

PROGRAMMING COURSE

# Working with JSON

Syntax, JavaScript and Python, jq, JSON Schema, Pointer, Path and Patch, JSON Lines, other formats, security, and a capstone pipeline - every example run

12 Chapters

Philip Osztromok

Generated with Claude

WORKING WITH JSON

# Contents


| | | |
|---|---|---|
| **Chapter 1** | JSON Syntax | 2 |
| **Chapter 2** | JSON in JavaScript | 13 |
| **Chapter 3** | JSON in Python | 26 |
| **Chapter 4** | jq Basics | 38 |
| **Chapter 5** | jq Reshaping | 52 |
| **Chapter 6** | Schema Basics | 67 |
| **Chapter 7** | Validating | 80 |
| **Chapter 8** | Pointer, Path, Patch | 93 |
| **Chapter 9** | Lines and Scale | 105 |
| **Chapter 10** | Other Formats | 117 |
| **Chapter 11** | Security and Pitfalls | 129 |
| **Chapter 12** | Capstone | 141 |

# Working with JSON

Chapter 1 · What JSON Is: Syntax, Types and Rules

JSON looks like a JavaScript object and is read by every language, so people treat it as "obvious" and stop checking. Then a file with a stray comma fails, one parser accepts `NaN` while another rejects it, and an ID quietly changes its last digits. This chapter is the small rulebook behind all of that. You will learn the six value types, the exact list of things JSON forbids, the rules for numbers and strings, where two real parsers disagree, and how to read their error messages. The rest of the course (JavaScript, Python, `jq`, JSON Schema) builds on this.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Node 24.18.0 (`JSON.parse`) and Python 3.14.0 (`json` and `python -m json.tool`) on Windows 11, in a scratch folder outside this site. Error messages are copied from real output. They are **the messages of those versions**: other Node versions, Firefox, Safari and other Python versions word them differently. **Not run:** VS Code, Firefox and Safari. Statements about the standards are checked against RFC 8259 (the text is quoted where it matters).

## What JSON Is

JSON (JavaScript Object Notation) is a *text format* for data. A JSON document is just characters: it is not an object, and it is not JavaScript. A program **parses** the text into its own values (an object in JavaScript, a `dict` in Python) and **serialises** values back into text. It is defined twice, in matching terms: RFC 8259 from the IETF and ECMA-404 from Ecma International. Its media type is `application/json`, and RFC 8259 says text exchanged between systems outside a closed ecosystem *must* be encoded as UTF-8.

The format is deliberately tiny: six kinds of value, and a handful of punctuation marks. Here is one document that uses all six:

```
{
  "name": "Ada Lovelace",
  "born": 1815,
  "mathematician": true,
  "spouse": null,
  "languages": ["English", "French"],
```

```
  "address": { "city": "London", "postcode": null }
}
```

| TYPE | WRITTEN AS | NOTES |
|---|---|---|
| string | `"text"` | Double quotes only; Unicode; escapes with a backslash |
| number | `42` `-3.5` `1e3` | One number type, written in decimal; no hex, no `NaN`, no `Infinity` |
| boolean | `true` `false` | Lower case only |
| null | `null` | Lower case only; there is no `undefined` |
| array | `[1, "two", null]` | Ordered; items may be of different types |
| object | `{"key": value}` | Keys are always strings in double quotes; the keys *should* be unique |

A whole JSON text is *one value*. RFC 8259 says "A JSON text is a serialized value", and older specifications required an object or array at the top. Today `42`, `"hello"` and `null` are each a valid document. I parsed all six kinds at the top level in Node and each came back as the matching type; only `[]` reported as an array. Some older tools still insist on an object or array at the top, so an API that returns a bare string may meet a parser that rejects it.

## What JSON Forbids

Most JSON errors are people writing things that are fine in JavaScript, or in a hand-edited config file, but are not in the JSON grammar. I fed each of these to `JSON.parse` in Node and `json.loads` in Python and recorded the message:

| WHAT YOU WROTE | NODE 24.18 SAID | PYTHON 3.14 SAID |
|---|---|---|
| `{"a":1 /* c */}` a comment | Expected ',' or '}' after property value in JSON at position 7 (line 1 column 8) | Expecting ',' delimiter: line 1 column 8 (char 7) |
| `{"a":1,}` trailing comma | Expected double-quoted property name in JSON at position 7 (line 1 column 8) | Illegal trailing comma before end of object: line 1 column 7 (char 6) |
| `[1,2,]` trailing comma | Unexpected token ']', "[1,2,]" is not valid JSON | Illegal trailing comma before end of array: line 1 column 5 (char 4) |

| WHAT YOU WROTE | NODE 24.18 SAID | PYTHON 3.14 SAID |
|---|---|---|
| `{'a':1}` single quotes | Expected property name or '}' in JSON at position 1 (line 1 column 2) | Expecting property name enclosed in double quotes: line 1 column 2 (char 1) |
| `{a:1}` unquoted key | Expected property name or '}' in JSON at position 1 (line 1 column 2) | Expecting property name enclosed in double quotes: line 1 column 2 (char 1) |
| `{"a":undefined}` | Unexpected token 'u', "{"a":undefined}" is not valid JSON | Expecting value: line 1 column 6 (char 5) |
| `{"a":01}` leading zero | Unexpected number in JSON at position 6 (line 1 column 7) | Expecting ',' delimiter: line 1 column 7 (char 6) |
| `{"a":+1}` plus sign | Unexpected token '+', "{"a":+1}" is not valid JSON | Expecting value: line 1 column 6 (char 5) |
| `{"a":0x1F}` hex number | Expected ',' or '}' after property value in JSON at position 6 (line 1 column 7) | Expecting ',' delimiter: line 1 column 7 (char 6) |
| `{"a":NaN}` | Unexpected token 'N', "{"a":NaN}" is not valid JSON | **accepted**: `{'a': nan}` |
| a raw tab or newline inside a string | Bad control character in string literal in JSON at position 7 (line 1 column 8) | Invalid control character at: line 1 column 8 (char 7) |
| `{"a":1}{"b":2}` two values | Unexpected non-whitespace character after JSON at position 7 (line 1 column 8) | Extra data: line 1 column 8 (char 7) |
| an empty string | Unexpected end of JSON input | Expecting value: line 1 column 1 (char 0) |

So the full list of things people try and JSON does not allow: **comments**, **trailing commas**, **single-quoted strings**, **unquoted keys**, `undefined`, **numbers with a leading zero, a leading plus, a leading or trailing dot, or in hex**, `NaN` **and** `Infinity`, and **raw control characters inside strings**. The one surprise in the table is Python's `NaN`, covered below. Comments are the one people miss most: if you want them in a config file, JSON is the wrong format, or you need a superset such as JSONC or JSON5 (*Working with JSON 10* compares them).

## Numbers

JSON numbers are written in decimal: an optional minus, digits (no leading zero unless the number is just `0`), an optional fraction, an optional exponent. The standard sets *no limit* on size or precision: RFC 8259 says implementations "set limits on the range and precision of numbers accepted". That sentence is where the trouble starts, because each language picks its own limit. I parsed the same four numbers in both:

| JSON TEXT | NODE (JAVASCRIPT NUMBER) | PYTHON |
|---|---|---|
| `12345678901234567890` | 12345678901234567000 (digits lost) | 12345678901234567890 (`int`, exact) |
| `9007199254740993` ($2^{53}+1$) | 9007199254740992 (off by one) | 9007199254740993 (exact) |
| `1e400` | `Infinity` | `inf` |
| `-0` | `-0` (kept; `Object.is(x, -0)` is true) | `0` (an `int`) |
| `1.0` | `1` (JavaScript has no 1.0) | `1.0` (a `float`) |

The same JSON text gives different numbers in different languages. The practical rule: a JavaScript number is a 64-bit float, so integers are exact only up to 9007199254740991 (`Number.MAX_SAFE_INTEGER`). An ID larger than that, read by JavaScript, silently changes. That is why large IDs are often sent as strings. *Working with JSON 11* returns to this with a demonstration. Notice also that `1e400` was *accepted* by both parsers and became infinity, even though the standard has no infinity. Parsing succeeds; what happens to the value is the language's business.

## Strings and Escapes

A string is wrapped in **double quotes** and can contain any Unicode character except the quote, the backslash and the control characters (code points below U+0020), which must be escaped. There are exactly eight short escapes plus the `\uXXXX` form. I parsed each short escape inside a string and read back the character code:

| WRITE | MEANS | CODE READ BACK |
|---|---|---|
| `\"` | double quote | 34 |

| WRITE | MEANS | CODE READ BACK |
|---|---|---|
| `\\` | backslash | 92 |
| `\/` | forward slash (optional; `/` needs no escape) | 47 |
| `\b` `\f` `\n` `\r` `\t` | backspace, form feed, newline, carriage return, tab | 8, 12, 10, 13, 9 |
| `\u00e9` | the character with that 4-digit hex code (here é) | e9 (hex) |

Anything else after a backslash is an error. I tried three: `\x41` (the JavaScript hex escape) gave `Bad escaped character in JSON at position 7`; `\u41`, with only two digits, gave `Bad Unicode escape in JSON at position 10`; and the capital `\U0041` gave `Bad escaped character` too. JSON has no `\x`, no `\'` and no `\0`.

**Characters outside the first 65,536.** `\uXXXX` holds only four hex digits, so something like 😜 (code point U+1F600) is written as a *surrogate pair*: two escapes in a row, `\ud83d\ude00`. Both parsers turned that pair back into one character with code 1f600. You may instead write the emoji itself, as long as the file is UTF-8; the result is identical. The grammar also allows a *lone* surrogate such as `\ud83d`, and both Node and Python accepted it and produced a string holding the broken half (code d83d). RFC 8259 warns that software receiving such text behaves "unpredictably", so do not produce it, and do not trust that it survives being re-encoded as UTF-8.

## Whitespace, Duplicate Keys and the Byte Order Mark

- **Whitespace.** Space, tab, newline and carriage return are allowed between any two tokens, so a document can be one line or pretty-printed. Other whitespace is *not* allowed: a non-breaking space (U+00A0) before a value gave `Unexpected token ' '`, and a vertical tab after the value gave `Unexpected non-whitespace character after JSON`. This is the reason a document pasted from a web page or word processor sometimes fails with nothing visibly wrong.
- **Duplicate keys.** `{"a":1,"a":2}` is accepted by the grammar, and both parsers returned `a` = 2 (the last one wins). RFC 8259 only says the names "SHOULD be unique" and that duplicates make the result "unpredictable", so another parser may keep the first or reject it. Treat duplicates as a bug in the data.

- **Byte order mark.** A text file saved as "UTF-8 with BOM" starts with the invisible character U+FEFF. RFC 8259 says writers "MUST NOT add" it and parsers "MAY ignore" it. Both parsers I ran rejected it: Node with `Unexpected token` and the invisible character in the message, and Python with `Unexpected UTF-8 BOM (decode using utf-8-sig)`. Save JSON as UTF-8 *without* BOM.

## JSON Is Not a JavaScript Object

The name suggests JSON is "JavaScript objects as text". It began as a subset of the language, but today they are different things, and mixing them up is the cause of most of the forbidden-list errors above. I wrote one real JavaScript object and asked `JSON.stringify` to make JSON from it:

```
const lit = {
  name: 'Ada', 'x-y': 1,
  f() { return 1; },
  u: undefined, n: NaN, i: -Infinity,
  d: new Date(0), nested: { ok: true },
};
console.log(JSON.stringify(lit));
```

```
{"name":"Ada","x-y":1,"n":null,"i":null,"d":"1970-01-01T00:00:00.000Z","nested":{"ok":tr
```

| | JAVASCRIPT OBJECT LITERAL | JSON TEXT |
|---|---|---|
| **Keys** | Unquoted if a valid name; single, double or no quotes | Always double-quoted strings |
| **Strings** | Single, double or backtick | Double quotes only |
| **Comments, trailing commas** | Allowed | Forbidden |
| **Values** | Functions, `undefined`, `NaN`, `Infinity`, dates, anything | Only the six types; `NaN`, `Infinity` and `undefined` are gone, a function is dropped |
| **What it is** | Live data inside a running program | Text, to be parsed |

Look at what happened to the object: the method `f` and the property `u` vanished, `NaN` and `-Infinity` became `null`, and the date turned into a string. Serialising is lossy, and Chapter 2 maps every case. The point for now: **copying a JavaScript literal into a `.json` file does not produce JSON.** A file that works as JavaScript, such as `{ name: 'Ada', }`, is rejected by every JSON parser.

## Parsers Disagree

Because the standard leaves room, two correct parsers can give different answers for the same text. I ran twelve inputs through Node and Python and compared them side by side. Where the cell says **differs**, the two results were not the same:

| INPUT | NODE 24.18 | PYTHON 3.14 | |
|---|---|---|---|
| `{"a":NaN}` | error | ok, `nan` | differs |
| `[Infinity]` | error | ok, `inf` | differs |
| `[1e400]` | ok, Infinity | ok, `inf` | same |
| `[12345678901234567890]` | ok, digits lost | ok, exact | differs |
| `[9007199254740993]` | ok, becomes ...992 | ok, exact | differs |
| `[-0]` | ok, `-0` | ok, `0` | differs |
| `[1.0]` | ok, `1` | ok, `1.0` (float) | differs |
| `{"a":1,"a":2}` | ok, 2 | ok, 2 | same |
| lone surrogate | ok, code d83d | ok, code d83d | same |
| surrogate pair | ok, code 1f600 | ok, code 1f600 | same |
| BOM first | error | error | same |
| 10,000 levels of `[` nesting | ok | ok | same |

Python's behaviour on `NaN` and `Infinity` is a documented extension, not a bug: the module accepts them by default, so a Python program can write JSON that a JavaScript program cannot read. *Working with JSON 3* shows how to switch it off. Nesting depth is another place parsers

differ: both of mine handled 10,000 levels, but I only tested those two, and other parsers set a lower limit and fail. If data crosses a language boundary, test it on both sides.

## Reading a Parser's Error Message

Every message above has the same three parts: **what it expected or found**, **where** (a position, or a line and column) and sometimes **a copy of the text**. Three habits make them much more useful. First, **the position is where the parser noticed, not always where the mistake is.** In `{"a":01}` the real mistake is the leading zero at column 6, but both parsers complain about column 7, the `1` after the zero. Second, **the same mistake is described differently in each language**: Node and Python worded the single quote, the plus sign and the comment differently, so search for the position, not the words. Third, **fix errors one at a time from the top**, because everything after the first mistake is unreliable.

I took a file with six errors in it and fixed them in the order the parser reported them, printing each message with a caret under the position:

```
{
  // the shop's config
  'name': "Corner Shop",
  "open": True,
  "hours": [9, 17,],
  "discount": 010,
  "owner": {"first": "Ada" "last": "Lovelace"}
}
```

```
round 0: line 2 col 3: Expecting property name enclosed in double quotes      (the comme
round 1: line 2 col 3: Expecting property name enclosed in double quotes      (the singl
round 2: line 3 col 11: Expecting value                                       (True)
round 3: line 4 col 18: Illegal trailing comma before end of array
round 4: line 5 col 16: Expecting ',' delimiter                               (010)
round 5: line 6 col 28: Expecting ',' delimiter                               (missing c
round 6: valid
```

Rounds 0 and 1 give the *same* message at the *same* position, because after the first fix the next problem sits on the same spot. Python's message in round 0 does not mention comments at all; it says it wanted a property name, and a comment is simply not one. When a message seems to describe something else, look at that exact column.

## Formatting and Checking JSON

- **Pretty-print in JavaScript:** `JSON.stringify(value, null, 2)` indents with two spaces. It makes JSON from a value; to check a *text*, call `JSON.parse(text)` in the browser console or Node and see whether it throws.
- **From a terminal, with Python installed:** `py -m json.tool file.json` (on Linux and macOS, `python3 -m json.tool file.json`) prints it re-indented (four spaces), or prints the error and exits with status 1. I ran it on a valid file and on one with a bad comma; the bad one printed `Expecting value: line 2 column 11 (char 18)` and exited 1. `--compact` prints it on one line, and the valid file came back as `{"name":"Ada","tags":["x","y"],"n":{"k":null}}`.
- **In VS Code:** a file with the `.json` extension is checked as you type, with red marks and a Problems entry for errors. I did not run VS Code for this chapter, so I cannot show the messages. Some editors also accept comments in files they treat as "JSON with Comments" (JSONC); that is not JSON, and I did not check which of your files VS Code treats that way.
- **The command line, later:** `jq` (Chapter 4) also reports an error with a line and column and a non-zero exit status.

## Where This Fits

JSON is the usual body format for the HTTP APIs described in *API Types & Design* (REST and HTTP, chapters 2 to 4); how to send and inspect such requests is in *API Testing & Tooling* (cURL, chapters 6 and 7). This course stays on the data format itself.

MISTAKES WHILE PREPARING THIS CHAPTER

**1. My first escape tests were not testing JSON at all.** I wrote `\x41` and `\u00e9` inside ordinary JavaScript strings, so JavaScript turned them into `A` and `é` before `JSON.parse` ever saw them, and the "bad escape" case appeared to pass. **How to avoid it:** when testing a language's own escapes, build the text with `String.fromCharCode(92)` (a backslash) or `String.raw`, and print the text you are about to parse to check it still holds a backslash.

**2. My first "negative zero" line printed `0`.** `JSON.parse` did keep `-0`; it was `JSON.stringify(-0)`, which prints `0`, that hid it. **How to avoid it:** check a value with `Object.is(x, -0)`, not by printing a serialised copy.

**3. I assumed the two parsers would agree on all twelve inputs.** They disagreed on six. **How to**

**avoid it:** when two systems share a format, run the same samples through both before you write "interoperable" in a design.

## Hands-On Exercises

**EXERCISE 1**

Write a Node script with about twenty candidate texts (plain objects, top-level values, single quotes, unquoted keys, trailing commas, comments, `undefined`, `NaN`, leading zeros, hex, a raw tab, an empty string, duplicate keys). Write your **prediction** (valid or not) next to each, run `JSON.parse` on all of them, print the error message for each failure and count wrong predictions. Then flip one prediction on purpose to show the counter works.

View solution

**EXERCISE 2**

Parse the same twelve inputs (NaN, Infinity, 1e400, a 20-digit integer, $2^{53} + 1$, `-0`, `1.0`, duplicate keys, a lone surrogate, a surrogate pair, a BOM, 10,000 levels of nesting) with Node and with Python, print one comparable line per input from each, and list the inputs on which the two languages disagree. Build the escape texts without writing a backslash inside a string literal.

View solution

**EXERCISE 3**

Take a file with six mistakes (a comment, single-quoted key, `True`, a trailing comma, a leading-zero number, a missing comma). Write a Python loop that parses it, prints the error line, column, message and a caret under the position, applies one fix and tries again until it is valid. Then run `py -m json.tool` on a valid and an invalid file and report the exit status of each.

View solution

**CHAPTER 1 QUICK REFERENCE**

- JSON is **text**, defined by RFC 8259 and ECMA-404; media type `application/json`; UTF-8 when exchanged between systems; a whole document is one value (even `42` or `"hi"`)
- Six types: string, number, `true` / `false`, `null`, array, object (keys always double-quoted strings)
- Forbidden: comments, trailing commas, single quotes, unquoted keys, `undefined`, leading zeros or plus sign or hex in numbers, `NaN` / `Infinity` (Python accepts them by default), raw control characters in strings, more than one value per document
- Escapes: `\" \\ \/ \b \f \n \r \t \uXXXX` only; no `\x41`, no `\U0041`; characters above U+FFFF are a surrogate pair (`\ud83d\ude00`) or just written in UTF-8
- Numbers have no size limit in the standard, but JavaScript keeps 15 to 17 digits: 12345678901234567890 became 12345678901234567000, and $2^{53} + 1$ became $2^{53}$; Python kept both exactly
- Duplicate keys: both parsers kept the last; the RFC says the result is unpredictable. A BOM at the start made both fail; save without it. Only space, tab, newline and carriage return count as whitespace
- A JavaScript object literal is not JSON; `JSON.stringify` dropped functions and `undefined` and turned `NaN` and dates into `null` and strings
- Error positions point where the parser noticed the problem; fix from the top, one at a time. Check a file with `py -m json.tool file.json` (exit status 1 on error)

**COMING NEXT**

Chapter 2 is JSON in JavaScript: `JSON.parse` and `JSON.stringify` in depth, the indent, replacer and reviver arguments, `toJSON`, everything that silently disappears or changes on the way out, circular references, `fetch` and `response.json()`, and when `structuredClone` is the better copy. Start with *Working with JSON 2*.

# Working with JSON

Chapter 2 · Parsing and Generating in JavaScript

JavaScript has exactly two built-in JSON functions: `JSON.parse` turns text into values, and `JSON.stringify` turns values into text. Both take optional arguments that most people never use, and `stringify` quietly *changes or drops* things it cannot represent. This chapter covers every argument, lists exactly what is lost (as measured, not remembered), and then moves to the places JSON meets JavaScript in real programs: `fetch`, copying objects, and circular data. It assumes the six types and the rules from *Working with JSON 1*; basic JavaScript syntax comes from *JavaScript Fundamentals*.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Node 24.18.0 on Windows 11. The `fetch` examples run against small HTTP servers started inside the same script, on `127.0.0.1` (a random port each time, so port numbers in the output differ from yours). The two newest features (`JSON.rawJSON` and the reviver's `context` argument) were also checked in headless Chrome 154, where both exist. **Not run:** Firefox, Safari, older Node versions, and any real remote API. Check MDN's compatibility tables before relying on those two features in a browser you support.

## The Two Functions

```
const value = JSON.parse(text, reviver);              // text -> value (reviver optiona
const text  = JSON.stringify(value, replacer, indent);    // value -> text (replacer, in
```

`JSON.parse` throws a `SyntaxError` for bad text, so any text that came from outside your program belongs in a `try` block. `JSON.stringify` returns a string, except that it returns `undefined` (not a string, and not `"undefined"`) when the top-level value is `undefined`, a function or a symbol. I checked all three: `JSON.stringify(undefined)`, `JSON.stringify(() => 1)` and `JSON.stringify(Symbol('x'))` each returned `undefined`. Code that does `.length` on the result of a stringify of an unknown value can crash on that.

## Pretty-Printing with the Third Argument

The third argument, `indent`, is a number of spaces or a string to indent with. Without it you get the compact form. I printed one object four ways:

```
const o = { a: 1, b: [1, 2], c: { d: null } };
console.log(JSON.stringify(o));
console.log(JSON.stringify(o, null, 2));
console.log(JSON.stringify(o, null, '--'));
```

```
{"a":1,"b":[1,2],"c":{"d":null}}
{
  "a": 1,
  "b": [
    1,
    2
  ],
  "c": {
    "d": null
  }
}
{
--"a": 1,
--"b": [   (first lines of the third form)
```

Three details I measured. The second argument is `null` when you want the indent but no replacer. A number above 10 is capped (`20` produced 10 spaces), and a string indent is cut to its first 10 characters. Empty containers stay on one line: `JSON.stringify([], null, 2)` gave `[]`, and a nested empty array printed as `[]` on its own line. Use compact output for network traffic and pretty output for files people read.

