belongs to the client, such as a user’s saved preferences. A field the database has to join on, constrain or sum deserves a column of its own.

What it looks like in practice

On one platform the PostgreSQL functions assemble each complete response as JSON themselves, and the Go API forwards that JSON from the database to the browser untouched. The handler takes it as a `json.RawMessage` and passes it on without decoding or re-encoding it, and a small `QueryJSON` helper made that the default. The DTOs, mappers and response structs of a layered API fell away. When a function's output changes, the new shape reaches the client with no edit to handler code.

Stored JSON brings a different problem, because it outlives the code that wrote it. One platform keeps notification settings and similar preferences as a user's saved values laid over a set of defaults, originally merged one level deep. A key added to the defaults later was absent from every row saved before it existed, so client code read an undefined field and crashed, and it crashed for the longest-standing users. The fix was a deep merge that treats the defaults as a catalogue: every key in the catalogue comes out present, nested keys included, whatever the saved blob holds. A migration applied it to existing rows at once, and the frontend dropped its defensive checks.

JSON is also simply how data arrives. For historical electricity data delivered as JSONL, one JSON event per line, the task was to turn raw events into hourly consumption per zone, and three variants were built, in Python, JQ and SQL. Python was the fastest, at 34ms for one day of data and 180ms for thirty days.

Where it goes wrong

The grammar is small enough that parsers agree on what valid text looks like. Trouble comes from everything the grammar leaves open.

A large ID comes back one digit different. The grammar puts no ceiling on numbers, but most parsers, JavaScript's among them, read them into 64-bit floating point, which holds integers exactly only up to $2^{53} - 1$. RFC 8259 names the range from $-(2^{53} - 1)$ to $2^{53} - 1$ as the one implementations agree on. Past it, identifiers change without a warning:

```
JSON.parse('{"id": 9007199254740993}').id;
// 9007199254740992
```

Send 64-bit identifiers as strings, and send money as a string or as an integer count of the smallest unit, never as a float.

Two services read the same object differently. RFC 8259 says the names in an object should be unique and leaves the rest open: given a repeated name, many parsers keep the last value, others raise an error, and some hand back every pair. When a gateway checks the first `role` and the backend acts on the second, the check protected nothing. Reject duplicate names where data enters the system. I-JSON, defined in RFC 7493, is a profile of JSON that forbids them, and a contract can cite it by name.

A hand-edited file stops parsing. Comments and trailing commas are outside the grammar, so the first person to annotate a config file, or to leave a comma after the last item, breaks it. Some tools accept a relaxed dialect, which makes the same file valid for one reader and broken for the next. Keep JSON for data machines write, and give hand-maintained files a format made for editing, such as YAML or TOML.

Text arrives garbled, or one parser rejects what another accepts. RFC 8259 requires UTF-8 between systems and forbids a byte order mark at the start. A producer that writes Latin-1, or an editor that prepends a mark, yields text whose fate depends on the parser. The grammar also allows an escape such as `"\uDEAD"`, half of a surrogate pair with its partner missing, which no valid Unicode string can hold, and libraries treat it in different ways. Encode as UTF-8 at the source and reject invalid sequences at the boundary.

One small request ties up the parser. A few kilobytes of nested brackets can drive a recursive parser thousands of levels deep, and an oversized body is parsed in full before any schema sees it. RFC 8259 lets an implementation limit both text size and nesting depth, and an API that accepts JSON from strangers should use both: cap the request size before reading, cap the depth while parsing, and check the shape only after that.

Further reading

- RFC 8259: The JSON Data Interchange Format — the current IETF definition, with its notes on numbers, duplicate names and encoding.
- ECMA-404: The JSON Data Interchange Syntax — Ecma's standard for the same grammar, which RFC 8259 cites as normative.

- RFC 7493: The I-JSON Message Format — a stricter profile that requires UTF-8 and forbids duplicate names.
- JSON Schema specification — the vocabulary for declaring which properties a document must have.
- PostgreSQL: JSON Types — `json` against `jsonb`, and what each keeps of the original text.
- Go: json.RawMessage — the type that lets a Go program delay decoding or pass JSON along untouched.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is JSON?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/json/