## What `stringify` Drops or Changes

I serialised one object holding most of the awkward values in JavaScript and read the result back:

| VALUE IN THE OBJECT | APPEARS IN THE JSON AS | NOTES |
|---|---|---|
| `undefined` (property) | (the key is omitted) | But in an *array* it becomes `null` |
| a function / method (property) | (omitted) | In an array: `null` |

| VALUE IN THE OBJECT | APPEARS IN THE JSON AS | NOTES |
| --- | --- | --- |
| `Symbol('s')` (property) | (omitted) | In an array: `null` |
| `NaN`, `Infinity` | `null` | Silent: no error |
| `-0` | `0` | The sign is lost |
| `new Date(...)` | `"2020-01-02T03:04:05.000Z"` | Through its `toJSON`; an invalid date gives `null` |
| `/x/g` (RegExp), `new Error('boom')` | `{}` | Nothing of their content is kept |
| `new Map([[1,2]])`, `new Set([1])` | `{}` | Empty object; the data is gone |
| `new Uint8Array([1,2])` | `{"0":1,"1":2}` | An object with index keys, not an array |
| `new Number(5)`, `new String('s')`, `new Boolean(false)` | `5`, `"s"`, `false` | Wrapper objects are unwrapped |
| `10n` (BigInt) | **throws** `TypeError: Do not know how to serialize a BigInt` | The one value that fails loudly |

The pattern to remember: **most things are lost silently, and the program carries on with less data.** The array rule matters too: dropping an `undefined` from an array would shift the other items, so `stringify` writes `null` in its place (my array `[undefined, function(){}, Symbol('q'), NaN]` came out as `[null,null,null,null]`).

## Controlling Output with `toJSON`

If a value has a method named `toJSON`, `stringify` calls it and serialises *what it returns*. That is how `Date` works (its `toJSON` returns the ISO string), and you can give your own classes the same behaviour. The method receives the *key* the value was found under:

```
class Money {
  constructor(cents) { this.cents = cents; }
  toJSON(key) { return { amount: (this.cents / 100).toFixed(2), key }; }
```

```
}
console.log(JSON.stringify({ price: new Money(1999), list: [new Money(5)] }));
```

```
{"price":{"amount":"19.99","key":"price"},"list":[{"amount":"0.05","key":"0"}]}
```

Note that the key for an array item is its index as a string (`"0"`). `toJSON` runs *before* the replacer, so the replacer sees the already-converted value (for a date, the string, not a `Date`), which matters in the next section.

## The Replacer: an Array or a Function

The second argument filters or transforms. As an **array** it is a whitelist of property names. I tried it on an object with a `password` at two levels:

```
const user = { id: 1, name: 'Ada', password: 'x',
  profile: { name: 'inner', age: 36, password: 'y' } };
JSON.stringify(user, ['id', 'name', 'profile']);
JSON.stringify(user, ['id', 'name', 'profile', 'age']);
```

```
{"id":1,"name":"Ada","profile":{"name":"inner"}}
{"id":1,"name":"Ada","profile":{"name":"inner","age":36}}
```

The whitelist applies **at every depth**: `age` inside `profile` disappeared until I added it to the list. That makes the array form easy to get wrong with nested data, and a whitelist can be wrong in the dangerous direction if you forget a key, because the omission is silent.

As a **function**, the replacer is called for every key and value, starting with the whole value under the key `""` (an empty string). Return the value to keep it, return something else to replace it, or return `undefined` to drop the property. Inside it, `this` is the object or array that holds the key. For `{ a: [1, { b: 2 }] }` I logged the calls:

```
JSON.stringify(user, (key, value) => key === 'password' ? undefined : value);
JSON.stringify({ n: 1, s: 'x', big: 5n }, (key, value) => typeof value === 'bigint' ? va
```

```
{"id":1,"name":"Ada","profile":{"name":"inner","age":36}}
{"n":1,"s":"x","big":"5"}
```

```
calls for { a: [1, { b: 2 }] }, in order:
  ""  this=object;  "a"  this=object;  "0"  this=array;  "1"  this=array;  "b"  this=obj
```

The function form removed `password` at both levels without listing the keys to keep, and it fixed the BigInt that would have thrown. It runs from the outside in: the whole value first, then each property. A function replacer is the right tool for "remove secrets" (a blacklist), and the array is right for "send only these fields" when the shape is flat.

## The Reviver: Rebuilding Types on the Way In

`JSON.parse` only ever produces objects, arrays, strings, numbers, booleans and `null`. There is no way for JSON to say "this string is a date". The second argument, the *reviver*, is a function called for each value after parsing, so you can convert. I logged the order of the calls:

```
const text = '{"name":"Ada","born":"1815-12-10T00:00:00.000Z","tags":["a","b"],"n":{"x":
const r = JSON.parse(text, (key, value) => key === 'born' ? new Date(value) : value);
console.log(r.born instanceof Date, r.born.getUTCFullYear());
```

```
visit order: name born 0 1 tags x n (root)
true 1815
```

The reviver works **from the inside out**: children before parents, and the whole value (key `""`) last, which is the opposite of the replacer. Returning `undefined` deletes the property: `{"a":1,"b":2,"c":3}` with `b` removed gave `{ a: 1, c: 3 }`, and doing the same to index `1` of `[1,2,3]` left a hole, `[ 1, <1 empty item>, 3 ]`.

> **A REVIVER THAT MATCHES BY KEY NAME MATCHES AT EVERY DEPTH**
>
> A rule such as `key === 'id'` converts *every* property called `id`, in every nested object (in my test, an order line's `"A-77"` became `77n`). If only the top-level `id` should change, test the nesting too, or convert after parsing instead. A string that merely looks like a date (an order number such as `"2026-10-12"`) will also be converted if you match by value, so match by key where you can.

**Big numbers.** Recall that JavaScript read `12345678901234567890` as `12345678901234567000`. Recent engines pass the reviver a third argument, `context`, whose `source` holds the *original text* of a primitive value. On Node 24.18 and Chrome 154, `context` was

`{"source":"12345678901234567890"}`, so the exact digits can be turned into a BigInt. The matching writer is `JSON.rawJSON`, which inserts a number's own digits into the output; `JSON.isRawJSON` tests for one:

```
const text = '{"id":12345678901234567890,"small":42}';
console.log(JSON.parse(text).id);
console.log(JSON.parse(text, (k, v, ctx) => k === 'id' ? BigInt(ctx.source) : v).id);
console.log(JSON.stringify({ id: JSON.rawJSON('12345678901234567890') }));
```

```
12345678901234567000
12345678901234567890n
{"id":12345678901234567890}
```

`rawJSON` accepts only a primitive's JSON text: given an object, `{"a":1}`, it threw a `SyntaxError`, and a number with a leading space (`" 1"`) also threw. On older runtimes these functions may not exist, so test `typeof JSON.rawJSON === 'function'` before relying on them (I only tested the two runtimes named above).

## Circular References

JSON is a tree, and an object that contains itself, directly or through a chain, has no finite text. `stringify` detects this and throws:

```
const a = { name: 'a' }; a.self = a;
try { JSON.stringify(a); } catch (e) { console.log(e.message); }
```

```
TypeError: Converting circular structure to JSON
    --> starting at object with constructor 'Object'
    --- property 'self' closes the circle
```

For a longer chain the message names each step (`property 'child' -> object with constructor 'Object'`, then `property 'parent' closes the circle`), which is a good map of what to fix. An object that is merely *shared* (the same object used in two places) is **not** circular: `{ a: shared, b: shared }` serialised fine, writing the content twice.

A common fix on the web is a replacer that remembers every object it has seen and returns `"[Circular]"` for repeats. **That version is wrong:** on the shared (not circular) example it

replaced the second use with `"[Circular]"` and lost data. The correct test is "is this object one of my own *ancestors*?". I wrote that version, using the fact that `this` is the current holder, and tried it on the circle, on the shared object, and on a deeper structure that has both a shared object and a cycle back to an ancestor:

```
const circularOnly = () => {
  const path = [];                                  // the chain of objects from the r
  return function (key, value) {
    while (path.length && path[path.length - 1] !== this) path.pop();  // climb back to
    if (value && typeof value === 'object') {
      if (path.includes(value)) return '[Circular]';
      path.push(value);
    }
    return value;
  };
};
```

```
circle              {"name":"a","self":"[Circular]"}
shared              {"a":{"z":1},"b":{"z":1}}
deeper structure    {"list":[{"z":1},{"inner":{"z":1},"deep":{"again":{"z":1},"up":"[Cir
```

It gave the right answer on all three. That is three tests, not a proof, so use a maintained library if circular data is a regular part of your work. Often the better fix is in the data model: store an id instead of a back-reference.

## `fetch` and `response.json()`

`response.json()` reads the body and runs `JSON.parse` on it, so everything above applies, and it adds new ways to fail. I started one small local server with six routes and called each route two ways: the shortcut everyone writes, `(await fetch(url)).json()`, and a careful helper. (How HTTP itself works is in *API Types & Design*; sending these requests from a terminal is in *API Testing & Tooling*.)

```
async function getJson(url) {
  const res  = await fetch(url);
  const type = res.headers.get('content-type') ?? '';
  const body = await res.text();                    // read once, as text
```

```
  if (!res.ok) throw new Error('HTTP ' + res.status + ': ' + (type.includes('json') ? bo
  if (res.status === 204 || body === '') return null;
  try { return JSON.parse(body); }
  catch (e) { throw new Error('bad JSON (' + e.message + '), starts with ' + JSON.string
}
```

| ROUTE AND WHAT IT SENDS | `(await fetch(u)).json()` | `getJSON(u)` |
|---|---|---|
| /ok — 200, JSON | {"id":7,"name":"Ada"} | {"id":7,"name":"Ada"} |
| /notfound — 404, JSON error body | {"error":"no such user"} (**looks like success**) | throws `HTTP 404: {"error":"no such user"}` |
| /html — 502, HTML page | throws `Unexpected token '<', "<html> <bod"... is not valid JSON` | throws `HTTP 502: <html><body>Bad Gateway</body></html>` |
| /empty — 204, no body | throws `Unexpected end of JSON input` | null |
| /badjson — 200, text cut off | throws `Expected double-quoted property name in JSON at position 8` | throws `bad JSON (...), starts with "{\"id\":7,"` |
| /wrongtype — 200, `text/plain` but valid JSON | {"id":8} (the content type is ignored) | {"id":8} |

Three lessons from the table. `fetch` **does not fail on 404 or 502:** only a network failure rejects the promise, so you must test `res.ok` or the status yourself, and a 404 whose body is JSON sails straight through the shortcut. `.json()` **ignores the** `Content-Type`**:** it parsed a `text/plain` body without complaint, so the header is no safeguard. And **the "unexpected token <" error means you received HTML**, usually an error page, so the first characters of the body are the most useful thing to log. The helper reads the body as text once and parses it itself, which gives it the text to quote in an error. A body can only be read once: a second `await r.json()` on the same response threw `TypeError: Body is unusable: Body has already been read`.

**Sending JSON** has its own trap. I posted the same body to an echo server three ways:

```
fetch(url, { method: 'POST', body: JSON.stringify({ a: 1 }) });
fetch(url, { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: JSON
```

```
fetch(url, { method: 'POST', headers: { 'Content-Type': 'application/json' }, body: { a:
```

```
no header      -> server saw content-type text/plain;charset=UTF-8, body {"a":1}
with header    -> server saw content-type application/json,        body {"a":1}
object as body -> server saw content-type application/json,        body [object Object]
```

Without the header, `fetch` labels a string body `text/plain`, and a server that checks the type may refuse it. Passing an object instead of a string sent the literal text `[object Object]`: `fetch` does not call `JSON.stringify` for you.

## Copying: JSON Round-Trip or `structuredClone`

`JSON.parse(JSON.stringify(x))` is a famous way to deep-copy an object, and it is a lossy one, for every reason in the table above. `structuredClone(x)` is built in to Node and browsers for exactly this job. I copied the same object both ways:

```
const src = { d: new Date(0), u: undefined, n: NaN, m: new Map([[1, 2]]), nested: { x: 1
console.log(JSON.parse(JSON.stringify(src)));
console.log(structuredClone(src));
```

```
{ d: '1970-01-01T00:00:00.000Z', n: null, m: {}, nested: { x: 1 } }
{ d: 1970-01-01T00:00:00.000Z, u: undefined, n: NaN, m: Map(1) { 1 => 2 }, nested: { x:
```

| | JSON ROUND-TRIP | `STRUCTUREDCLONE` |
|---|---|---|
| Date | becomes a string | stays a Date |
| `undefined`, `NaN` | key dropped, `null` | kept |
| Map, Set | become `{}` | kept |
| circular references | throws | kept (I checked the copy's `self` pointed into the copy) |
| functions | dropped | **throws** `DOMException: f() {} could not be cloned.` |

| | JSON ROUND-TRIP | `STRUCTUREDCLONE` |
|---|---|---|
| class instances | own data only | own data only, as a plain object (the prototype and methods are not copied) |
| purpose | produces *text* you can send or save | produces a live copy; it is *not* text |

Use `structuredClone` to copy data inside a program and JSON only when you need text.

## Key Order

JSON objects are "unordered" in the standard, but JavaScript objects do have an order, and `stringify` follows it. The rule I observed: **keys that look like array indexes (non-negative integers) come first, in numeric order; all other keys follow in the order they were added.** I serialised `{ b: 1, a: 2, 10: 'x', 2: 'y', '-1': 'neg', '01': 'z' }` and got:

```
{"2":"y","10":"x","b":1,"a":2,"-1":"neg","01":"z"}
```

`"2"` and `"10"` were pulled to the front and sorted as numbers; `"-1"` and `"01"` (not canonical array indexes) stayed in insertion order. So do not rely on key order carrying meaning (put an ordered list in an *array*), and when you compare two JSON texts, parse them first rather than comparing strings.

## One Safety Note: the `__proto__` Key

`JSON.parse` treats `"__proto__"` as an ordinary key. Parsing `{"__proto__":{"admin":true},"a":1}` gave an object whose own keys were `__proto__,a`; the global `({}).admin` was still `undefined` and so was `p.admin`. The danger comes later, when code *merges* such an object into another with a naive loop, and *Working with JSON 11* demonstrates and fixes that.

MISTAKES WHILE PREPARING THIS CHAPTER

**1. I planned to teach the popular "seen objects" circular-reference replacer.** When I tested it on an object that is merely *shared*, it replaced the second use with `"[Circular]"` and lost data. **How to avoid it:** test a helper on the case that looks similar but is different (shared versus circular), not only on the case it was written for. The ancestor-path version in this chapter passed both, and a third deeper

test.

**2. The outline said to check whether `JSON.rawJSON` exists and label it if not.** I half-expected to write "not available yet". In fact Node 24.18 and Chrome 154 both have it and the reviver `context` argument. **How to avoid it:** run `typeof` on the feature in each runtime you name, rather than carrying over what you remember from a year ago; and say which runtimes you did not test, as above.

**3. The reviver exercise first matched `key === 'id'`.** That worked only because my sample had a single `id`. When I added an order line with `"id": "A-77"` and a customer `"id": "42"`, the same rule turned them into `77n` and `42n` (the letters of A-77 silently stripped). **How to avoid it:** when a key-name rule is the whole logic, build one sample where the same key appears at a second depth and see what happens.

## Hands-On Exercises

**EXERCISE 1**

Build an order object holding a BigInt id, a nested customer with a `password`, a Date, a Set of tags, an `undefined` field and a `NaN`. Show that a plain `stringify` throws, then write one replacer function that makes it serialise (BigInt as a string, Set as an array, password removed) and print the result with a 2-space indent. Parse it back with a reviver that restores the BigInt, the Date and the Set, and check each with a comparison. Finish with a control: parse the same text without the reviver and print the types you get.

View solution

**EXERCISE 2**

Start a local Node HTTP server with six routes: a 200 JSON body, a 404 with a JSON error body, a 502 with an HTML body, a 204 with no body, a 200 whose JSON is cut off, and a 200 with a `text/plain` type but valid JSON. Call each with `(await fetch(url)).json()` and with your own `getJson` helper, and print both results side by side. Then show that reading a body twice fails, and POST the same object three ways (no header, with header, body not stringified) to an echo route to show what the server received.

View solution

**EXERCISE 3**

Copy one object with a Date, `undefined`, `NaN` and a Map using a JSON round-trip and using `structuredClone`, and list what each kept. Then make a circular object and show the stringify error, write a circular-reference replacer, and test it on the circle *and* on a merely shared object (also test the naive "seen" version to see it fail). Finally show key order for a mixed object, and read a 20-digit id exactly with the reviver's `context.source` and write it with `JSON.rawJSON`.

View solution

## CHAPTER 2 QUICK REFERENCE

- `JSON.parse(text, reviver)` throws `SyntaxError` on bad text; `JSON.stringify(value, replacer, indent)` returns `undefined` for a top-level `undefined`, function or symbol
- Indent: a number (capped at 10) or a string (first 10 characters); empty arrays and objects print as `[]` and `{}`
- Silently lost: `undefined`, functions, symbols (as properties); `NaN` / `Infinity` become `null`; `-0` becomes `0`; Map, Set, RegExp, Error become `{}`; arrays turn `undefined` into `null`. A BigInt throws
- `toJSON(key)` decides what an object serialises as (Date already has one); it runs before the replacer
- Replacer array = whitelist at *every* depth; replacer function = called outside-in starting with key `""`, return `undefined` to drop a key; `this` is the holder
- Reviver is called inside-out, root last; returning `undefined` deletes; a key-name rule matches at every depth; third argument `context.source` gives the original number text (Node 24.18, Chrome 154)
- `JSON.rawJSON('123...')` writes exact digits; an object, or a number with a leading space, is rejected; check `typeof JSON.rawJSON` first
- Circular data throws `Converting circular structure to JSON`; a replacer must test *ancestors*, not every object seen, or shared objects are wrongly replaced
- `fetch` resolves on 404 and 502: check `res.ok`; `.json()` ignores `Content-Type`; a 204 gives "Unexpected end of JSON input"; "Unexpected token <" means HTML arrived; a body can be read once; `fetch` needs `Content-Type: application/json` and a stringified body
- Copy data with `structuredClone` (keeps Date, Map, Set, `undefined`, `NaN`, cycles; throws on functions); use JSON only to make text. Integer-like keys are written first, in numeric order

**COMING NEXT**

Chapter 3 is JSON in Python: `json.loads`, `json.dumps`, `load` and `dump`, the `indent`, `sort_keys`, `separators` and `ensure_ascii` arguments, how Python types map to JSON, `default=` and `object_hook` for dates, `Decimal` and dataclasses, the `NaN` and `Infinity` extension and `allow_nan`, and a comparison with this chapter. Start with *Working with JSON 3*.

# Working with JSON

Chapter 3 · Parsing and Generating in Python

Python's standard library has a `json` module with the same four jobs as JavaScript's two functions: text to values, values to text, and the same for files. Its defaults differ from JavaScript's in ways that matter: it adds spaces, it escapes non-ASCII characters, it accepts `NaN`, and it turns tuples and integer keys into things that do not come back the same. This chapter walks through the module, shows what survives a round trip and what does not (measured), then covers hooks for dates, `Decimal` and dataclasses, and the one place Python bites on Windows: reading and writing files without saying which encoding. It builds on *Working with JSON 1* (the format's rules) and *Working with JSON 2* (the JavaScript side to compare with); basic Python comes from *Python Fundamentals*.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

Python 3.14.0 on Windows 11, with the standard-library `json` module only. The Windows console and file defaults matter in this chapter (the machine's default text encoding is `cp1252`); on Linux and macOS the default is normally UTF-8, so the encoding problems below may not appear there. **Not run:** Linux, macOS, other Python versions, and the third-party libraries named in the last section (not installed, so no timing is claimed).

## The Four Functions

| FUNCTION | FROM | TO |
|---|---|---|
| `json.loads(s)` | a string (or bytes) | a Python value |
| `json.dumps(obj)` | a Python value | a string |
| `json.load(f)` | an open file | a Python value |
| `json.dump(obj, f)` | a Python value | written into an open file |

The "s" in `loads` and `dumps` stands for *string*, a common source of confusion with `load` and `dump`, which take files. I round-tripped a dictionary:

```
import json
d = {"name": "Ada", "age": 36, "tags": ["a", "b"], "ok": True, "x": None}
s = json.dumps(d)
print(s)
print(json.loads(s) == d)
```

```
{"name": "Ada", "age": 36, "tags": ["a", "b"], "ok": true, "x": null}
True
```

Notice what changed on the way: `True` became `true` and `None` became `null`, and there is a space after every `:` and `,`. JavaScript's `stringify` prints `{"a":1}` with no spaces. Both are valid JSON; the difference only matters for size and for tests that compare strings. The `loads` function also accepts *bytes*: I passed UTF-8 bytes and UTF-16 bytes and it detected each encoding and parsed both.

## Formatting Options for `dumps`

```
json.dumps(d, indent=2)                                  # pretty-printed, 2 spaces
json.dumps({"b": 1, "a": 2, "é": "ü"}, sort_keys=True)
json.dumps({"é": "ü😀"})                              # default: ensure_ascii=True
json.dumps({"é": "ü😀"}, ensure_ascii=False)
json.dumps(d, separators=(',', ':'))                     # compact, like JavaScript
```

```
{"a": 2, "b": 1, "\u00e9": "\u00fc"}
{"\u00e9": "\u00fc\ud83d\ude00"}
{"é": "ü😀"}
{"name":"Ada","age":36,"tags":["a","b"],"ok":true,"x":null}
```

- `indent`: a number of spaces or a string (I tried `'\t'` for tabs). With `indent=0` you still get newlines, one item per line, but no indentation.
- `sort_keys=True`: keys in alphabetical order, which makes output stable for diffs and tests. Unlike JavaScript, Python keeps insertion order by default and never reorders numeric-looking keys.
- `ensure_ascii`: `True` (the default) turns every non-ASCII character into `\uXXXX`, with a surrogate pair for 😀 (`\ud83d\ude00`). The output is pure ASCII, safe in any file or pipe, but

unreadable for non-English text and about 6 characters per accented letter. `False` keeps the characters, and then *you* must write the result in UTF-8 (see the file section).

- **`separators`**: a pair `(item_separator, key_separator)`. The default is `(', ', ': ')` normally and `(',', ': ')` when `indent` is set (so no trailing spaces at line ends). `(',', ':')` is the compact form.

Comparing the two languages' floats on the same numbers, I found Python and JavaScript mostly agree but not always in how they *write* them: `1e-7` is `1e-07` in Python and `1e-7` in JavaScript; `1.0` stays `1.0` in Python but is `1` in JavaScript; `1e16` is `1e+16` in Python and `10000000000000000` in JavaScript. All of these are valid JSON and parse to the same numbers.

## The Type Mapping, and What Does Not Round-Trip

| JSON | PYTHON (`loads`) | PYTHON (`dumps`) ACCEPTS |
|---|---|---|
| object | `dict` | `dict` (with str, int, float, bool or None keys) |
| array | `list` | `list` and `tuple` |
| string | `str` | `str` |
| number (integer form) | `int` (any size) | `int` |
| number (fraction or exponent) | `float` | `float` |
| `true` / `false` | `True` / `False` | `True` / `False` |
| `null` | `None` | `None` |

Everything not in the right-hand column raises `TypeError`. I ran each of these through `dumps` then `loads` and compared with the original:

```
value                dumps                                  loads(dumps) == orig
dict, str keys       '{"a": 1, "b": [true, null, 2.5]}'     True
tuple                '[1, 2]'                               False   -> [1, 2]  (
dict, int keys       '{"1": "a", "2": "b"}'                 False   -> {'1': 'a'
dict, float key      '{"2.5": "x"}'                         False
dict, bool/None key  '{"true": "x", "null": "y"}'           False
int 10**20           '100000000000000000000'                True
float -0.0           '-0.0'                                 True
nan                  'NaN'                                  both are NaN (nan !=
```

```
inf                  'Infinity'                                  True
date, Decimal, set, bytes                                        TypeError (no defaul
key collision        '{"1": "int", "1": "str"}'                  False   -> {'1': 'st
```

Three results deserve attention. **Tuples come back as lists** and **every non-string key comes back as a string**, so a dictionary keyed by integers is not the same dictionary after a round trip. Worse, **the key collision case produced invalid-looking output**: `{1: 'int', '1': 'str'}` became `{"1": "int", "1": "str"}`, with a duplicate key, and reading it back kept only the last. Python had two different keys, JSON has one. Nothing raised an error. The other failures are loud: `TypeError: Object of type set is not JSON serializable` (the same message for `datetime`, `bytes`, `complex` and plain objects), and `TypeError: keys must be str, int, float, bool or None, not tuple` for a tuple key. `skipkeys=True` silently drops such keys, which is rarely what you want.

A circular dictionary raised `ValueError: Circular reference detected`, the counterpart of the JavaScript error from Chapter 2. Python integers have no size limit, so `10**20` round-trips exactly, unlike in JavaScript. There is one limit, though, added to Python for safety: converting a very long digit string to an int is capped. I parsed a 4,300-digit number fine, and a 4,301-digit one failed with `ValueError: Exceeds the limit (4300 digits) for integer string conversion` (the limit is `sys.get_int_max_str_digits()`). Ordinary data never gets near it, but an attacker's input can, which is why the limit exists.

## Teaching `dumps` New Types: `default=`

A `default` function is called for every object the encoder does not know. It should return something the encoder *does* know, or raise `TypeError`. Here is one that handles the types most programs have, tested on a dictionary holding every one of them at once:

```
import json, dataclasses, datetime as dt, uuid, pathlib
from decimal import Decimal

@dataclasses.dataclass
class Line:
    sku: str
    price: Decimal
    added: dt.date

def default(o):
```

```
    if isinstance(o, (dt.datetime, dt.date)): return o.isoformat()
    if isinstance(o, Decimal): return str(o)                # a string keeps every dig
    if dataclasses.is_dataclass(o): return dataclasses.asdict(o)
    if isinstance(o, (set, frozenset)): return sorted(o)
    raise TypeError('Object of type ' + type(o).__name__ + ' is not JSON serializable')

order = {'id': 1001, 'placed': dt.datetime(2026, 10, 12, 8, 30, tzinfo=dt.timezone.utc),
         'lines': [Line('A1', Decimal('19.99'), dt.date(2026, 1, 2))], 'tags': {'rush',
print(json.dumps(order, default=default, indent=2))
```

```
{
  "id": 1001,
  "placed": "2026-10-12T08:30:00+00:00",
  "lines": [
    {
      "sku": "A1",
      "price": "19.99",
      "added": "2026-01-02"
    }
  ],
    "gift",
    "rush"
  ]
}
```

Details worth knowing. `dataclasses.asdict` converts nested dataclasses for you, and then `default` is called again for the `Decimal` and `date` inside. A `set` has no order, so sort it to make the output repeatable. `pathlib` paths, `uuid.UUID` and `enum.Enum` values work the same way: I tested `PurePosixPath('/tmp/x')` (via `.as_posix()`), a UUID (via `str`) and an Enum (via `.value`) with a similar function and each produced a plain string. If you prefer a class, subclass `json.JSONEncoder`, override its `default` method, and pass `cls=MyEncoder`; I confirmed that gave the same output.

**WHY A `DECIMAL` GOES OUT AS A STRING**

A `float` cannot hold `0.1` exactly: `0.1 + 0.2` printed `0.30000000000000004`, while `Decimal('0.1') + Decimal('0.2')` printed `0.3`. Converting the `Decimal` to `float` before writing brings the error back, and I found no simple built-in way in the standard module to write a `Decimal` as a bare JSON number. A string keeps every digit. The price of that choice: the reader must know that field is a decimal string. Chapter 6 (JSON Schema) is where such a contract is written down.

## Teaching `loads` New Types: Hooks

Python has four hooks on `loads`. `object_hook` is called with each decoded dictionary, innermost first (like the JavaScript reviver). I logged the order for `{"a":{"b":{}}, "c":[{"d":1}]}`: the hook saw `[]` (the innermost empty object), then `['b']`, then `['d']`, then `['a', 'c']`, the outermost, last.

```
def hook(o):
    if 'placed' in o: o['placed'] = dt.datetime.fromisoformat(o['placed'])
    if 'added'  in o: o['added']  = dt.date.fromisoformat(o['added'])
    if 'price'  in o: o['price']  = Decimal(o['price'])
    return o
back = json.loads(text, object_hook=hook)
```

```
placed   datetime.datetime(2026, 10, 12, 8, 30, tzinfo=datetime.timezone.utc)   == origi
line 0   {'sku': 'A1', 'price': Decimal('19.99'), 'added': datetime.date(2026, 1, 2)}
tags is now a list (a set cannot come back by itself)
```

The hook rebuilt the datetime, the date and the Decimal, and the comparison with the original was `True`. Like the JavaScript reviver, it matches by *key name*: a string such as `"2026-10-12 is a date"` under another key was left alone in my test, but any key called `price` at any depth is converted. The `tags` set came back as a list, because JSON has no way to say "this array is a set"; convert it yourself if you need one.

The other three hooks, all measured:

- **`parse_float=Decimal`**: every number with a fraction or exponent is created with that function. `sum(json.loads('[0.1,0.2]', parse_float=Decimal))` gave `0.3`, and without it `0.30000000000000004`. This is the way to read money from JSON as numbers without any rounding error. (`parse_int` works the same way for integers; `parse_int=str` turned `1` and `3` into strings and left `2.0` a float.)
- **`parse_constant`**: called for `NaN`, `Infinity` and `-Infinity`. Raising inside it rejects them (next section).
- **`object_pairs_hook`**: called with the list of *(key, value) pairs* before they become a dictionary, so it can see duplicates. `json.loads('{"a":1,"a":2}')` normally returned `{'a': 2}`, but a small hook that raises on a repeated key made it fail with `duplicate key: 'a'`, and the same

hook accepted clean input. Passing `object_pairs_hook=list` returned the pairs themselves in file order.

## `NaN` and `Infinity`: Python's Extension

As Chapter 1 showed, Python accepts `NaN`, `Infinity` and `-Infinity` by default, and writes them too: `json.dumps([float('nan'), float('inf'), -float('inf')])` gave `[NaN, Infinity, -Infinity]`, which no standard-conforming parser (including JavaScript's) will read. To keep your output portable, switch it off in both directions:

```
try: json.dumps([float('nan')], allow_nan=False)
except ValueError as e: print('dumps:', e)

def refuse(name): raise ValueError('refusing ' + name)
try: json.loads('[NaN]', parse_constant=refuse)
except ValueError as e: print('loads:', e)
```

```
dumps: Out of range float values are not JSON compliant: nan
loads: refusing NaN
```

`allow_nan=False` makes `dumps` raise instead of writing invalid JSON; for `loads`, the way to refuse is `parse_constant`. Note that `1e400`, which is a legal JSON number, still parses to `inf`, so a value can become infinite without the word appearing. If you must handle NaN in your data, decide on a replacement (usually `null`) and convert before writing.

## Reading Errors: `JSONDecodeError`

Bad input raises `json.JSONDecodeError`, a subclass of `ValueError`, so catching `ValueError` catches it too. It carries the details as attributes. For `'{"a": 1,\n "b": }'` I read:

```
e.msg = Expecting value | e.lineno = 2 | e.colno = 7 | e.pos = 15 | e.doc[e.pos] = '}'
```

`pos` is a character offset into `doc` (the original text), counted from 0; `lineno` and `colno` count from 1. They are what Chapter 1's exercise used to put a caret under the error. Keep in mind that the 4,301-digit failure above is a plain `ValueError`, *not* a `JSONDecodeError`.

## Files, Encodings and Line Endings (Where Windows Bites)

`json.load` and `json.dump` work on files you open yourself, which means the encoding is chosen by `open()`, not by the `json` module. On this machine `locale.getpreferredencoding(False)` is `cp1252`, and the results are instructive. I wrote a file containing `Zoë` and 😜 in UTF-8, then read it back several ways:

| WHAT I DID | WHAT HAPPENED |
|---|---|
| `open(p, encoding='utf-8')` to write and to read | Round trip equal: `True` |
| `open(p)` to *read* that UTF-8 file (no encoding) | **No error, wrong text**: `'Zo\xc3\xab'` and `'\xf0Ÿ˜€'` (mojibake) |
| `open(p, 'w')` to write with `ensure_ascii=False` (no encoding) | `UnicodeEncodeError` (cp1252 has no 😜) |
| `open(p, 'w')` to write with the default `ensure_ascii=True` | Works: pure ASCII, `{"name": "Zo\u00eb", "emoji": "\ud83d\ude00"}` |
| read a file that starts with a UTF-8 BOM, `encoding='utf-8'` | `JSONDecodeError: Unexpected UTF-8 BOM (decode using utf-8-sig)` |
| same file, `encoding='utf-8-sig'` | `{'a': 1}` (the BOM is stripped) |
| write with `indent` in text mode on Windows | Line endings are `\r\n` (CRLF); `newline='\n'` in `open()` gave `\n` |

The dangerous row is the second: **reading UTF-8 JSON with the wrong encoding does not fail, it quietly produces garbage.** The cure is to always write `encoding='utf-8'` in `open()`, which RFC 8259 also requires for JSON exchanged between systems. The default `ensure_ascii=True` is the reason many programs have never seen the problem: ASCII-only output reads the same in every encoding. The same file written with CRLF line endings is still valid JSON (newlines are whitespace), but it produces noisy diffs in a Git repository that expects LF.

## The Command Line: `python -m json.tool`

The module can be run as a tool to check and format a file (as in Chapter 1). Python 3.14's options, from its `--help`, and my run of each on `{"b":1,"a":{"z":"é"}}`:

```
py -m json.tool t.json                          4-space indent (the default), non-ASCII e
py -m json.tool --sort-keys t.json              keys in alphabetical order
py -m json.tool --indent 1 t.json               one-space indent
py -m json.tool --tab t.json                    tabs
py -m json.tool --compact t.json                {"b":1,"a":{"z":"\u00e9"}}
py -m json.tool --no-ensure-ascii --compact t.json    {"b":1,"a":{"z":"é"}}
```

The help also lists `--no-indent` and `--json-lines` (parse JSON Lines input); I did not run those two, and I did not test older Pythons, which may offer fewer options.

## Side by Side with JavaScript

| | JAVASCRIPT (CHAPTER 2) | PYTHON |
|---|---|---|
| **Default output** | Compact: `{"a":[1,2]}` | Spaces: `{"a": [1, 2]}` |
| **Non-ASCII characters** | Written as they are | Escaped as `\uXXXX` unless `ensure_ascii=False` |
| **NaN / Infinity** | Become `null` when writing; rejected when reading | Written and read as `NaN` / `Infinity` unless `allow_nan=False` / `parse_constant` |
| **Unknown type** | Dropped, or `{}`, silently | `TypeError` unless `default=` handles it |
| **Big integers** | Lose digits past $2^{53}$ | Exact (to the 4,300-digit parse limit) |
| **Hook on reading** | Reviver (key, value; inside-out) | `object_hook` (whole dict; inside-out) |
| **Hook on writing** | `toJSON` and replacer | `default=` or a `JSONEncoder` subclass |
| **Key order** | Integer-like keys first, sorted | Insertion order (`sort_keys=True` to sort) |
| **Circular data** | `TypeError: Converting circular structure` | `ValueError: Circular reference detected` |

## Faster Third-Party Libraries (Names Only)

The standard module is fast enough for most work. Libraries that advertise being faster exist; three that are widely used are `orjson`, `ujson` and `msgspec`. I looked up their latest versions

on PyPI today (orjson 3.13.0, ujson 6.0.0, msgspec 0.22.0) but **did not install, test or time any of them**, so this chapter makes no speed claim and no claim about how they differ from `json` on the edge cases above. Measure on your own data before switching, and run this chapter's round-trip table against any replacement, since libraries make different choices about big integers, `NaN` and key types.

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My first test script crashed printing an emoji.** The Windows console encoding (`cp1252`) could not show 😀, and Python raised `UnicodeEncodeError` from `print`, not from `json`. **How to avoid it:** print with `ascii(value)` when you want to see exactly what a string holds, or run with `py -X utf8`.

**2. Running with `-X utf8` then hid the encoding problem.** With UTF-8 mode on, `open()` with no encoding read my file correctly, so my first "no encoding" test showed no problem at all. **How to avoid it:** reproduce an environment-dependent bug in the environment's default settings, and print `locale.getpreferredencoding(False)` so the reader knows what was in force.

**3. My first "money arithmetic" example used `19.99 + 0.10 + 0.20` and showed no error.** That sum happens to print as `20.29` in floating point, so the example demonstrated nothing. **How to avoid it:** find an input that actually fails before using it to teach: `0.1 + 0.2` does.

## Hands-On Exercises

**EXERCISE 1**

Write a script that takes about sixteen Python values (a dict with string keys, a tuple, dicts with int, float and bool/None keys, `10**20`, `-0.0`, NaN, infinity, a string with an accented letter and an emoji, a date, a Decimal, a set, bytes, and a dict whose keys `1` and `'1'` collide). For each, run `json.dumps`, then `json.loads`, and print the JSON text, whether the result equals the original, and what came back, with a clear marker for values that cannot be serialised. Handle NaN's "never equal to itself" correctly.

View solution

**EXERCISE 2**

Take an order containing a datetime, a list of dataclass lines each with a Decimal and a date, and a set of tags. Show that plain `dumps` fails, write a `default` function and print the indented JSON, then write

an `object_hook` that restores the datetime, dates and Decimals and check one value with `==`. Then show `0.1 + 0.2` read with and without `parse_float=Decimal`, and write an `object_pairs_hook` that rejects duplicate keys, testing it on both a duplicate and a clean document.

View solution

**EXERCISE 3**

In a temporary folder, print the default text encoding, then write a JSON file containing "Zoë" and an emoji with `encoding='utf-8'` and read it back. Show what happens when you read it with no encoding, and when you write with no encoding both with and without `ensure_ascii=False`. Then test a file with a BOM using `utf-8` and `utf-8-sig`, compare line endings with and without `newline='\n'`, show the `allow_nan=False` and `parse_constant` refusals, and run `json.tool` with six option combinations. Do not use `-X utf8` for the first part.

View solution

## CHAPTER 3 QUICK REFERENCE

- `loads` / `dumps` work on strings (`loads` also takes bytes), `load` / `dump` on open files; the encoding is chosen by `open()`, so always write `encoding='utf-8'`
- `dumps` options: `indent`, `sort_keys`, `separators=(',', ':')` for compact, `ensure_ascii=False` to keep non-ASCII; default output has spaces and escapes non-ASCII as `\uXXXX`
- Mapping: dict/list/str/int/float/True/False/None. `tuple` becomes a list; non-string keys become strings and can collide silently; date, Decimal, set, bytes, complex and plain objects raise `TypeError`
- `default=` (or a `JSONEncoder` subclass) handles new types on the way out: `isoformat()` for dates, `str()` for Decimal, `dataclasses.asdict`, `sorted()` for sets
- On the way in: `object_hook` (each dict, innermost first), `parse_float=Decimal`, `parse_int`, `parse_constant`, `object_pairs_hook` (sees duplicate keys)
- NaN and Infinity are accepted and written by default; `allow_nan=False` makes `dumps` raise, `parse_constant` can make `loads` raise; `1e400` still becomes `inf`
- `JSONDecodeError` (a `ValueError`) has `msg`, `doc`, `pos`, `lineno`, `colno`; a 4,301-digit integer raises a plain `ValueError` (limit 4,300)
- Windows pitfalls: `open()` with no encoding read UTF-8 as cp1252 without an error; a BOM needs `utf-8-sig`; text mode writes CRLF unless `newline='\n'`

- `py -m json.tool`: `--sort-keys`, `--indent N`, `--tab`, `--compact`, `--no-ensure-ascii` (also `--no-indent`, `--json-lines`, not run)

**COMING NEXT**

Chapter 4 is the `jq` command line: installing it (it is not on this machine yet, so the chapter starts with a checked install on Windows), the program-on-stdin model, `.key`, `.[]`, pipes, `select` and `map`, the `-r`, `-c`, `-s` and `--arg` options, `curl | jq`, and quoting in bash, PowerShell and cmd. Start with *Working with JSON 4*.

# Working with JSON

Chapter 4 · The jq Command Line

Chapters 2 and 3 handled JSON inside programs. Often you only want to *look* at some JSON: pick one field from an API response, count the items in a file, list the names that match a condition. Writing a script for each question is slow. `jq` is a small command-line program that reads JSON, applies a short *filter* you write, and prints the result. This chapter installs it, teaches the filters you will use every day (selecting, filtering, the main options), connects it to `curl`, and covers the part that trips people up on Windows: quoting a filter correctly in bash, PowerShell and cmd. The next chapter covers reshaping data.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

jq 1.8.2 (the latest release when this was written, published 2026-06-20 on jq's GitHub releases page) on Windows 11, with Git Bash, Windows PowerShell 5.1 and cmd.exe, and curl 8.12.1. The sample data and local test server are made up for this chapter. **Not run:** Linux, macOS, PowerShell 7, and any install method other than the Windows release download (`winget`, Homebrew, `apt` and so on exist, but I did not try them). Results that depend on Windows are labelled. The jq manual at jqlang.org is the reference for everything not shown here.

## Installing jq on Windows (Checked)

jq was **not installed** on this machine, so I installed it into a scratch folder instead of changing the system: download the program from the project's GitHub releases page, *verify the download*, and put it on the `PATH` for the session. The release page lists a single file for 64-bit Windows, `jq-windows-amd64.exe` (1,035,264 bytes), and a text file of checksums (`sha256sum.txt`).

```
curl -sL -o jq.exe https://github.com/jqlang/jq/releases/download/jq-1.8.2/jq-windows-am
curl -sL -o sha256sum.txt https://github.com/jqlang/jq/releases/download/jq-1.8.2/sha256
certutil -hashfile jq.exe SHA256
jq.exe --version
```

```
expected sha256: a6fc67fedaf9128a3309a1e2ebb8b986aeccf70122ee46d2cb4849e423f0c627   (fro
actual sha256  : a6fc67fedaf9128a3309a1e2ebb8b986aeccf70122ee46d2cb4849e423f0c627   (fro
checksum: MATCH
```

```
control: a copy with one extra byte added does not match (good)
jq-1.8.2
```

The two hashes matched, and a deliberately damaged copy did not, which shows the comparison can fail. Always check the hash of a program you download. To make `jq` available everywhere you would put the `.exe` in a folder on your `PATH` (and rename it `jq.exe`, as above); I did not do that. From here on, `jq` means this binary. Run `jq --version` to see which version your shell is actually using.

## The Model: Input In, Filter, Output Out

`jq` reads JSON, runs your *filter* on each JSON value it reads, and prints every result it produces as pretty-printed JSON. A filter can produce **zero, one or many** results for a single input, and that idea explains most of jq. All the examples use this sample file, `shop.json`:

```
{
  "shop": "Corner Shop",
  "open": true,
  "owner": {"first": "Ada", "last": "Lovelace", "phone": null},
  "products": [
    {"sku": "A1", "name": "Tea",      "price": 3.5,  "stock": 40,  "tags": ["drink", "ho
    {"sku": "B2", "name": "Coffee",   "price": 4.25, "stock": 0,   "tags": ["drink", "ho
    {"sku": "C3", "name": "Biscuits", "price": 1.99, "stock": 120, "tags": ["food"]},
    {"sku": "D4", "name": "Milk",     "price": 0.89, "stock": 15}
  ]
}
```

The simplest filter is `.`, "the input itself", which makes `jq . file.json` a formatter and validator (it prints the file indented, or reports an error). A filter is the first argument and the file name comes after; with no file name, jq reads standard input. The rest of the chapter is about the filters.

## Selecting: Keys, Indexes and `[]`

| FILTER | RESULT ON `shop.json` |
|---|---|
| `.shop` | `"Corner Shop"` |
| `.owner.first` | `"Ada"` |
| `.owner.phone` | `null` (the key exists, its value is null) |
| `.owner.missing` | `null` (no such key; same result, no error) |
| `."shop"` | `"Corner Shop"` (quotes are needed for keys with spaces or dashes: `."my-key"`) |
| `.products[0].name` | `"Tea"` |
| `.products[-1].name` | `"Milk"` (negative counts from the end) |
| `.products \| length` | `4` |
| `.products[1:3] \| map(.sku)` | `["B2","C3"]` (a slice: items 1 and 2, end not included) |
| `.products[].name` | four results: `"Tea"`, `"Coffee"`, `"Biscuits"`, `"Milk"` |
| `.owner \| keys` | `["first","last","phone"]` |

`[]` is the "one result per item" filter: `.products[]` turned one array into four separate objects, and `.products[].name` then took `name` from each. This is the "zero, one or many" idea at work. Notice that a missing key gives `null`, not an error, which is convenient and also a trap: a typo in a key name looks like missing data. The fourth product, Milk, has no `tags` at all, and that shows up next:

```
jq '.products[].tags[]'  shop.json
jq '.products[].tags[]?' shop.json
```

```
"drink"
"hot"
"drink"
"hot"
"food"
jq: error (at shop.json:11): Cannot iterate over null (null)      (exit status 5; this i

"drink"                                                           (second command, with
"hot"
```

```
  "drink"
  "hot"
  "food"
```

For Milk, `.tags` is `null` and `[]` cannot step through `null`, so jq printed the five tags from the first three products, then stopped with an error and exit status 5. The `?` suffix means "if this fails, quietly produce nothing", and it gave the same five tags with no error. Use `?` deliberately: it also hides real mistakes.

## Pipes, Commas and Building Output

`|` feeds the results of the left side into the right side, one at a time, exactly like a shell pipe. `,` gives several results from the same input. And `|` binds **loosest**: `.shop | type, (.open|type)` does *not* mean "the type of `.shop`, then the type of `.open`". I ran it:

```
$ jq '.shop | type, (.open|type)' shop.json
jq: error (at shop.json:11): Cannot index string with string ("open")
"string"
```

The comma sits *inside* the right side of the pipe, so `(.open|type)` ran on the string `"Corner Shop"` and failed. The fix is to write what you mean: `(.shop | type), (.open | type)`, or `.shop, .open | type`. When in doubt, add parentheses.

You can also build new values. An **object** from parts: `.products[0] | {name, cost: .price}` gave `{"name":"Tea","cost":3.5}` (`{name}` is shorthand for `{name: .name}`). An **array** from many results: `[.products[] | .name | length]` gave `[3,6,8,4]`. These are the building blocks for Chapter 5.

## Filtering and the Everyday Functions

`select(condition)` passes its input through only if the condition is true; `map(f)` is `[.[] | f]`. These ran on `shop.json`:

| FILTER | RESULT |
|---|---|
| `.products[] \| select(.stock > 0) \| .name` | Tea, Biscuits, Milk |

| FILTER | RESULT |
| --- | --- |
| `.products[] \| select(.price < 2) \| {name, price}` | `{"name":"Biscuits","price":1.99}` and `{"name":"Milk","price":0.89}` |
| `.products \| map(select(.stock == 0)) \| length` | `1` |
| `.products \| map(.price) \| add` | `10.63` |
| `.products \| map(.price) \| min, max` | `0.89` then `4.25` |
| `[.products[].price] \| add / length` | `2.6575` (the average) |
| `.products \| min_by(.price) \| .name` | `"Milk"` |
| `.products \| sort_by(.price) \| map(.name)` | `["Milk","Biscuits","Tea","Coffee"]` |
| `.products \| map(.name) \| join(", ")` | `"Tea, Coffee, Biscuits, Milk"` |
| `.products[] \| select(.tags \| index("hot")) \| .name` | Tea, Coffee (`index` gives a position, or `null`; `null` is false) |
| `.products[] \| select(.name \| test("^C"; "i")) \| .sku` | `"B2"` (`test` is a regular-expression match; `"i"` ignores case) |
| `.owner \| has("first"), has("middle")` | `true`, `false` (`has` tests for a key, or an array index) |
| `"first" \| in({"first": 1})` | `true` (`in` is `has` the other way round) |
| `.products \| map(.stock) \| any(. == 0), all(. > 0)` | `true`, `false` |
| `.products[0].price * 2` | `7` |
| `.shop \| type`; `.products \| type`; `.open \| type` | `"string"`; `"array"`; `"boolean"` |
| `.products[] \| select(.tags != null and (.tags \| contains(["food"]))) \| .name` | `"Biscuits"` (`contains` on an array tests that every listed item is present) |

Two lessons from this table. First, **guard against missing keys before calling a function on them**: the `tags != null and ...` in the last row is needed because `contains` on `null` would fail for Milk. Second, `select` produces *nothing* (not `false`) when the condition fails, which is why a filter that matches nothing prints nothing at all.

## Comparison Traps: Strings Are Not Numbers

jq has a total ordering across types: `null` < `false` < `true` < numbers < strings < arrays < objects. I confirmed it by sorting `[1, "1", null, true, false, [], {}]` and got `[null,false,true,1,"1",[],{}]`. The consequence for command-line use: `--arg` always creates a *string*, so this filter compares a number with a string:

```
jq -c --arg min 2 '[.products[] | select(.price > $min) | .name]' shop.json
jq -c --arg min 2 '[.products[] | select(.price > ($min | tonumber)) | .name]' shop.json
jq -c --argjson min 2 '[.products[] | select(.price > $min) | .name]' shop.json
```

```
[]
["Tea","Coffee"]
["Tea","Coffee"]
```

The first command printed an empty list with exit status 0 and no warning: every number sorts *before* every string, so `3.5 > "2"` is false (`jq -n '3.5 > "2"'` printed `false`). Convert with `tonumber`, or pass the value with `--argjson`, which parses its argument as JSON. The test `jq -n --arg n 5 --argjson m 5 '[$n, $m, ($n == "5"), ($m == 5)]'` gave `["5",5,true,true]`.

## The Options You Will Use

| OPTION | EFFECT | WHAT I SAW |
|---|---|---|
| `-r` | Raw output: print strings without quotes | `.shop` gave `Corner Shop` instead of `"Corner Shop"` |
| `-c` | Compact: one line per result | `.products[0]` became one line |
| `-e` | Set the exit status from the last result | `true` gave exit 0; `null` gave exit 1; no output at all gave exit 4 |
| `-n` | No input: run the filter on `null` | `jq -n '1 + 2'` printed `3`; `jq -n '[range(5)]' -c` printed `[0,1,2,3,4]` |
| `-s` | Slurp: read *all* inputs into one array | Three lines `{"a":1}` `{"a":2}` `{"a":3}` became `[{"a":1},{"a":2},{"a":3}]`; `-s 'map(.a) \| add'` gave `6` |

| OPTION | EFFECT | WHAT I SAW |
|---|---|---|
| `--arg name value` | Pass a *string* as `$name` | See the comparison trap above |
| `--argjson name json` | Pass a parsed JSON value as `$name` | `2` stayed a number |
| `--args` | Remaining arguments become `$ARGS.positional` | `jq -n --args '$ARGS.positional' a b c -c` gave `["a","b","c"]` |
| `-S`, `--tab`, `--indent n` | Sort keys; indent with tabs; indent with n spaces (up to 7) | Listed in `jq --help` |
| `-a` | ASCII output: escape non-ASCII characters | A string with é and an emoji printed as `"caf\u00e9 \ud83d\ude00"` with `-a`, and as the characters themselves without it |
| `-f file` | Read the filter from a file | Used in the quoting section below |
| `-b` | Binary: do not convert line endings (Windows) | See "Line endings on Windows" below |

`-r` is the one you will use most, because it makes jq's output usable by other commands: `jq -r '.products[].name'` prints one name per line, ready for a loop. `-e` is for scripts: it lets an `if` test whether a filter found something. Note the three different exit statuses of `-e` (a "null or false" result is 1, "no output" is 4), and that without `-e` a filter that matches nothing exits 0.

## What jq Does to Your Numbers and Input

I fed jq the awkward inputs from Chapter 1:

| INPUT | `JQ -C .` PRINTED | COMMENT |
|---|---|---|
| `{"a":12345678901234567890}` | `{"a":12345678901234567890}` | The original digits are kept when you only pass the value through |
| `[12345678901234567890]` with `.[0] + 1` | `12345678901234567000` | As soon as you do arithmetic, it is an ordinary 64-bit float |
| `1.0` and `1e2` | `1.0` and `1E+2` | Written back the way jq 1.8.2 preserved them, not as `1` and `100` |

| INPUT | `JQ -C .` PRINTED | COMMENT |
|---|---|---|
| `{"a":NaN}` | `{"a":null}` | jq accepts it as input and prints `null`; Chapter 1's parsers either failed or kept NaN |
| `{"a":1,"a":2}` | `{"a":2}` | Last duplicate wins |
| a UTF-8 BOM before the document | `{"a":1}` | Accepted: *unlike* Node and Python, which both rejected it |
| `{"a":1} {"a":2}` (two values) | `1` and `2` from `.a` | jq reads every value in the input, one after another; this is how it handles JSON Lines |

The point for this course: jq's handling of numbers and of invalid input is *its own*, a third behaviour beside JavaScript's and Python's. When data must pass through all three, do not assume they agree. Chapter 1's rule stands: test the actual inputs.

## `curl` into `jq`

The most common use of jq is looking inside an API response. `curl` downloads, jq reads from the pipe (the `curl` side, including its options, is covered in *API Testing & Tooling* 6 and 7). I started a tiny server on `127.0.0.1` that returns the sample file, a 404 with a JSON error body, and a 502 HTML page, and tried each way of calling it:

| COMMAND | WHAT HAPPENED |
|---|---|
| `curl -s URL/shop \| jq -r '.products[].name'` | Tea, Coffee, Biscuits, Milk, one per line |
| `curl URL/hello 2>&1 \| jq -c .` (no `-s`, stderr merged) | `jq: parse error: Invalid numeric literal at line 1, column 4` — curl's progress meter text went into jq |
| `curl -s URL/missing \| jq -c .` (404, JSON body) | `{"error":"not found"}`, exits curl 0, jq 0 — **nothing says it failed** |
| `curl -sf URL/missing \| jq -c .` | No output; curl exit 22, jq exit 0 (jq got empty input) |
| `curl -s --fail-with-body URL/missing \| jq -c .` | `{"error":"not found"}` printed *and* curl exit 22 |

| COMMAND | WHAT HAPPENED |
| --- | --- |
| `curl -s URL/html \| jq .` (502, HTML) | `jq: parse error: Invalid numeric literal at line 1, column 10`, jq exit 5 |
| `set -o pipefail; curl -sf URL/missing \| jq -c .` | The pipeline's exit status was 22 (curl's), not jq's |
| `code=$(curl -s -o body.tmp -w '%{http_code}' URL/missing)`, then test `$code` | Status `404`; the body was kept in a file to read or ignore |

Three lessons. **A pipe's exit status is the last command's** unless you use `pipefail`, so `jq` reporting success says nothing about whether the request worked. **A server's JSON error body is still valid JSON**, so jq prints it as if it were your data; check the status with `-f`, `--fail-with-body` or the `%{http_code}` variable. And **"Invalid numeric literal" almost always means the input was not JSON at all**: an HTML page or progress text. Look at the raw body first. This is the same mistake as the `Unexpected token <` error in Chapter 2.

## Reading jq's Error Messages

| CAUSE | MESSAGE | EXIT |
| --- | --- | --- |
| Input cut off: `{"a":1` | `jq: parse error: Unfinished JSON term at EOF at line 2, column 0` | 5 |
| Input is not JSON: `not json` | `jq: parse error: Invalid numeric literal at line 1, column 4` | 5 |
| Single quotes in the data | `jq: parse error: Invalid string literal; expected ", but got ' at line 1, column 5` | 5 |
| Trailing comma: `[1,2,]` | `jq: parse error: Expected another array element at line 1, column 6` | 5 |
| Wrong type: `.a.b` on `{"a":1}` | `jq: error (at <stdin>:1): Cannot index number with string ("b")` | 5 |
| `.[0]` on an object | `jq: error (at <stdin>:1): Cannot index object with number (0)` | 5 |
| `keys` on a number | `jq: error (at <stdin>:1): number (1) has no keys` | 5 |

| CAUSE | MESSAGE | EXIT |
|---|---|---|
| Filter ends too soon: `.a \|` | `jq: error: syntax error, unexpected end of file at <top-level>, line 1, column 4:` followed by the filter with a caret, then `jq: 1 compile error` | 3 |
| File does not exist | `jq: error: Could not open file nosuchfile.json: No such file or directory` | 2 |

The messages come in three families, and the first words tell you which. `parse error` means the *data* is not valid JSON: look at the input. `error (at file:line)` means the data was fine but the filter did something impossible to it (indexing a number): check the shape of the data, with `jq 'type'` or `jq keys`. `compile error` means the *filter* itself is malformed: check quoting and parentheses. The exit statuses I saw were 2 (cannot open the file), 3 (compile error) and 5 (parse or run-time error), and, as above, 1 and 4 from `-e`.

## Quoting a Filter on Windows

A jq filter is full of characters the shell wants for itself: `|`, `"`, `$`, parentheses. The right quoting differs per shell. I ran the same filter, `.products[] | select(.name == "Tea") | .sku` (expected output `"A1"`), in each:

| SHELL | HOW I WROTE IT | RESULT |
|---|---|---|
| Git Bash | single quotes around the filter, double quotes inside | Works |
| PowerShell 5.1 | `'.products[] \| select(.name == "Tea") \| .sku'` | **Fails**: `jq: error: Tea/0 is not defined` — the inner double quotes were removed before jq saw the filter (exit 3) |
| PowerShell 5.1 | inner quotes escaped with a backslash: `'... == \"Tea\" ...'` | Works (`"A1"`) |
| PowerShell 5.1 | `--arg n Tea '... == $n ...'` | Works: no inner quotes needed |
| PowerShell 5.1 | filter in a file: `jq -f f.jq shop.json` | Works |

| SHELL | HOW I WROTE IT | RESULT |
|---|---|---|
| PowerShell 5.1 | `jq --% ".products[] \| select(.name == \"Tea\") \| .sku" shop.json` | Works, but everything after `--%` on the line is passed to jq, so nothing else can follow it |
| cmd.exe | double quotes around the filter: `jq ".shop" shop.json` | Works |
| cmd.exe | single quotes: `jq '.shop' shop.json` | **Fails**: `unexpected INVALID_CHARACTER` — cmd does not treat `'` as quoting |
| cmd.exe | inner quotes as `\"Tea\"` inside double quotes | Works |
| cmd.exe | `--arg n Tea`, and `-f f.jq` | Both work |

Two habits make quoting nearly painless on any shell. **Keep strings out of the filter** with `--arg` and `--argjson`, so the filter has no double quotes. And **put long filters in a file** with `-f`: the shell never sees them. I tested Windows PowerShell 5.1 only; PowerShell 7 may pass arguments to programs differently, so check the backslash row there before trusting it. Piping into jq from PowerShell also worked: `@{ a = 1; b = 'x' } | ConvertTo-Json -Compress | jq -c '.'` printed `{"a":1,"b":"x"}`.

## Line Endings on Windows

I looked at the bytes jq wrote for `jq -r .shop shop.json` redirected to a file: they ended in `0d 0a` (a carriage return and a line feed, CRLF), and with `-b` in `0a` alone. So on Windows, jq writes Windows line endings to files and pipes unless you add `-b`. That matters when the output goes to a Linux program or into a repository that expects LF. In Git Bash, command substitution (`x=$(jq -r .shop shop.json)`) still gave a string of 11 characters, without a stray carriage return, both with and without `-b`, so loops over `$(jq -r ...)` were fine in my test; other shells and tools may not strip it.

> **MISTAKES WHILE PREPARING THIS CHAPTER**
>
> **1. I wrote `.shop | type, (.open|type)` expecting two answers.** The pipe binds loosest, so the second `type` ran on the string and jq reported `Cannot index string with string ("open")`. **How to avoid it:** when a pipe and a comma appear together, add parentheses around each side.

**2. My first `--arg min 2` filter printed nothing and exited 0.** It compared numbers with the string `"2"`, and every number sorts before every string. **How to avoid it:** use `--argjson` for numbers, or `tonumber`, and treat an unexpectedly empty result as a type problem before a data problem.

**3. My first PowerShell test put `--%` in the middle of a line.** It passed the rest of the line (my `; "exit " + $LASTEXITCODE` text) to jq as file names. **How to avoid it:** `--%` must be the last thing on its line.

**4. Two of my test runs hung or misbehaved when launched from another shell.** A PowerShell script started from Git Bash waited forever until I closed its input (`< /dev/null`), and `cmd /c script.cmd` from Git Bash started an interactive cmd session instead of running the script, because Git Bash rewrote `/c` as a path. **How to avoid it:** run each shell's scripts from that shell, and put a timeout on anything automatic.

## Hands-On Exercises

**EXERCISE 1**

Install jq into a temporary folder without touching your `PATH` permanently: download the Windows binary and the checksum file from the official releases page, compare the SHA-256 hash with `certutil`, and print `jq --version`. Show a control: a copy with one byte added must *not* match. Then run ten filters on the sample `shop.json` (a key, a nested key, a null value, an array length, one array item, every name, only in-stock names with `-r`, the sum of prices, a slice, and the names of products that have a `tags` field).

View solution

**EXERCISE 2**

Answer ten questions about `shop.json` with one filter each (in-stock names, how many are out of stock, total stock, average price, cheapest product, the owner's full name, names joined with commas, the hot drinks, a CSV of `sku,stock`, the products with no `tags`), checking each answer against a value you worked out by hand, and print PASS or FAIL for each. Add one deliberately wrong expectation to prove the checker can fail. Then show the exit status of a filter that matches nothing, with and without `-e`, and of `-e` on a `null`.

View solution

**EXERCISE 3**

Start a small local server (200 JSON, 404 with a JSON error body, 502 HTML) and call it with `curl` piped into `jq` in eight ways: plain, `2>&1` without `-s`, the 404 with `-s`, `-sf` and `--fail-with-body`, the HTML page, `pipefail`, and a status check with `%{http_code}`. Print each exit status. Then run one filter with an inner double-quoted string in Git Bash, Windows PowerShell and cmd, show which quoting works and which does not, and show that `--arg` and `-f` work in all three.

View solution

## CHAPTER 4 QUICK REFERENCE

- jq reads JSON, runs a filter on each value, prints every result; a filter may give zero, one or many results. Windows install: download `jq-windows-amd64.exe`, check its SHA-256 against `sha256sum.txt` (jq 1.8.2, 2026-06-20), run `jq --version`
- Select: `.key`, `.a.b`, `."odd-key"`, `.[0]`, `.[-1]`, `.[1:3]`, `.[]` (one result per item); a missing key is `null`; `?` hides errors (and mistakes)
- `|` pipes; `,` gives several results; `|` binds loosest, so use parentheses; build with `{name, cost: .price}` and `[ ... ]`
- Everyday: `select`, `map`, `length`, `keys`, `type`, `has`, `in`, `contains`, `add`, `min_by`, `sort_by`, `join`, `index`, `test`, `any`, `all`
- Type order: null < false < true < numbers < strings < arrays < objects; `--arg` is always a string, so use `--argjson` or `tonumber` for numbers
- Options: `-r` raw strings, `-c` compact, `-e` exit status (null/false 1, no output 4), `-n` no input, `-s` slurp, `--arg`, `--argjson`, `--args`, `-f file`, `-a`, `-b`
- Exit statuses seen: 0 ok, 2 cannot open file, 3 compile error, 5 parse or run-time error. Message prefixes: `parse error` = bad data, `error (at ...)` = filter vs data shape, `compile error` = bad filter
- jq keeps big-number text until arithmetic, prints NaN as `null`, accepts a BOM and duplicate keys (last wins), and reads several values in a row
- `curl -s URL | jq`: a 404 with a JSON body looks like success; use `-f` / `--fail-with-body`, `pipefail` or `%{http_code}`; "Invalid numeric literal" usually means HTML or progress text; do not use `2>&1` without `-s`
- Quoting: Bash single quotes; PowerShell 5.1 strips inner double quotes (escape with `\"`, or use `--arg` or `-f`); cmd needs double quotes, not single. On Windows jq writes CRLF unless `-b`

**COMING NEXT**

Chapter 5 is reshaping data with jq: building objects and arrays, `to_entries` and `with_entries`, `group_by` and `unique_by`, `reduce` and variables, conditionals, `try`, paths and `del`, the `@csv`, `@tsv`, `@sh`, `@base64` and `@uri` formats, and writing your own functions. Start with *Working with JSON 5*.

# Working with JSON

Chapter 5 · Reshaping Data with jq

Chapter 4 was about *looking* at JSON. This chapter is about *changing its shape*: turning a list of orders into totals per customer, an array into an object keyed by ID, a nested structure into CSV, or a document with some fields removed. It builds on the filters, options and quoting rules from *Working with JSON 4*, and uses two sample files: the `shop.json` from that chapter and a new `orders.json` of six orders. The chapter ends with a section on what changed in jq 1.7 and 1.8, because several things you will read online about jq are out of date, and one example in jq's own release notes does not hold in the version I tested.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

jq 1.8.2 on Windows 11, called from Git Bash (the same install as Chapter 4), and Python 3.14.0 for the CSV check. The sample data was made up for this chapter. The list of changes between versions comes from the project's `NEWS.md` for 1.8.2, and **each feature I list was run**. **Not run:** jq 1.6 or older, jq 1.7.x or 1.8.0 (so statements about what *older* versions did come from the release notes, not from my tests), Linux and macOS, and the date and time functions, regular-expression functions beyond `test` and `scan`, and `input` / `inputs`.

## The Sample Data

```
[
  {"id": 1, "customer": "ada",   "status": "shipped",   "date": "2026-10-01", "items": [
  {"id": 2, "customer": "grace", "status": "pending",   "date": "2026-10-02", "items": [
  {"id": 3, "customer": "ada",   "status": "shipped",   "date": "2026-10-03", "items": [
  {"id": 4, "customer": "linus", "status": "cancelled", "date": "2026-10-03", "items": [
  {"id": 5, "customer": "grace", "status": "shipped",   "date": "2026-10-05", "items": [
  {"id": 6, "customer": "ada",   "status": "pending",   "date": "2026-10-06", "items": [
]
```

## Building Objects and Arrays

An object literal builds a new object from the current input; an array literal collects every result of its contents into one array. I ran these on `orders.json`:

| FILTER (WITH `-c`) | RESULT |
|---|---|
| `.[] \| {id, customer, n: (.items \| length)}` | six objects, e.g. `{"id":1,"customer":"ada","n":2}` |
| `map({id, total: (.items \| map(.qty * .price) \| add // 0)})` | `[{"id":1,"total":8.99},{"id":2,"total":4.25},{"id":3,"total":5.34},{"id":4,"total":0},{"id":5,"total":17.97},{"id":6,"total":7.96}]` |
| `.[0] \| {id, first: .items[0].sku, skus: [.items[].sku], "order-id": .id, ("k" + "ey"): 1}` | `{"id":1,"first":"A1","skus":["A1","C3"],"order-id":1,"key":1}` |
| `.[0] \| {id, items: [.items[] \| {sku, line: (.qty * .price)}]}` | `{"id":1,"items":[{"sku":"A1","line":7},{"sku":"C3","line":1.99}]}` |

The rules: `{id}` is shorthand for `{id: .id}`; a key that is not a plain word needs quotes (`"order-id"`); a key computed from an expression goes in parentheses (`("k" + "ey")`). Order 4 has an empty `items` list, so `.items | map(...) | add` would give `null`; `add // 0` turns that into `0` (the `//` operator is explained below). Notice the total for order 4 is `0` in the output.

**Two ways to combine objects.** `+` merges at the top level, with the right side winning; `*` merges recursively. I ran both on `{a:{b:1,c:2}}` and `{a:{c:3,d:4}}`: `+` gave `{"a":{"c":3,"d":4}}` (the whole `a` replaced, `b` lost) and `*` gave `{"a":{"b":1,"c":3,"d":4}}`.

## Entries: Turning Objects into Lists and Back

To work on an object's *keys* as data, convert it to a list of `{key, value}` pairs with `to_entries`, change the list, then convert back with `from_entries`. `with_entries(f)` does all three steps:

| FILTER ON `.owner` (`-c`) | RESULT |
|---|---|
| `to_entries` | `[{"key":"first","value":"Ada"},{"key":"last","value":"Lovelace"},{"key":"phone","value":null}]` |
| `with_entries(select(.value != null))` | `{"first":"Ada","last":"Lovelace"}` (drops null fields) |

| FILTER ON `.OWNER` (`-C`) | RESULT |
|---|---|
| `with_entries(.key \|= ascii_upcase)` | `{"FIRST":"Ada","LAST":"Lovelace","PHONE":null}` |
| `with_entries(.value \|= (. // "n/a"))` | `{"first":"Ada","last":"Lovelace","phone":"n/a"}` |
| `map_values(. // "-")` | `{"first":"Ada","last":"Lovelace","phone":"-"}` |
| `del(.phone)` | `{"first":"Ada","last":"Lovelace"}` |

`from_entries` is also the way to build an object from an array. Three ways to turn the four products into an object keyed by SKU all gave `{"A1":40,"B2":0,"C3":120,"D4":15}` for the stock levels: `map({(.sku): .stock}) | add`, `map({key: .sku, value: .stock}) | from_entries`, and `INDEX(.sku) | map_values(.stock)`.

**What `from_entries` accepts is narrower than people assume.** I tried eleven entry shapes on jq 1.8.2:

| ENTRY | RESULT |
|---|---|
| `{"key":"a","value":1}`, `{"name":"a","value":1}`, `{"Name":"a","Value":1}` | `{"a":1}` (works) |
| `{"Key":"a","Value":2}` | `{"a":2}` (works) |
| `{"k":"a","v":1}` and `{"K":"a","V":1}` | **error**: `Cannot use null (null) as object key` |
| `{"key":"a","v":1}` | `{"a":null}` (the value was not found, so it is silently `null`) |
| `{"key":5,"value":1}` | **error**: `Cannot use number (5) as object key` |
| `{"key":true,"value":1}` | **error**: `Cannot use boolean (true) as object key` |
| `{"key":null,"value":1}` | **error**: `Cannot use null (null) as object key` |
| `{"key":"a"}` | `{"a":null}` |

So: use `key` and `value` exactly (or `name` / `Name`, `Key` / `Value`), make keys strings with `tostring`, and be suspicious of a result full of `null`s. A wrong value name does not error; it just gives `null`.

## Sorting, Grouping and Picking

| FILTER ON `orders.json` (`-c`) | RESULT |
|---|---|
| `group_by(.customer) \| map({customer: .[0].customer, orders: length})` | `[{"customer":"ada","orders":3},{"customer":"grace","orders":2},{"customer":"linus","orders":1}]` |
| `group_by(.status) \| map({status: .[0].status, ids: map(.id)})` | `[{"status":"cancelled","ids":[4]},{"status":"pending","ids":[2,6]},{"status":"shipped","ids":[1,3,5]}]` |
| `unique_by(.customer) \| map(.id)` | `[1,2,4]` (the first order of each customer) |
| `[.[].customer] \| unique` | `["ada","grace","linus"]` |
| `sort_by(.customer, -.id) \| map("\(.customer)\(.id)")` | `["ada6","ada3","ada1","grace5","grace2","linus4"]` (customer ascending, then id descending) |
| `sort_by(.date) \| reverse \| map(.id)` | `[6,5,4,3,2,1]` |
| `min_by(.date).id, max_by(.id).id` | `1` and `6` |
| `[.[].items[]] \| group_by(.sku) \| map({sku: .[0].sku, qty: (map(.qty) \| add)})` | `[{"sku":"A1","qty":3},{"sku":"B2","qty":3},{"sku":"C3","qty":8},{"sku":"D4","qty":6}]` |
| `[.[] \| select(.status != "cancelled")] \| group_by(.customer) \| map({customer: .[0].customer, spent: (map(.items \| map(.qty * .price) \| add) \| add)})` | `[{"customer":"ada","spent":22.29},{"customer":"grace","spent":22.22}]` |
| `map(select(.customer \| IN("ada","linus"))) \| map(.id)` | `[1,3,4,6]` |
| `[limit(2; .[] \| .id)]` ; `first(.[] \| select(.status == "pending")).id` | `[1,2]` ; `2` |
| `[.[].id] \| .[2:4], .[:2], .[-2:]` | `[3,4]`, `[1,2]`, `[5,6]` |

`group_by(f)` **sorts by** `f` **first** and returns an array of arrays, one per distinct value, which is why `.[0]` inside each group is a handy way to name the group. Each group is a plain array, so `length` and `map` work on it directly. `unique_by` keeps the first item of each group. A group-by result is the usual first step of a "totals per X" report.

## Variables, `reduce` and `foreach`

`EXPR as $name | ...` stores a value for use later in the same expression. It does not change the input: `.` is still the same. It also destructures: `.[0] as {customer: $c, items: [{sku: $first}]} | [$c, $first]` gave `["ada","A1"]`, and `.items | . as [$a, $b] | [$a.sku, $b.sku]` on order 5 gave `["A1","B2"]`.

`reduce` folds a stream into one value. `foreach` does the same but *emits each intermediate state*:

```
# orders per customer
jq -c 'reduce .[] as $o ({}; .[$o.customer] += 1)' orders.json
# order ids per status
jq -c 'reduce .[] as $o ({}; .[$o.status] += [$o.id])' orders.json
# running count of items, order by order
jq -c '[foreach .[] as $o (0; . + ($o.items | length); {id: $o.id, running: .})]' orders
```

```
{"ada":3,"grace":2,"linus":1}
{"shipped":[1,3,5],"pending":[2,6],"cancelled":[4]}
[{"id":1,"running":2},{"id":2,"running":3},{"id":3,"running":4},{"id":4,"running":4},{"i
```

The pattern `.[$key] += 1` works because a missing key is `null` and `null + 1` is `1`: it needs no setup. Notice the key order in the second result: `reduce` keeps the order it first saw each status (`shipped`, `pending`, `cancelled`), while `group_by` sorted them alphabetically. The two results are the same *data* in a different *order*, and **compare JSON from two different filters with** `jq -S` (sort keys), as the exercise does. `reduce` on an empty stream returns the starting value: `reduce empty as $x (0; . + 1)` gave `0`.

> **A BEHAVIOUR THAT CHANGED TWICE: WHAT HAPPENS WHEN THE UPDATE PRODUCES NOTHING**
>
> The jq 1.8.0 release notes describe a change and show an example: `reduce range(5) as $x (0; .+$x | select($x!=2))` "prints `8`" (the state kept its old value when `select` produced nothing). I ran that

exact filter on jq 1.8.2 and it printed **7**, and the `foreach` example printed `[[0,0],[1,1],[3,3],[4,7]]` instead of the notes' `[[0,0],[1,1],[3,4],[4,8]]`. The reason is in the 1.8.1 notes: "*Revert the change of reduce/foreach state variable in 1.8.0*". In 1.8.2 an empty update resets the state to `null`, so the next step computed `null + 3`, which jq says is 3 (I ran it). **Avoid depending on this:** never let the update of a `reduce` produce `empty`; use `if` inside it to return the unchanged state.

## Conditionals, the `//` Operator, `try` and Early Exit

`if COND then A elif COND2 then B else C end` is an expression. `empty` produces no output, which drops the item: `map(if .status == "cancelled" then empty else .id end)` gave `[1,2,3,5,6]`. The `else` branch is optional since jq 1.7 (the missing branch is `.`): `5 | if . > 3 then "big" end` printed `"big"`. In a condition, **only `false` and `null` are false**: I tested `[1, null, false, 0, "", [], {}]` and got falsy for `null` and `false` only. `0`, `""`, `[]` and `{}` are all *truthy*.

`a // b` ("alternative") means "`a`, unless it is `false`, `null` or empty, then `b`". Same truthiness rule: `[false, null, 0] | map(. // "default")` gave `["default","default",0]`, so a real `0` is kept. `map(.items[0].sku // "none")` gave `["A1","B2","D4","none","A1","C3"]` (order 4 has no items). With a stream on the left it gives *all the truthy values*: `(false, 1, 2) // 9` printed `1` and `2`, and `[(false, null) // 9]` gave `[9]`.

`try` catches an error. `try EXPR` alone produces nothing on error; `try EXPR catch HANDLER` runs the handler on the error message; `EXPR?` is shorthand for `try EXPR`:

| FILTER | RESULT |
|---|---|
| `"abc" \| tonumber` | error: `string ("abc") cannot be parsed as a number` |
| `"abc" \| try tonumber` | no output, no error |
| `"abc" \| try tonumber catch "not a number"` | `"not a number"` |
| `["1","x","3"] \| map(try tonumber catch null)` | `[1,null,3]` |
| `["1","x","3"] \| map(tonumber? // -1)` | `[1,-1,3]` |
| `try error("boom") catch .` | `"boom"` |
| `try error({code: 42}) catch .code` | `42` (you can raise any value) |
| `{"a":1} \| try .a.b catch "caught: \(.)"` | `"caught: Cannot index number with string (\"b\")"` |

Early exit: `first(f)` and `limit(n; f)` stop producing as soon as they have enough, which matters for infinite streams: `[limit(3; repeat("x"))]` gave `["x","x","x"]`. `label $out | 1, 2, break $out, 3` printed only `1` and `2`. Do not overuse `?` and `try`: they hide real mistakes (a misspelt filter inside `try` just silently produces nothing).

## Updating in Place: `=`, `|=`, `+=`, `//=`, `del`

| FILTER (`-N`, `-C`) | RESULT |
|---|---|
| `{} \| .a.b.c = 1` | `{"a":{"b":{"c":1}}}` (missing levels are created) |
| `{"a":[1,2,3]} \| .a[1] = 9 \| .a \|= map(. * 2)` | `{"a":[2,18,6]}` |
| `{"a":1} \| .b //= 5 \| .a //= 7` | `{"a":1,"b":5}` (`//=` only fills missing or null) |
| `{"a":1} \| .a += 1 \| .a *= 3` | `{"a":6}` |
| `{"a":[1,2,3]} \| del(.a[0])`; `del(.a[] \| select(. > 1))` | `{"a":[2,3]}`; `{"a":[1]}` |
| `[1,2,3,4] \| del(.[0,2])` | `[2,4]` (both indexes are removed from the original, not one after the other) |

`=` assigns a value computed from the *whole* input; `|=` updates a location using its *own current value* (`.a |= map(. * 2)`). Prefer `|=` when the new value depends on the old one. Every update returns the whole modified document, not just the part you changed.

## Paths: `paths`, `getpath`, `setpath`, `walk`, `pick`

A *path* is an array like `["a","b",1,"c"]` that leads to a value. The sample is `{"a":{"b":[10,{"c":null}],"d":"x"},"e":true}`:

| FILTER (`-C`) | RESULT |
|---|---|
| `[paths]` | `[["a"],["a","b"],["a","b",0],["a","b",1],["a","b",1,"c"],["a","d"],["e"]]` |
| `[paths(scalars)]` | `[["a","b",0],["a","d"],["e"]]` (paths to values that are not objects or arrays; the `null` at `["a","b",1,"c"]` is not included) |

| FILTER ( `-c` ) | RESULT |
|---|---|
| `getpath(["a","b",1,"c"])` | `null` |
| `getpath(["a","nope","x"])` | `null` (**the same answer for a missing path**) |
| `setpath(["a","b",0]; 99) \| .a.b` | `[99,{"c":null}]` |
| `path(.a.b[1].c)` | `["a","b",1,"c"]` |
| `delpaths([["a","d"],["e"]])` | `{"a":{"b":[10,{"c":null}]}}` |
| `[.. \| numbers]` | `[10]` ( `..` visits every value, at every depth) |
| `[paths \| map(tostring) \| join(".")]` | `["a","a.b","a.b.0","a.b.1","a.b.1.c","a.d","e"]` |
| `walk(if type == "number" then . + 1 else . end)` | `{"a":{"b":[11,{"c":null}],"d":"x"},"e":true}` |
| `walk(if type == "object" then del(.c) else . end)` | `{"a":{"b":[10,{}],"d":"x"},"e":true}` |
| `pick(.a.d, .e)` | `{"a":{"d":"x"},"e":true}` |
| `pick(.a.b[1].c)` | `{"a":{"b":[null,{"c":null}]}}` (an array index is padded with `null` ) |
| `pick(.zzz)` | `{"zzz":null}` (a path that does not exist is added as `null` ) |

Take two warnings from the table. `getpath` cannot tell "the value is `null` " from "there is no such path"; to ask whether a path exists, compare against the list of paths ( `[paths] \| map(. == ["a","b",1,"c"]) \| any` gave `true` , and `false` for a made-up path). And `pick` manufactures `null` s for anything absent, so a typo in a `pick` looks like data. `walk(f)` applies a function to every value from the inside out (on `[[1,2],[3]]` , a rule that joins each array's items with commas gave `"1,2,3"` , so the inner arrays were converted before the outer one), which is the tool for changes that apply "everywhere", such as upper-casing all strings or deleting a key at any depth. `leaf_paths` , which older tutorials use, no longer exists: it was removed in 1.7, and `[leaf_paths]` gave `leaf_paths/0 is not defined` . Use `paths(scalars)` .

## Output Formats: `@csv` , `@tsv` , `@sh` , `@uri` and Friends

Format strings turn an array or string into text for another program. Use them with `-r` so the quotes of the JSON string do not appear. First, CSV with a header row:

```
jq -r '(["sku","name","price"] | @csv), (.products[] | [.sku, .name, .price] | @csv)' sh
```

```
"sku","name","price"
"A1","Tea",3.5
"B2","Coffee",4.25
"C3","Biscuits",1.99
"D4","Milk",0.89
```

`@csv` quotes strings, leaves numbers bare, writes `null` as an empty field and `true` as `true`. To test that it quotes *correctly*, I built a file of five items whose names contained a comma, a double quote, a newline, a `null` and an accented letter, wrote it as CSV with jq, and read the result back with Python's `csv` module, comparing with the original. All five rows matched. As a control, I built the same rows by joining with commas (no quoting) and the same check reported mismatches on four rows. The first result, as CSV text:

```
"id","name","note"
1,"Tea, hot","plain"
2,"say ""hi""","quote inside"
3,"two
lines","newline inside"
4,,"null name"
5,"café","accent"
```

`@csv` accepts only an *array of scalars*: giving it an object failed with `object ({"sku":"A1","name":"Tea",...}) cannot be csv-formatted, only array`, and a row containing an array failed with `array (["drink","hot"]) is not valid in a csv row`. Flatten first: `(.tags // [] | join("|"))` turned the tags into one field (and Milk's missing tags into an empty string).

| FORMAT | EXAMPLE INPUT | OUTPUT (`-R`) |
|---|---|---|
| `@tsv` | `["a\tb", "x\ny", "back\\slash"]` (tab, newline, backslash) | fields joined by a real tab; the tab, newline and backslash inside fields are written as the two characters `\t`, `\n` and `\\` |
| `@sh` | `"it's a test; echo INJECTED"` | `'it'\''s a test; echo INJECTED'` — when I ran that through `eval printf`, the output was the literal text, and nothing was executed |

| FORMAT | EXAMPLE INPUT | OUTPUT (`-R`) |
|---|---|---|
| `@sh` on an array | `["a b", "c'd", 5]` | `'a b' 'c'\''d 5` (one quoted word per item) |
| `@uri` | `"a b&c=d/é?"` | `a%20b%26c%3Dd%2F%C3%A9%3F` |
| `@urid` (1.8) | `"a%20b%26c"` | `a b&c`; and `@uri \| @urid` returned the original string |
| `@base64` / `@base64d` | `"héllo wörld"` | `aMOpbGxvIHfDtnJsZA==`, and `@base64 \| @base64d` returned the original; `"x" \| @base64d` failed: `trailing base64 byte found` |
| `@html` | `"<a href=\"x\">T&C</a>"` | `<a href="x">T&C</a>` |
| `@json`, `@text`, `tojson`, `tostring` | `{"a":[1,"x"]}` | all four printed `{"a":[1,"x"]}` for this object (for the string `"x"`, `@json` and `tojson` gave `"x"` with the quotes, `@text` and `tostring` gave `x`) |

**String interpolation** is `"text \(expression) text"`, and a format name in front applies that format to *each interpolated part*: `"Total: \(.products | map(.price) | add)"` gave `Total: 10.63`; `@uri "https://example.com/search?q=\(.owner.first + " & " + .owner.last)"` gave `https://example.com/search?q=Ada%20%26%20Lovelace` (the literal text is left alone, the interpolated part is encoded); `@sh "echo \(.owner.first) \(.shop)"` gave `echo 'Ada' 'Corner Shop'`. **Always use `@sh` when jq output goes into a shell command**: it is what stopped the "INJECTED" test string from running (*Working with JSON 11* comes back to injection).

## Your Own Functions and Modules

`def name: body;` defines a function that works on its input; `def name(a; b): body;` takes *filter* parameters, separated with semicolons. I tested:

| DEFINITION AND CALL (`-c`) | RESULT |
|---|---|
| `def double: . * 2; .products \| map(.price \| double)` | `[7,8.5,3.98,1.78]` |
| `def addvat(rate): . * (1 + rate); .products \| map(.price \| addvat(0.2) \| . * 100 \| round / 100)` | `[4.2,5.1,2.39,1.07]` |
| `def inStock: select(.stock > 0); [.products[] \| inStock \| .name]` | `["Tea","Biscuits","Milk"]` (a function can be a filter that drops items) |

| DEFINITION AND CALL (`-c`) | RESULT |
|---|---|
| `def fac: if . <= 1 then 1 else . * (. - 1 \| fac) end; [range(1;7) \| fac]` | `[1,2,6,24,120,720]` (recursion works) |
| `def f(x): x + 1; [f(1, 10)]` and `def f($x): $x + 1; [f(1, 10)]` | `[2,11]` for both |
| `def f: 1; def f: 2; f` | `2` (the later definition replaces the earlier) |
| `[...] \| def avg: add / length; avg` on the four stock numbers | `43.75` |

A plain parameter like `rate` is a *filter* that is re-evaluated each time it is used (so it could be `.stock`); a `$rate` parameter is evaluated once to a value. For simple values, `$name` is the safer form. For real reuse, put definitions in a **module**, a file ending in `.jq`. I wrote `lib/shoplib.jq` with three functions:

```
# lib/shoplib.jq
def in_stock: select(.stock > 0);
def money: . * 100 | round / 100;
def with_vat($rate): .price * (1 + $rate) | money;

jq -L lib -c 'import "shoplib" as s; [.products[] | s::in_stock | {name, vat: s::with_va
jq -L lib -c 'include "shoplib"; [.products[] | in_stock | .sku]' shop.json
jq -c 'import "shoplib" as s; 1' shop.json
```

```
[{"name":"Tea","vat":4.2},{"name":"Biscuits","vat":2.39},{"name":"Milk","vat":1.07}]
["A1","C3","D4"]
jq: error: module not found: shoplib          (exit status 3: the last command had no -L
```

`import "name" as prefix;` keeps the functions behind a prefix (`s::money`); `include "name";` pastes them in with no prefix. `-L dir` (long form `--library-path`) says where to look. Without it, jq did not find the module, with a clear message and exit status 3.

## What Changed in jq 1.7 and 1.8 (Each One Run)

jq changed more in 2023 to 2026 than in the previous five years, and many online answers assume 1.6. The list below comes from the project's `NEWS.md`; **every row was run on 1.8.2** and the result is shown. A feature marked 1.7 or 1.8 is not there in older versions: if a script must run on an old jq, check `jq --version` first.

| SINCE | FEATURE | RESULT ON 1.8.2 |
|---|---|---|
| 1.7 | `pick(stream)` | `{"a":1,"b":{"c":2},"x":null}` for `pick(.a, .b.c, .x)`; for an array, `[1,2,3,4] \| pick(.[2])` gave `[null,null,3]` |
| 1.7 | `debug(msgs)` | wrote `["DEBUG:","value is 1"]` to stderr and passed the input on |
| 1.7 | `scan($re; $flags)` | `"abAB" \| [scan("ab"; "i")]` gave `["ab","AB"]` |
| 1.7 | `abs` | `[-5, 5, -0.5] \| map(abs)` gave `[5,5,0.5]` |
| 1.7 | `if` without `else` | `5 \| if . > 3 then "big" end` gave `"big"` |
| 1.7 | `$binding` as an object key; `.a.b.[0]` | `"k" as $k \| {$k, ($k): 1}` gave `{"k":1}`; `{a:{b:[1,2]}} \| .a.b.[0]` gave `1` |
| 1.7 | `leaf_paths` **removed** | `leaf_paths/0 is not defined` |
| 1.7 | `-b` (Windows) and `--raw-output0` | `-b`: Chapter 4; `--raw-output0` separated outputs with a zero byte ( `a` `00` `b` `00` ) |
| 1.8 | `trim`, `ltrim`, `rtrim` | `" hello " \| [trim, ltrim, rtrim]` gave `["hello","hello "," hello"]`; `1 \| trim` failed: `trim input must be a string` |
| 1.8 | `trimstr` | `"foobarfoo" \| trimstr("foo")` gave `"bar"` |
| 1.8 | `add(generator)` | `{"xs":[1,2,3]} \| .sum = add(.xs[])` gave `{"xs":[1,2,3],"sum":6}` |
| 1.8 | `skip(n; f)` | `[1,2,3,4,5] \| [skip(2; .[])]` gave `[3,4,5]` |
| 1.8 | `toboolean` | `["true","false"] \| map(toboolean)` gave `[true,false]` |
| 1.8 | `@urid` | `"%6a%71" \| @urid` gave `jq` |
| 1.8 (breaking) | `tonumber` rejects spaces | `" 12 " \| tonumber` failed; `" 12 " \| trim \| tonumber` gave `12` |

| SINCE | FEATURE | RESULT ON 1.8.2 |
|---|---|---|
| 1.8 (breaking) | `index` counts characters, not bytes | `"héllo" \| index("l")` gave `2`; `utf8bytelength` gave `6` |
| 1.8 (breaking) | `limit` with a negative count is an error; `ltrimstr` on a non-string is an error | `limit doesn't support negative count`; `1 \| ltrimstr("a")` failed with `startswith() requires string inputs` |
| 1.8 | `--indent 0` no longer implies compact output | printed the object one item per line with no indentation |

One feature I expected is **not** there: `toarray`. `[1] | toarray` failed with `toarray/0 is not defined` on 1.8.2, although I had seen it recommended. Another thing to know is the 1.8.0 change in how `as` binds against operators: `[-1 as $x | 1, $x]` gave `[1,-1]` on 1.8.2 (the notes say it used to be `[-1,-1]`), and `1 | . + 2 as $x | -$x` gave `-3`. If an old script behaves differently, these notes are the first place to look.

#### MISTAKES WHILE PREPARING THIS CHAPTER

**1. I wrote `{"k":"b","v":2}` as a valid `from_entries` input from memory.** It failed with `Cannot use null (null) as object key` on 1.8.2, and a wrong value name silently gave `null`. **How to avoid it:** test each form a function claims to accept; the table above is what jq 1.8.2 actually did.

**2. I trusted an example from the 1.8.0 release notes.** It does not hold on 1.8.2 because 1.8.1 reverted the change. **How to avoid it:** read the notes of every release after the one you quote, and run examples on the version you will use.

**3. My first interpolation tests printed `null` everywhere.** My helper function passed `-n` (no input) along with the data file, so there was nothing to interpolate. **How to avoid it:** when every result is `null`, check the options before the filter.

**4. My first "two ways" comparison reported differences that were not differences.** `group_by` sorts the keys and `reduce` keeps first-seen order, so the text differed while the data was identical. **How to avoid it:** compare with `jq -S`, and keep a deliberately different pair as a control so a "same" verdict means something.

**5. My check for "does this path exist" used `index` and said `false` for a path that does exist.** `index` on an array of arrays searches for a *sub-array*, not an item. **How to avoid it:** test a check on a case where you know the answer is true as well as one where it is false.

**6. I assumed `toarray` existed. How to avoid it:** run a function once before you put it in a lesson; "I have seen it recommended" is not evidence.

## Hands-On Exercises

**EXERCISE 1**

Using `orders.json`, write filters that produce: one summary object per order (id, customer, item count, total); spend per customer excluding cancelled orders; orders per status; units sold per SKU; a running count of items with `foreach`; and the busiest date with its order ids. For the three "totals" questions answer each *two ways* (`group_by` and `reduce`) and have a script compare the two answers with `jq -S`. Include a deliberately different pair to prove the comparison can report a difference.

View solution

**EXERCISE 2**

On the owner object, drop null fields, upper-case the keys and default the null values. Then find out exactly which entry shapes `from_entries` accepts by trying eleven (`key/value`, `name/value`, `k/v`, numeric and boolean keys, a missing value and so on) and recording the result or error of each. Build an object keyed by SKU in three ways. On a small nested document, show `paths`, `getpath` for a present and a missing path, an existence check that returns true for a real path and false for a fake one, `setpath`, `del`, `walk` with a before-and-after count, `pick` (including a missing path) and the removed `leaf_paths`.

View solution

**EXERCISE 3**

Write a CSV (with a header) from five records whose names contain a comma, a double quote, a newline, a null and an accented letter, and verify it by parsing it back with Python's `csv` module; include a control built with `join(",")` that must fail the same check. Show `@sh` making a hostile string safe to `eval`, and `@uri` / `@urid` and `@base64` / `@base64d` round trips. Write a module with three functions and use it with `import` and `include`, including the error when `-L` is missing. Finally run the release-notes `reduce` example and your list of 1.7 and 1.8 features on your own jq version, and say which results differ from the notes.

View solution

### CHAPTER 5 QUICK REFERENCE

- Build: `{id, total: (...)}` (shorthand, parentheses for computed keys), `[ ... ]` collects all results; `+` merges objects at the top level, `*` recursively; `add // 0` for an empty list
- Entries: `to_entries`, `from_entries` (use `key`/`value`, `name`/`Name` or `Key`/`Value`; keys must be strings; `k`/`v` failed on 1.8.2), `with_entries(f)`, `map_values(f)`, `INDEX(.k)`
- `group_by(f)` sorts by f then makes one array per value; `unique_by`, `sort_by(a, -b)`, `min_by`/`max_by`, `limit`, `first`; compare two filters' results with `jq -S`
- `EXPR as $v | ...` (also destructuring); `reduce STREAM as $x (init; update)`; `foreach` emits each state; never let a reduce update produce `empty` (1.8.0 notes say 8, 1.8.2 gave 7)
- Only `false` and `null` are false; `a // b` keeps `0` and `""`; `try`, `try ... catch`, `?`; `label`/`break`; `first(f)` stops early
- Update: `=` (value from the whole input), `|=` (from the old value), `+=`, `//=`, `del(path)`, `delpaths`
- Paths: `paths`, `paths(scalars)`, `getpath` (null for missing and for null), `setpath`, `path(f)`, `..`, `walk(f)`, `pick(...)` (pads missing paths with null); `leaf_paths` was removed in 1.7
- Formats (use `-r`): `@csv` (array of scalars only), `@tsv`, `@sh` (always for shell commands), `@uri`/`@urid`, `@base64`/`@base64d`, `@html`, `@json`; a format before an interpolated string applies to each interpolated part
- `def f: ...;`, `def f(x; y): ...;` (`$x` evaluates once); modules: `-L dir`, `import "m" as p;` (`p::f`) or `include "m";`; a missing module gives exit status 3
- New since 1.7: `pick`, `debug(msgs)`, `scan($re; $flags)`, `abs`, `if` without `else`, `--raw-output0`. New in 1.8: `trim`/`ltrim`/`rtrim`, `trimstr`, `add(f)`, `skip`, `toboolean`, `@urid`; `tonumber` rejects spaces and `index` counts characters. `toarray` was not defined on 1.8.2

**COMING NEXT**

Chapter 6 moves from querying data to *describing* it: JSON Schema. It covers what a schema is for, the 2020-12 vocabulary (`type`, `properties`, `required`, `items`, `enum`, `$ref` and `$defs`), why `format` is only an annotation by default, and writing a schema for a real API response. Start with *Working with JSON 6*.

# Working with JSON

Chapter 6 · JSON Schema: Describing Data

Until now JSON has been "whatever the sender produced". That works until a field is missing, a number arrives as a string, or a price is zero, and the failure shows up three steps later in code that trusted the data. **JSON Schema** is a standard way to write down what a valid document looks like, as JSON, so a program can check it. A schema is a *contract*: it can reject bad input at the door, document an API for humans, and feed editors and code generators. This chapter covers the vocabulary of the current version, the defaults that surprise people, how to write a schema for a real API response, and how to read a validator's error output. The next chapter covers validators in more depth (combining schemas, conditionals, editor integration).

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Two validators, both installed in a scratch folder: **Ajv 8.20.0** in Node 24.18.0 (class `ajv/dist/2020`, with **ajv-formats 3.0.1**) and **jsonschema 4.26.0** in Python 3.14.0 (`Draft202012Validator`). Where a result is the same in both I say so; where it differs I show both. **Not run:** any other validator (Go, Java, .NET, Rust), the draft-07 and 2019-09 behaviour of these libraries, the unreleased "v1" specification, and keywords not listed here (`if/then/else`, `allOf`, `dependentRequired` and `unevaluatedProperties` are Chapter 7). The version history comes from json-schema.org.

## What a Schema Is For

- **Validation:** "is this document acceptable?" with a list of exactly what is wrong. This is the main use, in API servers, config loaders and data pipelines.
- **Documentation:** a schema says which fields exist, their types, their limits and which are required, in a form tools can read.
- **Tooling:** editors can offer completion and red squiggles from a schema, and some tools generate types or forms from one (Chapter 7).

What a schema does *not* do: it does not change the data (it will not turn `"1"` into `1`), it does not check things that need other data (is this customer id in the database?), and it does not make a bad design good. It answers one question, "is this JSON the right shape?".

## Your First Schema

A schema is itself a JSON document, usually an object, made of *keywords*. Each keyword adds one rule, and a document is valid only if it passes *all* the rules. Here is a schema for one product from `shop.json`:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "sku":   { "type": "string",  "pattern": "^[A-Z][0-9]$" },
    "name":  { "type": "string",  "minLength": 1 },
    "price": { "type": "number",  "exclusiveMinimum": 0 },
    "stock": { "type": "integer", "minimum": 0 },
    "tags":  { "type": "array",   "items": { "type": "string" }, "uniqueItems": true }
  },
  "required": ["sku", "name", "price", "stock"]
}
```

Read it as a list of conditions. It must be an object. If it has `sku`, that must be a string like `A1`; if it has `stock`, an integer of zero or more; and four properties must be present. The `$schema` line says which version of the standard the schema is written for (see "Versions" below).

## The Core Keywords

I tested every keyword below in **both** validators with documents that should pass and fail; where a cell shows a result, both gave it unless noted.

| KEYWORD | MEANING | TESTED BEHAVIOUR |
|---|---|---|
| `type` | `"string"`, `"number"`, `"integer"`, `"boolean"`, `"null"`, `"array"`, `"object"`, or a list of them | `["string","null"]` accepted `"x"` and `null`, rejected `1`; `"integer"` accepted the JSON text `1.0` and `1e2` but rejected `1.5` and `true` |
| `enum` | the value must equal one of a list | `["a", 1, null, {"x":1}]` accepted each of those exactly; `"1"` and `{"x":2}` failed; `enum: [1]` accepted `1` and `1.0` but rejected `true` |

| KEYWORD | MEANING | TESTED BEHAVIOUR |
| --- | --- | --- |
| `const` | the value must equal this one value | `{"a":[1,2]}` passed and `{"a":[2,1]}` failed (array order matters) |
| `properties` | a schema for each named key, *if present* | `{}` passed a schema with only `properties`: nothing is required by it |
| `required` | these keys must be present | passed for an object that had the key, failed for `{}` |
| `additionalProperties` | a schema (or `false`) for keys not listed in `properties` | `false` rejected `{"a":1,"b":2}`; a schema `{"type":"integer"}` accepted `b: 1` and rejected `b: "y"` |
| `patternProperties`, `propertyNames`, `minProperties` | rules by key pattern; a schema for the key *names*; a minimum key count | `{"^x-": {"type":"string"}}` checked only `x-` keys; `propertyNames: {pattern: "^[a-z]+$"}` rejected the key `A-b`; `minProperties: 1` rejected `{}` |
| `items`, `prefixItems` | `items`: a schema for every array element (after any `prefixItems`); `prefixItems`: one schema per leading position | `items: {type: integer}` rejected `[1,"x"]`; `prefixItems: [string, integer]` with `items: false` accepted `["a",1]` and `["a"]` but rejected `["a",1,2]` and `[1,"a"]` |
| `minItems`, `maxItems`, `uniqueItems` | array length limits; no duplicates | `uniqueItems` rejected `[1,2,1]`, `[{"a":1},{"a":1}]` and `[[1],[1]]`, and accepted `[1,"1"]`; in Python also rejected `[1, 1.0]` and accepted `[1, true]` |
| `contains`, `minContains` | at least one (or n) element must match | `contains: {type: number, minimum: 5}` accepted `[1,9]` and rejected `[1,2]` and `[]`; `minContains: 2` with `const: 1` accepted `[1,1]` |
| `minLength`, `maxLength`, `pattern` | string length (in characters); a regular expression | `maxLength: 1` accepted a single emoji in both (counted as one character); `pattern: "[0-9]"` accepted `"a1"` (see the pitfalls below) |
| `minimum`, `maximum`, `exclusiveMinimum`, `exclusiveMaximum` | number limits, inclusive or exclusive | `exclusiveMinimum: 0` rejected `0` and `-1`, accepted `0.0001` |

| KEYWORD | MEANING | TESTED BEHAVIOUR |
| --- | --- | --- |
| `multipleOf` | the number must be a whole multiple | `multipleOf: 1` accepted `3` and rejected `3.5`; with `0.01` see the pitfalls |

Two more building blocks need no keyword at all: an **empty schema `{}` accepts everything** (I fed it a number, a string, `null`, an array and an object), and a schema can be just `true` (accept everything) or `false` (reject everything). `false` is useful as `items: false` or `additionalProperties: false`.

## Defaults That Surprise People

- **Nothing is required unless you say so.** `properties` only describes keys that are there; an object with none of them still passes. Add `required`.
- **Extra properties are allowed.** In my shop test, adding an unknown top-level field to a valid document left it valid. Add `additionalProperties: false` only when you really want a closed shape, and read the Chapter 7 warning before combining it with `allOf`.
- **Keywords are ignored for the wrong type.** `{"required": ["a"]}` alone rejected `{}` but *accepted* `[]` and `"x"`, because `required` only applies to objects. State the `type` too: `{"type":"object","required":["a"]}` rejected all three.
- **Integers are about value, not spelling.** The JSON text `1.0` and `1e2` are valid integers; `true` is not an integer, in either validator. (In Node, `JSON.parse("1.0")` is already the number 1, so the JavaScript side cannot even see the difference.)
- **`pattern` is not anchored.** `"[0-9]"` accepted `"a1"`. Write `^...$` when the whole string must match. In JSON text a backslash in a pattern is doubled (`"^[0-9]+\\.[0-9]{2}$"` is the regular expression `^[0-9]+\.[0-9]{2}$`).
- **Unknown keywords are ignored, silently, by default.** Python's `jsonschema` returned valid for `{"type":"object","foo":1}`, so a typo such as `"requird"` is not an error. See "Strictness" below.

## `format`: Often Not Checked

`format` names a common pattern (`date`, `date-time`, `email`, `uuid`, `uri` and others). In the 2020-12 specification it is an *annotation* by default: a validator is allowed to ignore it, and the

validators I ran did. The same string `"2026-13-45"` against `{"type":"string","format":"date"}`:

| SETUP | `"2026-13-45"` | `"2026-10-12"` |
|---|---|---|
| Ajv with `validateFormats: false` | valid | - |
| Ajv with `ajv-formats` added | **invalid** (`must match format "date"`) | valid |
| Ajv with default options and no plugin | throws at compile time: `unknown format "date" ignored in schema at path "#"` | |
| Python `jsonschema`, no `format_checker` | valid | - |
| Python with `format_checker=FormatChecker()` | **invalid** (`'2026-13-45' is not a 'date'`) | valid |

So the same schema can accept a nonsense date in one program and reject it in another. **If you rely on `format`, turn format checking on explicitly** (Ajv: add `ajv-formats`; Python: pass a `FormatChecker`), and for anything critical add a `pattern` as well. I only tested `date`; which other formats a checker supports depends on the library and on extra packages. The January 2025 post on json-schema.org says the planned next specification will make `format` validate by default (see "Versions").

## `$schema`, `$id`, `$defs` and `$ref`

Repeating the same sub-shape (an order's items) in several places invites mistakes. Put it under `$defs` once and point to it with `$ref`. The pieces:

- **`$schema`**: which version of the rules this document uses. Include it. Python's `validators.validator_for(schema)` picked `Draft202012Validator` from it. In Ajv, the *class* you choose must match: the default `new Ajv()` failed on a 2020-12 schema with `no schema with key or ref "https://json-schema.org/draft/2020-12/schema"`; use `require('ajv/dist/2020')`.
- **`$id`**: the schema's own URI (it does not need to exist online). Other schemas refer to it. In Ajv, a schema added with `addSchema` under its `$id` could be referenced from another schema, and an unknown id gave `can't resolve reference https://example.com/nope.json from id #`.

- `$defs`: a place for named sub-schemas. `{"$ref": "#/$defs/item"}` means "the schema at that path in this document". A missing target gave `can't resolve reference #/$defs/missing from id #` at compile time.
- `$anchor`: a plain-name target: a definition with `"$anchor": "num"` was reachable as `"$ref": "#num"` (tested in Ajv).
- **Recursion** is allowed: a `node` definition whose `children` are `$ref: "#/$defs/node"` validated a three-level tree and rejected a leaf missing `name` (tested in Ajv).
- **Siblings of `$ref`** are allowed since 2019-09: `{"$ref": "#/$defs/n", "minimum": 10}` where `n` is an integer rejected `5` (the sibling applied) and `"x"`, and accepted `15` (the same in both validators). Older drafts ignored siblings of `$ref`.

## A Schema for a Real API Response

Here is a schema for the order documents from Chapter 5, using `$defs` for the item shape and enums for the status. It is the schema used for every test in this chapter:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/schemas/order.json",
  "title": "Order",
  "type": "object",
  "properties": {
    "id":       { "type": "integer", "minimum": 1 },
    "customer": { "type": "string", "minLength": 1 },
    "status":   { "enum": ["pending", "shipped", "cancelled"] },
    "date":     { "type": "string", "format": "date" },
    "items":    { "type": "array", "items": { "$ref": "#/$defs/item" } }
  },
  "required": ["id", "customer", "status", "date", "items"],
  "additionalProperties": false,
  "$defs": {
    "item": {
      "type": "object",
      "properties": {
        "sku":   { "type": "string", "pattern": "^[A-Z][0-9]$" },
        "qty":   { "type": "integer", "minimum": 1 },
        "price": { "type": "number", "exclusiveMinimum": 0 }
      },
      "required": ["sku", "qty", "price"],
```

```
        "additionalProperties": false
      }
    }
  }
```

All six real orders from Chapter 5 validated in both validators, *including order 4*, which has an empty `items` list (the schema has no `minItems`, so empty is allowed; that is a design choice you make deliberately). Then I ran 14 test documents through both validators and compared their answers. The results, as Ajv reported them (keyword at path):

| TEST DOCUMENT | RESULT |
|---|---|
| a valid order | valid |
| no `items` | `/` required |
| `id` is the string `"1"` | `/id` type |
| `id` is 0 | `/id` minimum |
| status `"lost"` | `/status` enum |
| an extra field `note` | `/` additionalProperties |
| date `"2026-13-45"` | `/date` format |
| sku `"a1"` | `/items/0/sku` pattern |
| qty 0 and price 0 | `/items/0/price` exclusiveMinimum; `/items/0/qty` minimum |
| an item with no `price` | `/items/0` required |
| empty customer | `/customer` minLength |
| an array instead of an object; `null` | `/` type (both) |
| everything wrong at once | seven errors, one per mistake |

The two validators agreed on the verdict and on the keyword and path of every error in all 14 cases; the only differences are wording (below). As a check on the comparison, I deleted one Python result on purpose and the comparison reported exactly that case.

## Reading Validator Output

A validator gives you either a yes/no or a list of error records. Each record answers four questions: *where in the document*, *which rule*, *why*, and *where in the schema*. For the "everything wrong" document:

```
Ajv, allErrors: false  ->  1 error(s):
                     additionalProperties  must NOT have additional properties

Ajv, allErrors: true   ->  7 error(s):
                     additionalProperties  must NOT have additional properties
   /id               minimum               must be >= 1
   /customer         minLength             must NOT have fewer than 1 characters
   /status           enum                  must be equal to one of the allowed values
   /items/0/sku      pattern               must match pattern "^[A-Z][0-9]$"
   /items/0/qty      minimum               must be >= 1
   /items/0/price    exclusiveMinimum      must be > 0

Python iter_errors()   ->  7 errors, e.g.:
   path ['id']            validator minimum  schema_path ['properties','id','minimum']
   path ['items',0,'sku'] validator pattern  schema_path ['properties','items','items','
   path []                validator additionalProperties   Additional properties are not
```

- **Ajv stops at the first error by default.** With `allErrors: false` it reported one error; with `allErrors: true` it found all seven. Use `allErrors: true` to show a user everything at once (I did not measure how much slower it is, and I did not check Ajv's own advice about using it on untrusted input).
- **Each Ajv error** is an object with `instancePath` (`"/items/0/sku"`, a JSON Pointer: Chapter 8), `schemaPath` (`"#/$defs/item/properties/sku/pattern"` style), `keyword`, `params` (for example `{"additionalProperty":"note"}`) and `message`. The root of the document is the empty string `""`.
- **Python:** `iter_errors(doc)` yields every error; `validate(doc)` raises the first one (for this document `"0 is less than the minimum of 1"`), and `jsonschema.exceptions.best_match(errors)` picked `additionalProperties`. Different tools can pick a different "first" error, so do not build logic on which one comes first. Each error has `.path` (a list such as `['items', 0, 'sku']`), `.validator` (the keyword), `.message` and `.schema_path`.
- **Messages differ** (Ajv: `must be >= 1`; Python: `0 is less than the minimum of 1`), so match on `keyword` and path, never on message text.

## Strictness: Typos in Schemas

A schema with a mistake in it can silently validate nothing. The two libraries guard against that differently:

| SCHEMA PROBLEM | AJV (DEFAULT OPTIONS) | PYTHON `jsonschema` |
|---|---|---|
| unknown keyword `"foo": 1` | throws `strict mode: unknown keyword: "foo"` | silently ignored (`is_valid` returned `True`) |
| `properties` without `type: object` | compiled, with a warning: `strict mode: missing type "object" for keyword "properties" (strictTypes)` | no warning |
| `required` names a property not in `properties` | compiled (`strictRequired` is off by default; with `strict: true` it threw) | no warning |
| `{"type": "strng"}`, `{"minimum": "x"}`, `{"required": "a"}` | rejected when compiled | `Draft202012Validator.check_schema(schema)` rejected each (`'strng' is not valid under any of the given schemas`, `'x' is not of type 'number'`, `'a' is not of type 'array'`); **you must call it yourself**. (A misspelt type name such as `strng` also raised `UnknownType` the first time the validator was used; an unknown keyword never raises.) |

The habit: **always call `check_schema` in Python** (or compile in Ajv) when a schema loads, and keep a test with one valid and one invalid document for every schema, so a typo shows up as a test that no longer fails.

## Two Number Pitfalls

**`multipleOf` with a decimal fails on valid prices.** I checked `multipleOf: 0.01` in both validators:

```
multipleOf 0.01      19.99=NO  4.25=ok  0.07=NO  1.005=NO  0.1=ok       (same result in
integer cents        1999=ok   425=ok   7=ok     100.5=NO
price as text        "19.99"=ok  "4.250"=NO  "5"=NO  "1x99"=NO      (pattern ^[0-9]+\.[
```

`19.99` and `0.07`, perfectly good prices, were rejected: they are not exact multiples of `0.01` in binary floating point (this is the same cause as `0.1 + 0.2` in Chapter 3). So avoid `multipleOf` with a fraction. Send money as **integer cents** (`"type": "integer"`, which is exact) or as a **string** with a `pattern`.

**Large and precise numbers.** A schema validates what the parser produced, so the JavaScript problem from Chapter 1 applies: a 20-digit id arrives in Node as an already-rounded number, and the schema sees the rounded one. A schema cannot detect digits that the parser lost before validation.

## Versions: Which Draft?

JSON Schema is published as dated drafts. From json-schema.org's specification links page: Draft 4 (31 January 2013), Draft 6 (21 April 2017), Draft 7 (19 March 2018), **2019-09** (17 September 2019) and **2020-12**, published 16 June 2022, which the site calls "the current version" and which is the newest *published* one. The meta-schema URI is `https://json-schema.org/draft/2020-12/schema` (I fetched it; the server returned it as `application/schema+json`).

**The "v1" release.** The project has been working on a new, more stable release with strict compatibility rules. A January 2025 post on json-schema.org describes the upcoming release as breaking some things in 2020-12 ("`format` will validate by default", extra data needs an `x-` prefix, `$dynamicRef` is separated from `$ref`), says it was "really close", and gives no date. **As of 2026-10-12 I could not find a published v1:** the specification page and the specification-links page list 2020-12 as newest, and a "v1/2026" address mentioned in the project's process document returned a 404 when I requested it. So write for 2020-12, say so in `$schema`, and re-check the site before assuming anything else. Differences you may meet between drafts in older material: `items` as an array (replaced by `prefixItems` in 2020-12), `definitions` instead of `$defs`, `dependencies` split into `dependentRequired` and `dependentSchemas`, and siblings of `$ref` ignored before 2019-09. I did not test any draft older than 2020-12.

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My first keyword tests were mostly "COMPILE ERROR".** I had set Ajv's `strict` option to `true`, which turns every strictness check (including "missing type") into an error, whereas the default is much milder. **How to avoid it:** use the library's default options for the first experiment, then tighten them; and say which options you used.

**2. I "tested" the integer `1.0` in JavaScript.** `JSON.parse("1.0")` is just `1`, so the test could not show anything. **How to avoid it:** check that the value you test actually reaches the validator as what you think; I repeated the test with the JSON text in both languages.

**3. My price pattern accepted any character where the dot should be.** I typed `"^[0-9]+\.[0-9]{2}$"` in JavaScript source, where `\.` is just `.`, so the regular expression's dot matched anything. **How to avoid it:** double the backslash in source code, print the pattern, and include a control that must fail (`"1x99"`, which now does).

**4. My first "allErrors" demonstration only ever printed the `false` case.** A stray `break` ended the loop. **How to avoid it:** when two settings are being compared, check that the output shows both before reading the result.

## Hands-On Exercises

**EXERCISE 1**

Write a 2020-12 schema for `shop.json` (shop name, open flag, an owner whose `phone` may be a string or `null`, and a non-empty list of unique products with a SKU pattern, a positive price, an integer stock and optional unique tags, using `$defs` and `$ref`). Check that the real file is valid, then write fifteen mutated copies, each with the keyword and path you expect to complain (three of them controls that must stay valid, such as an extra field), and report how many behaved as predicted.

View solution

**EXERCISE 2**

In both Ajv and Python's `jsonschema`, demonstrate the schema traps: the JSON text `1.0` as an integer, `format` with and without a format plugin or checker, `required` without `type: object`, an unanchored `pattern`, an emoji against `maxLength: 1`, `multipleOf: 0.01` on prices and two exact alternatives (integer cents, a string with a pattern, with a control that must fail), an unknown keyword under each library's strictness, and `check_schema` on three broken schemas.

View solution

**EXERCISE 3**

Write the order schema from this chapter. Run fourteen documents (one valid and thirteen invalid in different ways) through both validators, each writing its results (keyword and path of every error, sorted) to a file, and write a third script that compares the two files and lists any case where they disagree. Add a control that deliberately damages one result and shows the comparison catches it. Then show Ajv's default class failing on a 2020-12 schema, a missing `$ref` and an unknown `$id`, and Ajv with `allErrors` off and on.

View solution

## CHAPTER 6 QUICK REFERENCE

- A schema is a JSON document of keywords; a document is valid if it passes *all*. It checks shape, never changes data. Write `"$schema": "https://json-schema.org/draft/2020-12/schema"` (2020-12, published 16 June 2022, is the newest published version; I found no published "v1" on 2026-10-12)
- Types: `type` (or a list such as `["string","null"]`), `enum`, `const`; objects: `properties`, `required`, `additionalProperties`, `patternProperties`, `propertyNames`, `minProperties`; arrays: `items`, `prefixItems`, `minItems`, `maxItems`, `uniqueItems`, `contains`; strings: `minLength`, `maxLength`, `pattern`; numbers: `minimum`, `exclusiveMinimum`, `maximum`, `multipleOf`
- Defaults: nothing required, extra properties allowed, keywords ignored for the wrong type (`required` alone accepts `[]`), `{}` accepts everything, `1.0` and `1e2` are integers, `true` is not, `pattern` is unanchored
- `format` was ignored without Ajv's `ajv-formats` or Python's `FormatChecker()`; with them `"2026-13-45"` failed `date`; Ajv throws on an unknown format when no plugin is present
- Reuse: `$defs` + `{"$ref": "#/$defs/name"}`, `$id` + `addSchema`, `$anchor`, recursion; siblings of `$ref` apply (2019-09+); Ajv needs `ajv/dist/2020` for 2020-12
- Output: Ajv errors have `instancePath`, `schemaPath`, `keyword`, `params`, `message` and it stops at the first error unless `allErrors: true`; Python `iter_errors` gives all, with `.path`, `.validator`, `.message`; match on keyword and path, not message text
- Strictness: Ajv throws on unknown keywords by default; Python ignores them silently, so call `check_schema`; keep one passing and one failing test document per schema

- Avoid `multipleOf` with a fraction (`19.99` failed `multipleOf: 0.01` in both libraries); use integer cents or a string with a `pattern`

**COMING NEXT**

Chapter 7 is validating in practice: Ajv in Node and `jsonschema` in Python in more depth (compile once, formats, good error messages), combining schemas with `allOf`, `anyOf`, `oneOf` and `not`, `if/then/else`, `dependentRequired`, the `unevaluatedProperties` fix for the `additionalProperties` problem, schemas in the editor, generating types or docs, and compatible schema changes. Start with *Working with JSON 7*.

# Working with JSON

Chapter 7 · Validating with JSON Schema

Chapter 6 taught the vocabulary. This chapter is about *using* it in a real program: how to run a validator without making it slow, how to express "one of these shapes" and "if this, then that", how to keep a schema closed without breaking it, how to get error messages a human can act on, how to see the schema working in your editor, how to generate types and documentation from it, and how to change a schema without breaking the people who depend on it. For where a schema sits in API design (request and response contracts), see *API Types & Design* 4.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

Node 24.18.0 with **Ajv 8.20.0**, **ajv-formats 3.0.1** and **ajv-errors 3.0.0**; Python 3.14.0 with **jsonschema 4.26.0**; **json-schema-to-typescript 16.0.0** with **TypeScript 7.0.2**; **json-schema-for-humans 1.5.1**; and **vscode-json-languageservice 6.0.0-next.4**, all installed in a scratch folder (nothing global). **Timings are single runs on one Windows 11 machine**: use them for the *ratios*, not the absolute values. **Not run:** the VS Code window itself (I ran the JSON language service package it is built on, below), other validators, and other generators.

## Run the Validator Efficiently: Compile Once

A validator library reads your schema and builds a checking function. In Ajv that step is *compiling* and it is expensive; running the compiled function is very cheap. I validated the six real orders, repeated to 20,000 documents:

| WHAT THE CODE DID (AJV, NODE) | TIME |
|---|---|
| compile the schema once | 39.7 ms, one time |
| then validate 20,000 documents with the compiled function | 8.1 ms total (0.40 microseconds each); all 20,000 valid |
| `ajv.validate(schemaObject, doc)` with the *same* schema object, 20,000 times | 20.4 ms (Ajv caches by object) |
| create a new Ajv and compile for *each* document, 200 documents | **1,943.5 ms (9.7 ms each)**, about 24,000 times slower per document than the compiled function |

| WHAT THE CODE DID (AJV, NODE) | TIME |
| --- | --- |
| `ajv.validate(freshCopyOfSchema, doc)` on one Ajv | threw on the second document: `schema with key or id "https://example.com/schemas/order.json" already exists` |

The last row is a trap worth remembering: a schema with an `$id` can only be registered once per Ajv. Code that re-reads or re-creates the schema for every request will either be slow (new Ajv each time) or throw (same Ajv, new copy). **Compile at start-up, keep the function, call it for every document.**

Python's `jsonschema` has a different cost profile:

| WHAT THE CODE DID (PYTHON) | TIME |
| --- | --- |
| `v = Draft202012Validator(schema, format_checker=FormatChecker())`, once | 0.1 ms |
| then `v.is_valid(doc)`, 5,000 times | 313.6 ms (62.7 microseconds each) |
| `jsonschema.validate(doc, schema)`, 500 times | 1,211 ms (2.42 ms each) |
| `Draft202012Validator(schema).is_valid(doc)`, 500 times (new validator each time) | 34.8 ms (0.07 ms each) |
| `Draft202012Validator.check_schema(schema)` alone, 500 times | 1,773 ms (3.55 ms each) |

Building a Python validator is cheap, but `jsonschema.validate()` is the slow convenience function: it checks that the *schema* is valid on every call (given a broken schema it raised `SchemaError`, while `Draft202012Validator(bad).is_valid` did not check it, and a misspelt type name raised `UnknownType` only at use time). Build one validator object, call `check_schema` once when the schema loads, then call `is_valid` or `iter_errors` per document. Python was about 150 times slower per document than compiled Ajv here (62.7 versus 0.40 microseconds); that is rarely a problem unless you validate millions of documents.

## Combining Schemas: `allOf`, `anyOf`, `oneOf`, `not`

| KEYWORD | VALID WHEN | TESTED |
| --- | --- | --- |
| `allOf` | every sub-schema passes | `[{type: string}, {minLength: 3}, {pattern: "^a"}]` accepted `"abc"`; rejected `"ab"` (minLength), `"xyz"` (pattern) and `5` (type) |

| KEYWORD | VALID WHEN | TESTED |
|---|---|---|
| `anyOf` | at least one passes | `[string, number]` accepted `"x"` and `1`; `null` failed with three errors |
| `oneOf` | **exactly** one passes | the same disjoint pair behaved like `anyOf`; but `[integer, number]` rejected `1` (it matches both: Ajv reported `passingSchemas: [0,1]`) and accepted `1.5`, while `anyOf` accepted both |
| `not` | the sub-schema does *not* pass | `{not: {type: null}}` rejected `null` (`must NOT be valid`) and accepted `0` and `"x"` |

The usual mistake is `oneOf` **where** `anyOf` **was meant**: if the branches overlap, a document that matches two fails. Use `oneOf` only when the branches are genuinely exclusive (usually because a field such as `type` differs). For "a string or null", prefer `"type": ["string", "null"]`, which gives one error instead of three. All of these gave the same verdicts in Python's `jsonschema`.

## Conditionals: `if` / `then` / `else` and `dependentRequired`

`if` picks a branch: when its schema passes, `then` must pass; otherwise `else` must. A UK address needs a postcode and a US address needs a state and a zip:

```
{
  "type": "object", "required": ["country"],
  "properties": { "country": { "enum": ["GB", "US", "FR"] } },
  "if": { "properties": { "country": { "const": "GB" } } },
  "then": { "required": ["postcode"] },
  "else": {
    "if": { "properties": { "country": { "const": "US" } } },
    "then": { "required": ["state", "zip"] }
  }
}
```

```
{"country":"GB","postcode":"EH1 1YZ"}      valid
{"country":"GB"}                          INVALID   required: must have required property
{"country":"US","state":"NY","zip":"10001"}  valid
{"country":"US","state":"NY"}             INVALID   required: must have required property
```

```
{"country":"FR"}                    valid
{}                                  INVALID   postcode required (the "then" branch
```

The last row is the `if` trap. `{}` has no `country`, and an `if` made only of `properties` *passes* when the property is absent, so the document was sent down the `then` branch and was also told that `postcode` is missing, which is a confusing message. **Put `"required": ["country"]` inside the `if`** so it only fires when the property is really there (the payment schema below does). Python gave the same verdicts.

Two lighter tools: `dependentRequired` says "if this property is present, these others must be too" (`{"card": ["expiry", "cvv"]}` accepted `{}` and a full card, rejected `{"card":"4111"}`; Ajv printed one message per missing property, so two lines), and `dependentSchemas` applies a whole schema when a property is present (`{"card": {required: ["billing"], properties: {billing: {type: "object"}}}}` rejected a card with no billing, and a card with `billing: "no"`).

## Closed Objects: `additionalProperties` versus `unevaluatedProperties`

`additionalProperties: false` means "no keys except the ones in *this schema's own* `properties`". It cannot see inside `allOf`. A common design, a base schema plus extras, therefore breaks:

```
const base = { type: 'object', properties: { id: { type: 'integer' } }, required: ['id']

// A: closed with additionalProperties (the problem)
{ allOf: [base], properties: { name: { type: 'string' } }, additionalProperties: false }
// B: closed with unevaluatedProperties (the fix)
{ allOf: [base], properties: { name: { type: 'string' } }, unevaluatedProperties: false
```

```
A:  {"id":1}                    INVALID  additionalProperties   (id is not in A's own
A:  {"id":1,"name":"x"}         INVALID  additionalProperties
B:  {"id":1}                    valid
B:  {"id":1,"name":"x"}         valid
B:  {"id":1,"name":"x","extra":1}  INVALID  unevaluatedProperties
```

Version A rejects *every* document, including the simplest valid one, because `id` is declared inside `allOf` where A's `additionalProperties` cannot see it. `unevaluatedProperties` (new in

2019-09) means "no keys that *no* keyword in this schema or its `allOf` / `$ref` / `if` branches evaluated", so B is closed and still allows both `id` and `name`. The older workaround is to repeat `additionalProperties: false` and every property in each branch, which I also tried: the base branch with `additionalProperties: false` then rejected `name`. `unevaluatedItems` is the same idea for arrays (`prefixItems` with `unevaluatedItems: false` accepted `["a"]` and rejected `["a", 1]`). Ajv and Python gave identical verdicts for all of these (except that the repeated-branch variant was run in Ajv only).

A caveat that I met in the payment exercise: when a `then` branch *fails*, the properties it would have evaluated count as unevaluated, so the validator also reports `unevaluatedProperties` errors for them. For the document `{"method":"card","amount":500,"card_number":"411111111111111","expiry":"10/29"}` (15 digits) Ajv listed the real error, `/card_number pattern`, and then two `unevaluatedProperties` errors that are just echoes. When you show errors to people, ignore the unevaluated ones if another error is present.

## A Realistic Schema: Payments

This schema combines most of the chapter: a required core, a method-dependent part via `if` / `then` (with `required` inside each `if`), `dependentRequired`, and `unevaluatedProperties: false` so unknown fields and fields for the wrong method are rejected:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "required": ["method", "amount"],
  "properties": {
    "method":   { "enum": ["card", "bank", "cash"] },
    "amount":   { "type": "integer", "minimum": 1 },
    "currency": { "enum": ["GBP", "EUR"] }
  },
  "allOf": [
    { "if": { "properties": { "method": { "const": "card" } }, "required": ["method"] },
      "then": { "required": ["card_number", "expiry"],
                "properties": { "card_number": { "type": "string", "pattern": "^[0-9]{16
                                "expiry": { "type": "string", "pattern": "^(0[1-9]|1[0-2
    { "if": { "properties": { "method": { "const": "bank" } }, "required": ["method"] },
      "then": { "required": ["iban"],
                "properties": { "iban": { "type": "string", "pattern": "^[A-Z]{2}[0-9]{2
```

```
    ],
    "dependentRequired": { "expiry": ["card_number"], "currency": ["amount"] },
    "unevaluatedProperties": false
  }
```

I wrote an expected verdict for each of 18 test documents (4 valid, 14 invalid) and ran them in both validators: **18 of 18 behaved as expected in each, and the two agreed on every document**. Examples: a complete card payment was valid; a cash payment with a stray `iban` was invalid (`unevaluatedProperties`); a card with expiry month `13` failed the `expiry` pattern; an amount of `9.99` failed `type` (money as integer cents again); `currency` without `amount` failed both `required` and `dependentRequired`. As a control, I flipped one verdict on purpose and the comparison named exactly that case. The IBAN pattern is only a shape check (two letters, two digits, then 11 to 30 letters or digits): it does not verify the IBAN checksum, which a schema cannot do.

## Error Messages People Can Use

The more flexible the schema, the noisier the errors. I took an event document that should have been a `click` but has `x: "ten"` (a string where an integer is needed), and validated it against three ways of writing "one of these three event shapes":

| SCHEMA STYLE | ERRORS FOR `{"TYPE":"CLICK","X":"TEN","Y":5}` |
|---|---|
| `oneOf` of three event schemas | **6 errors**: the real one (`/x type: must be integer`), then complaints from the other two branches (`required 'key'`, `const "key"`, `required 'dy'`, `const "scroll"`) and a final `oneOf` summary |
| a chain of `if` / `then` keyed on `type` (inside `allOf`) | **2 errors**: `/x type: must be integer` and `if: must match "then" schema`; an unknown type `"hover"` gave one clear `/type enum` error |
| Ajv with `discriminator: true` and a `discriminator: {propertyName: "type"}` keyword | **1 error**: `/x type: must be integer`; `"hover"` gave `value of tag "type" must be in oneOf`; a missing `type` gave `required` and `tag "type" must be string` |

So for documents with a "kind" field, **use `if` / `then` keyed on that field, or Ajv's discriminator option**, instead of a bare `oneOf`: the validator then looks only at the branch that applies. The discriminator keyword is an Ajv option, not part of the JSON Schema specification, so another validator would ignore it and fall back to ordinary `oneOf` behaviour. In Python, the

same `oneOf` produced a single top-level error, `... is not valid under any of the given schemas`, with the per-branch errors tucked into its `context` (five of them); `best_match` returned that top-level error, not the useful one, while the `if`/`then` chain gave just `['x'] type: 'ten' is not of type 'integer'`.

Two ways to make messages friendlier. **Custom messages in the schema**, with the `ajv-errors` plugin: an `errorMessage` keyword turned the results for `{}`, `{qty: 1.5}` and `{qty: 0}` into `quantity is required`, `quantity must be a whole number` and `quantity must be at least 1`. And **a small formatter in your code**; mine for Python turned the order errors into plain lines:

```
def lines(validator, doc):
    out = []
    for e in sorted(validator.iter_errors(doc), key=lambda e: [str(p) for p in e.absolut
        where = '/'.join(str(p) for p in e.absolute_path) or '(whole document)'
        out.append(where + ': ' + e.message)
    return out
```

```
customer: '' should be non-empty
id: 0 is less than the minimum of 1
items/0/price: 0 is less than or equal to the minimum of 0
items/0/qty: 0 is less than the minimum of 1
items/0/sku: 'a1' does not match '^[A-Z][0-9]$'
status: 'lost' is not one of ['pending', 'shipped', 'cancelled']
```

Group errors by path, show the path in the user's vocabulary (a form field name, not `items/0/sku`), and never match on message text: it differs between libraries and versions.

## Schemas in the Editor

VS Code can check a JSON file against a schema as you type. You map files to a schema in `settings.json` with `json.schemas`; the VS Code documentation shows `fileMatch` (a list of patterns, `*` is supported, a leading `!` excludes), a `url` (use a relative path such as `./order.schema.json` for a workspace schema), or an inline `schema` instead of a `url`:

```
{
  "json.schemas": [
    { "fileMatch": ["/orders/*.json"], "url": "./schemas/order.schema.json" }
```

```
    ]
  }
```

A file can also name its own schema with a `"$schema"` property (the docs call this VS Code-specific and warn it changes the file's contents, so other tools may fail validation). The documentation also says VS Code supports drafts 4 to 7 fully and has *limited* support for 2019-09 and 2020-12.

I did not drive the VS Code window. Instead I ran the **JSON language service package that VS Code's JSON support is built on** (`vscode-json-languageservice` 6.0.0-next.4; the JSON language server inside the VS Code 1.138.0 installed here depends on `^6.0.0-next.4`) and asked it for diagnostics. That shows what the editor's engine reports, not how the squiggles look. Results with my order schema:

| DOCUMENT | DIAGNOSTICS |
|---|---|
| a valid order | none |
| id 0, status `"lost"`, sku `"a1"`, qty 0, price 0, extra `note` | six warnings with line numbers, for example `Value is below the minimum of 1.`, `Value is not accepted. Valid values: "pending", "shipped", "cancelled".` and `Property note is not allowed.` |
| `{"id": 1}` | `Missing property "customer"`, and the same for `status`, `date` and `items` |
| date `"2026-13-45"` | **none**: the `format: date` rule was not reported |
| a trailing comma; a comment | severity 1 (an error): `Trailing comma`; `Comments are not permitted in JSON.` |
| 2020-12 keywords: `prefixItems` with `items: false`, `dependentRequired`, `unevaluatedProperties: false` | all three were reported: `Array has too many items according to schema. Expected 2 or fewer.`, `Object is missing property b required by property a.`, `Property zzz is not allowed.` |
| a `$schema` inside the document, for a schema that has `additionalProperties: false` | the good document got `Property $schema is not allowed.`; the bad one also got its real error; a `$schema` pointing at a missing file gave `Unable to load schema from ...` |

Three practical points. Schema problems showed as *warnings* (severity 2) and syntax problems as *errors* (severity 1). The engine understood the 2020-12 keywords I tried, even though the

documentation says "limited", so test your own schema rather than assuming either way. It did not check `format` for the one format I tried. And a schema that closes the object with `additionalProperties: false` will reject the very `$schema` line you add to link it, so for closed schemas map files with `json.schemas` instead.

## Generating Types and Documentation

**Types.** `json-schema-to-typescript` 16.0.0 turned the order schema into:

```
export interface Order {
  id: number;
  customer: string;
  status: "pending" | "shipped" | "cancelled";
  date: string;
  items: Item[];
}
export interface Item {
  sku: string;
  qty: number;
  price: number;
}
```

The `$defs` item became its own interface, the `enum` a union of strings, and the optional or open shapes differed: for the shop schema (no `additionalProperties: false`) each object also got `[k: string]: unknown;`, `phone` became `phone?: string | null`, and `minItems: 1` became the tuple `[Product, ...Product[]]`. I then wrote a TypeScript file with eight objects and compiled it with TypeScript 7.0.2 (`tsc --noEmit`, strict). The compiler **caught** five mistakes: `status: 'lost'`, a missing `items`, an extra `note` field on the closed order, `id: '1'` (a string), and an empty `products` list. It **did not catch** an order with `id: 0`, an empty `customer`, a nonsense `date`, `sku: 'a1'`, `qty: -3` and `price: 0`, because those rules (`minimum`, `pattern`, `format`) cannot be expressed as TypeScript types; nor an extra field on the open shop schema. **Generated types describe the shape; the validator still has to run on real data.**

**Documentation.** `json-schema-for-humans` 1.5.1 (`generate-schema-doc --config template_name=md order.schema.json order.md`) wrote a 5.4 KB Markdown page with a table of contents, one section per property and its restrictions (for `id`: type integer, required, minimum ≥ 1), and "Additional properties: Not allowed" for the closed object. It also marked the whole document's *Required* as "No" (that row is about whether the root itself is required in a larger

document), which is not what a reader might expect. Other generators exist; I tested only these two.

## Changing a Schema Without Breaking People

Once other programs rely on a schema, every change is a compatibility question, and there are two directions: is the **new schema backward compatible** (does it still accept every document that was valid under the old one, so existing data keeps working?) and is it **forward compatible** (does the *old* schema accept documents produced under the new one, so programs that have not upgraded keep working?). I tested eleven kinds of change with sample documents in both directions:

| CHANGE | OLD DOCUMENTS STILL VALID UNDER THE NEW SCHEMA? | NEW DOCUMENTS ACCEPTED BY THE OLD SCHEMA? |
|---|---|---|
| add an *optional* property (open schema) | 2/2 yes | 2/2 yes |
| add an optional property (schema with `additionalProperties: false`) | 2/2 yes | **1/2 no**: old readers reject documents using it |
| add a *required* property | **0/2 no** | 1/1 yes |
| make a required property optional | 2/2 yes | 1/2 no |
| add an enum value | 2/2 yes | 1/2 no (old readers reject the new value) |
| remove an enum value | **1/2 no** | 1/1 yes |
| widen a range (`minimum` 1 → 0) | 2/2 yes | 1/2 no |
| narrow a range (`minimum` 1 → 5) | **1/3 no** | 1/1 yes |
| widen a type (integer → number) | 2/2 yes | 1/2 no |
| narrow a type (number → integer) | **1/2 no** | 2/2 yes |
| rename a property | **0/2 no** | **0/1 no** |

The pattern: **loosening** a schema (optional property, extra enum value, wider range or type) keeps old data valid but can make old readers reject new data; **tightening** it (required property, fewer enum values, narrower range) does the opposite. A rename is both a removal and an

addition, so it breaks both directions. Keep the safe changes (adding optional properties to an *open* schema) for minor versions, put breaking ones behind a new version of the schema (a new `$id`, for example `.../order/v2.json`), and keep a corpus of example documents that every version must still accept. This table used a handful of sample documents per change, so it demonstrates the pattern; it is not a proof for any particular schema.

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. I expected `{}` to be valid against my `if`/`then` address schema.** The `if` passed because it had no `required`, so `then` ran and complained about a missing postcode. **How to avoid it:** put `required` inside the `if`, and test the empty document and a document missing the discriminator field.

**2. My "fresh copy of the schema" benchmark crashed after one document.** Ajv refuses to register the same `$id` twice. That was a result, not a failure of the test, and it is now a row in the table. **How to avoid it:** read a crashing benchmark's message before "fixing" the benchmark; it may be the finding.

**3. I first installed the default `vscode-json-languageservice` (6.0.0-next.3) while VS Code 1.138 depends on `^6.0.0-next.4`. How to avoid it:** read the dependency from the thing you are imitating, and install that version (I reinstalled and re-ran).

**4. Chapter 6 implied that a misspelt `type` is only caught by `check_schema`.** In fact `Draft202012Validator(schema).is_valid` raised `UnknownType` at first use (an unknown *keyword* is still silent). I added that to Chapter 6's table. **How to avoid it:** when a later experiment contradicts an earlier statement, go back and correct the earlier chapter.

## Hands-On Exercises

**EXERCISE 1**

Measure the cost of how you call a validator. In Node with Ajv: compile once and validate 20,000 documents, then `ajv.validate(schemaObject, doc)` with a cached object, then a new Ajv and compile per document (200 documents), then `ajv.validate` with a fresh copy of a schema that has an `$id`. In Python: build one validator and call `is_valid` 5,000 times, then `jsonschema.validate`, then a new validator per call, then `check_schema` alone. Check that every method gives the same number of valid documents, and show `validate()` rejecting a broken schema while `is_valid` on a broken type name raises at first use.

View solution

**EXERCISE 2**

Write the payment schema (card, bank or cash with method-specific required fields and patterns, `dependentRequired`, `unevaluatedProperties: false`). Write 18 test documents with an expected verdict each, run them in Ajv and in Python, report how many behave as expected in each, list any disagreement between the two, and prove the comparison can fail. Then demonstrate `allOf`, `anyOf`, `oneOf` (including the overlapping `integer` / `number` case) and `not`, the `if` trap with `{}`, and `additionalProperties: false` next to `allOf` against `unevaluatedProperties: false`.

View solution

**EXERCISE 3**

Using the order schema: compare the error output of three styles of "one of these event shapes" (`oneOf`, an `if` / `then` chain, Ajv's discriminator) on one wrong document, and show `ajv-errors` custom messages. Run the JSON language service on a valid order, an invalid one, a trailing comma and a 2020-12 schema, and note what it does not flag. Generate TypeScript types, compile eight deliberately varied objects with `tsc --noEmit` and list which mistakes the compiler catches and which it cannot. Generate Markdown documentation. Finally build a table of eleven schema changes and test each in both directions (old documents against the new schema, new documents against the old).

View solution

## CHAPTER 7 QUICK REFERENCE

- Compile once: Ajv compile 39.7 ms once, then 0.40 microseconds per document; a new Ajv per document was 9.7 ms each; the same `$id` twice throws. Python: build one validator, `check_schema` once, use `is_valid` / `iter_errors`; `jsonschema.validate()` was 2.42 ms per call
- `allOf` all, `anyOf` at least one, `oneOf` exactly one (overlap fails: `1` against `[integer, number]`), `not`; for "string or null" use `"type": ["string","null"]`
- `if` / `then` / `else`: put `required` inside the `if` or an empty object takes the `then` branch; `dependentRequired`, `dependentSchemas`
- `additionalProperties: false` cannot see inside `allOf` and rejected even `{"id":1}`; use `unevaluatedProperties: false` (and `unevaluatedItems`); a failing branch also produces echo

`unevaluatedProperties` errors

- Error quality: bare `oneOf` gave 6 errors for one real mistake, an `if`/`then` chain 2, Ajv's `discriminator` option 1; `ajv-errors` adds `errorMessage`; format errors yourself and match on keyword and path, never message text
- VS Code: `json.schemas` with `fileMatch` and `url`/`schema`, or `"$schema"` in the file (VS Code-specific; breaks closed schemas); docs say drafts 4 to 7 fully, 2019-09/2020-12 limited; the language service I ran flagged the 2020-12 keywords I tried but not `format`
- Generated types (json-schema-to-typescript 16.0.0, checked with tsc 7.0.2) catch enum, missing, extra and wrong-type mistakes but not `minimum`, `pattern` or `format`; `json-schema-for-humans` 1.5.1 writes Markdown docs
- Compatibility: loosening (optional property, more enum values, wider range or type) keeps old data valid but old readers may reject new data; tightening (required property, fewer enum values, narrower range) breaks old data; a rename breaks both; version breaking changes under a new `$id`; keep a corpus of documents every version must accept

**COMING NEXT**

Chapter 8 is addressing and changing JSON: JSON Pointer (RFC 6901, the same paths the validators printed), JSONPath (RFC 9535), JSON Patch (RFC 6902) and JSON Merge Patch (RFC 7396), applied with real libraries in Node and Python and with `jq`, and how to choose between them. Start with *Working with JSON 8*.

# Working with JSON

Chapter 8 · Addressing and Changing JSON

Four small standards cover most of what you need to *point at* a part of a JSON document and to *change* it, and they get confused constantly. **JSON Pointer** names exactly one location. **JSONPath** searches for many. **JSON Patch** is a precise list of edits. **JSON Merge Patch** is a much simpler "here is what the document should look like now" edit. You have already met the first one without knowing it: the `instancePath` in a validator's errors (Chapter 6) and the `$ref` paths are JSON Pointers. This chapter explains each one, applies each with real libraries in Node and Python and with `jq` (Chapters 4 and 5), measures where the libraries disagree with the standards, and ends with how to choose.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

Node 24.18.0 with **json-p3 3.0.1**, **fast-json-patch 3.1.1**, **jsonpath-plus 11.2.0** and **json-merge-patch 1.0.2**; Python 3.14.0 with **jsonpointer 3.2.1**, **jsonpatch 1.35**, **python-jsonpath 2.2.1**, **jsonpath-ng 1.10.1** and **json-merge-patch 0.3.0**; **jq 1.8.2**. All installed in a scratch folder. The RFC statements were checked against the RFC texts on rfc-editor.org. **Not run:** other libraries, the official JSONPath compliance test suite (I used 21 queries of my own, below), regular-expression behaviour beyond two simple patterns, and the URI-fragment form of a pointer (taken from RFC 6901).

## The Four Standards at a Glance

| STANDARD | WHAT IT IS | PUBLISHED | LOOKS LIKE |
|---|---|---|---|
| JSON Pointer, RFC 6901 | names **one** location | April 2013 | `/store/book/0/title` |
| JSONPath, RFC 9535 | a query that finds **zero or more** values | February 2024 | `$.store.book[?@.price<10].title` |
| JSON Patch, RFC 6902 | a list of precise edit operations | April 2013 | `[{"op":"replace","path":"/a","value":2}]` |
| JSON Merge Patch, RFC 7396 | a partial document that is merged into the original | October 2014 | `{"a":2,"b":null}` |

JSONPath existed for years as a loose convention with many incompatible versions; RFC 9535 (February 2024) finally standardised it, which matters for the library comparison below. Patch media types are `application/json-patch+json` and `application/merge-patch+json`, used with the HTTP `PATCH` method.

## JSON Pointer (RFC 6901)

A pointer is a string of **reference tokens, each starting with** `/`: `/foo/0` means "the member `foo`, then its element 0". Two characters need escaping inside a token because they have a meaning: `~` is written `~0` and `/` is written `~1`. When decoding, replace `~1` first and then `~0`, otherwise `~01` would be misread as `/`. Three special cases:

- The **empty string** `""` points to the whole document, but `"/"` points to the member whose name is the empty string `""`. They are different.
- Array indexes have **no leading zeros**: `/foo/0` is valid, `/foo/01` is not.
- The token `-` means "the element after the last one". It never points at an existing value, so reading it is an error; it is useful only for *adding* (JSON Patch below).

RFC 6901 (section 5) gives a test document with awkward keys; I checked my copy and the expected values against the RFC text. I wrote it to a file and resolved eighteen pointers (the twelve from the RFC plus six edge cases) with four implementations, one of them a jq function I wrote:

```
{ "foo": ["bar", "baz"], "": 0, "a/b": 1, "c%d": 2, "e^f": 3, "g|h": 4, "i\\j": 5, "k\"l
```

| POINTER (AS A JSON STRING) | RESULT |
|---|---|
| `""` | the whole document |
| `"/foo"`, `"/foo/0"`, `"/foo/1"` | `["bar","baz"]`, `"bar"`, `"baz"` |
| `"/"` | `0` (the empty-string key) |
| `"/a~1b"` | `1` (the key `a/b`) |
| `"/m~0n"` | `8` (the key `m~n`) |
| `"/c%d"`, `"/e^f"`, `"/g\|h"`, `"/ "` | `2`, `3`, `4`, `7` (these characters need no escaping in a pointer) |

| POINTER (AS A JSON STRING) | RESULT |
| --- | --- |
| `"/i\\j"`, `"/k\"l"` | `5`, `6` (backslash and quote need escaping only in the JSON string, not in the pointer) |
| `"/foo/-"`, `"/foo/2"`, `"/nope"`, `"/foo/bar"` | no value |
| `"/foo/01"` | no value (leading zero) |

**json-p3, Python's `jsonpointer` and my jq function matched the RFC on all 18 pointers. `fast-json-patch` matched on 17 and returned `"baz"` for `/foo/01`**, accepting a leading zero that the RFC forbids. In the other "no value" cases the libraries differ in *how* they say so: json-p3 throws a `JSONPointer...Error`, `fast-json-patch` returns `undefined`, Python's `jsonpointer` raises `JsonPointerException` (and for `/foo/-` returns a special `EndOfList` marker object, not an error). Check for "no value" the way your library reports it.

**In jq** a pointer is a path, so `getpath` does the work once the string is converted. My function splits on `/`, unescapes `~1` then `~0`, and treats a digit-only token as an index *only when the value there is an array* (so the key `"0"` in an object still works): `"/foo/0"` gave `"bar"`, and on `{"0":"zero-key","list":["a","b"]}` the pointers `/0` and `/list/0` gave `"zero-key"` and `"a"`. jq itself has no pointer syntax; the reverse conversion (`paths | map(tostring) | join("/")`) is not safe if keys contain `/` or `~`.

Escaping is the usual source of bugs: `/a/b` means two levels; `/a~1b` means the single key `a/b`. A pointer can also be written in a URI fragment (`#/c%25d` for the pointer `/c%d`, percent-encoded as RFC 6901 says; I did not run that form).

## JSONPath (RFC 9535)

A JSONPath query starts at `$` (the root) and chains *segments*. The result is a *nodelist*: zero or more values, each with a known location (RFC 9535 calls it a Normalized Path, written like `$['store']['book'][0]`). An empty result is valid, not an error.

| SYNTAX | MEANING |
| --- | --- |
| `$.name`, `$['name']` | child member (dot or bracket form) |
| `$.*`, `$[*]` | every child |

| SYNTAX | MEANING |
| --- | --- |
| `$..name`, `$..*` | descendant segment: at any depth |
| `[3]`, `[-1]`, `[0,1]` | array index (negative counts from the end), or several |
| `[1:4:2]`, `[::-1]` | slice `start:end:step`, end excluded; a negative step reverses |
| `[?@.price < 10]` | filter: `@` is the current item; the parentheses of older syntax (`?(...)`) are optional |
| `length()`, `count()`, `match()`, `search()`, `value()` | the five standard function extensions (`match` must match the whole string, `search` any part) |

I ran 21 queries on the bookstore document that RFC 9535 and older JSONPath articles use (four books and a bicycle; `store.json`), with the expected answer for each written by hand beforehand, in four libraries:

| LIBRARY | QUERIES WITH THE EXPECTED ANSWER | WHERE IT DIFFERED |
| --- | --- | --- |
| json-p3 3.0.1 (Node, an RFC 9535 implementation) | **21 of 21** | - |
| python-jsonpath 2.2.1 (Python, an RFC 9535 implementation) | **21 of 21** | -; and it gave identical answers to json-p3 on every query |
| jsonpath-plus 11.2.0 (Node, long-established) | 13 of 21 | filters without parentheses (`?@.isbn`, `?@.price<10`) returned nothing; `&&` filters, `length`, `match` and `search` returned nothing; the negative step `[::-1]` returned nothing; comparing two missing values returned nothing |
| jsonpath-ng 1.10.1, `ext` parser (Python, older) | 15 of 21 | `$..*` gave 23 values instead of 27; `&&`, the functions and the missing-value comparison were parse errors |

The older libraries are not "wrong" in their own terms: they predate the RFC and each has its own dialect. But `jsonpath-plus` returning an *empty result* for a filter it does not understand, with no error, is the dangerous kind of difference: the query looks like it ran and found nothing. The same query text can therefore return different data in different libraries. **Pick an RFC 9535**

**implementation when queries cross a language boundary or are stored,** and keep a few known-answer queries as a test. Two details of the RFC the compliant libraries implemented: a comparison of two queries that both select *nothing* is true (`$.store.book[?@.nope == $.store.book[0].nope].price` returned all four prices), and JSON `null` is an ordinary value, not "missing". `match()` and `search()` take I-Regexp patterns (RFC 9485), a deliberately small regular-expression dialect; I tested only `"J.*"` and `"Moby"`.

**jq has no JSONPath**, but it can do the same work with filters. These jq programs gave the same answers as their JSONPath twins on `store.json`:

| JSONPATH | JQ |
|---|---|
| `$..author` | `[.. \| .author? // empty]` |
| `$.store.book[?@.price<10].title` | `[.store.book[] \| select(.price < 10) \| .title]` |
| `$..book[?@.isbn]` | `[.. \| objects \| select(has("isbn")) \| .title]` (titles) |
| `$.store.book[1:4:2].title` | `[.store.book \| to_entries[] \| select(.key % 2 == 1) \| .value.title]` (jq slices have no step) |
| `$.store.book[?length(@.title) > 15].title` | `[.store.book[] \| select(.title \| length > 15) \| .title]` |
| the location of a match (a Normalized Path) | `[path(.. \| select(. == "Moby Dick"))]` gave `[["store","book",2,"title"]]` |

## JSON Patch (RFC 6902)

A patch is an **array of operations** applied in order. Each has an `op` and a `path` (a JSON Pointer):

| OP | NEEDS | WHAT IT DOES |
|---|---|---|
| `add` | `path`, `value` | adds an object member, *replaces* it if it exists, or inserts into an array at an index (`/-` appends); the parent must already exist |
| `remove` | `path` | deletes; the target *must exist* |
| `replace` | `path`, `value` | like remove-then-add; the target must exist |

| `op` | NEEDS | WHAT IT DOES |
|---|---|---|
| `move` | `from`, `path` | remove from one place, add at another; cannot move a value into its own child |
| `copy` | `from`, `path` | add a copy of an existing value |
| `test` | `path`, `value` | fails the whole patch unless the value at `path` equals `value` (numbers compare by value; objects ignore member order) |

RFC 6902 says a patch is applied as a whole: if any operation fails, the patch as a whole is not successful. I took the examples from the RFC's appendix, A.1 to A.16 except A.13 (a duplicate-`op` document that an ordinary parser cannot even represent), which I checked against the RFC text (add, remove, replace, move, test, the `~` escape, appending an array as a single value, unrecognised members ignored, and the error cases), added nine of my own (copy, add past the end of an array, remove or replace a missing member, add over an existing member, move into its own child, test 1 against 1.0, an unknown op, replacing the whole document with an empty path) and ran all **24** in three libraries:

| LIBRARY | CASES MATCHING THE RFC |
|---|---|
| jsonpatch 1.35 (Python) | 24 of 24 |
| json-p3 3.0.1 (Node) | 24 of 24 |
| fast-json-patch 3.1.1 with operation validation turned on (`applyPatch(doc, patch, true)`) | 24 of 24 |
| fast-json-patch 3.1.1 with its **default** `applyPatch(doc, patch)` | **21 of 24**: it *silently succeeded* on "add past the end of an array" (`["a"]` became `["a","x"]`), "remove a missing member" and "replace a missing member" (it created the member) |

The three libraries agreed with each other on every case when configured correctly. The lesson for `fast-json-patch`: **pass `true` as the third argument** (validate operations), or a patch that should have been rejected will go through.

**Atomicity is your job.** I applied a three-operation patch whose third operation fails (`remove /nope`) to `{"a":1}`:

| LIBRARY AND MODE | ORIGINAL DOCUMENT AFTERWARDS |
|---|---|
| fast-json-patch, default (mutates the document) | `{"a":99,"b":2}` — the first two operations had already been applied |
| fast-json-patch with `mutateDocument=false` (`applyPatch(doc, patch, true, false)`) | `{"a":1}` (untouched) |
| json-p3 | `{"a":99,"b":2}` (partly applied) |
| jsonpatch (Python), default `in_place=False` | `{"a": 1}` (untouched) |
| jsonpatch with `in_place=True` | `{"a": 99, "b": 2}` (partly applied) |

So a patch that fails halfway can leave a half-edited document unless you apply it to a *copy* and keep the copy only on success. The `test` operation is also how you do optimistic concurrency ("apply this only if `/version` is still 7"); in the HTTP example below a leading `test` on `/name` let the rest of the patch through.

Patches can also be *generated* by comparing two documents. Python's `jsonpatch.make_patch` turned `{"a":1,"b":[1,2,3],"c":{"d":4}}` and `{"a":2,"b":[1,3],"c":{"d":4,"e":5}}` into `[{"op":"replace","path":"/a","value":2}, {"op":"remove","path":"/b/1"}, {"op":"add","path":"/c/e","value":5}]`, and applying that patch to the first document reproduced the second. jq has no built-in JSON Patch; you express edits directly (`.a = 2 | del(.b[1]) | .c.e = 5`), and Chapter 5's `setpath` and `delpaths` are the primitives.

## JSON Merge Patch (RFC 7396)

A merge patch is just a JSON document shaped like the part of the target you want to change. The rules are short: if the patch is *not an object*, it replaces the target entirely; if it is an object, each member is merged in; a member whose value is `null` **deletes** that member; an array in the patch **replaces** the whole array. I ran the RFC's Appendix A table of 15 examples (checked against the RFC text) plus three of my own (18 cases) in the Node library, the Python library and a jq function I wrote from the RFC's pseudo-code: **all three gave the RFC's result on all 18**. Some of them:

| TARGET | PATCH | RESULT |
|---|---|---|
| `{"a":"b","c":{"d":"e","f":"g"}}` | `{"a":"z","c":{"f":null}}` | `{"a":"z","c":{"d":"e"}}` |
| `{"a":"b"}` | `{"a":null}` | `{}` |
| `{"a":["b"]}` | `{"a":"c"}` | `{"a":"c"}` |
| `{"tags":["x","y"]}` | `{"tags":["x","y","z"]}` | `{"tags":["x","y","z"]}` (the whole array was replaced) |
| `{"a":"b"}` | `["c"]` | `["c"]` (not an object: replaces the target) |
| `{"e":null}` | `{"a":1}` | `{"e":null,"a":1}` (an existing null is left alone) |
| `{}` | `{"a":{"bb":{"ccc":null}}}` | `{"a":{"bb":{}}}` |

Merge patch is pleasant for "update these fields", but it has **built-in limits**: you cannot set a value to `null` (that deletes it), you cannot change one element of an array (you must send the whole array), and a patch generated from two documents loses a `null`: `generate({a:1}, {a:1,f:null})` produced `{"f":null}`, and applying it to `{"a":1}` gave `{"a":1}`, not the intended `{"a":1,"f":null}`. Do not use merge patch where `null` is meaningful data. Both libraries also changed their input object in place unless I copied it first (`apply({a:1}, {a:2})` left the target as `{"a":2}`).

**jq's `*` operator is not merge patch.** I ran four expressions: `{"a":"b"} * {"a":null}` gave `{"a":null}` (RFC 7396 would delete `a`), the nested example gave `{"a":"z","c":{"d":"e","f":null}}` (the RFC's answer deletes `f`), `{"a":["b"]} * {"a":"c"}` gave `{"a":"c"}` (the same), and `+` merged only the top level (`{"a":{"x":1}} + {"a":{"y":2}}` gave `{"a":{"y":2}}`). So `*` is a recursive merge that keeps `null`s, close to merge patch but not equal. If you need real merge-patch semantics in jq, write the pseudo-code as a function (mine passed all 18 cases).

## Patching Over HTTP

The two patch formats are carried by the HTTP `PATCH` method, with the media type telling the server which one you sent. I wrote a small Node server (made for this chapter, so the status

codes are *my* choices, not a standard) that accepts both and applies the patch to a copy, and called it with `curl`:

```
curl -s -X PATCH -H 'Content-Type: application/merge-patch+json' -d '{"email":null,"addr
curl -s -X PATCH -H 'Content-Type: application/json-patch+json' -d '[{"op":"test","path"
curl -s -X PATCH -H 'Content-Type: application/json-patch+json' -d '[{"op":"replace","pa
```

```
start:   {"name":"Ada","email":"ada@example.com","tags":["a","b"],"address":{"city":"Lon

merge patch  -> 200  {"name":"Ada","tags":["a","b","c"],"address":{"city":"London","zip"
json patch   -> 200  {"name":"Ada","tags":["a","b","c","d"],"address":{"city":"Leeds","z
failing patch-> 409  {"error":"patch could not be applied","detail":"OPERATION_PATH_UNRE
state after  ->      {"name":"Ada","tags":["a","b","c","d"],"address":{"city":"Leeds","z
JSON Patch body that is an object, not an array -> 422
Content-Type: application/json                  -> 415 with the list of accepted types
merge patch body that is not JSON               -> 400
```

The failing patch left the stored document unchanged because the server applied it to a copy and replaced the stored document only on success: exactly the atomicity from the previous section. Note that the merge patch changed one field and removed another in one small request, while the JSON Patch needed an operation per edit but could *append* to the array and could check a precondition first.

## Choosing

| YOU WANT TO... | USE |
|---|---|
| name one exact location (in an error, a `$ref`, a patch) | JSON Pointer |
| search inside a document in your own code or in a stored/shared query, and the query may be written by someone else | JSONPath, with an RFC 9535 library; test it on known answers |
| explore a file or an API response from the terminal, or reshape it, or script a pipeline | jq (Chapters 4 and 5): far more powerful than JSONPath, but a program, not a standard query syntax |
| send a precise, ordered, checkable list of edits (append to an array, edit one element, precondition with `test`, move | JSON Patch |

| YOU WANT TO... | USE |
| --- | --- |
| or copy) | |
| send "set these fields, delete those" for plain object data with no meaningful `null` and no partial array edits | JSON Merge Patch |
| say what shape is allowed | JSON Schema (Chapters 6 and 7), not any of these |

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My first pointer test gave a `TypeError` for every pointer in json-p3.** I had called the constructor with a string, but it expects already-split tokens; the function that takes a pointer string is `resolve(pointer, document)`. **How to avoid it:** when a library throws on every input, read its API and print one working call before looping.

**2. My first JSONPath table printed `undefined` and `{}` for json-p3.** Its result is a list of *nodes* (`.value`, `.location`), not a list of values. **How to avoid it:** look at the shape of one result before comparing many.

**3. My jq pointer function treated `"/"` as the whole document.** In jq, `"" | split("/")` is an empty list, so the one-empty-token case vanished; the RFC says `"/"` is the member named `""`. The 18-pointer table caught it. **How to avoid it:** test a hand-written parser on the RFC's own awkward examples, not just on typical input.

**4. I assumed Python's `jsonpointer` would raise for `/foo/-`.** It returns an `EndOfList` marker, and my `json.dumps` of the result is what crashed. **How to avoid it:** print the type of a result, not only its text.

**5. My first `fast-json-patch` run looked perfect (24 of 24).** I had called its `validate` function first, which hid that the default `applyPatch` accepts three invalid patches. Running the default on its own showed it. **How to avoid it:** test a library the way a typical caller uses it, as well as the way its documentation recommends.

## Hands-On Exercises

**EXERCISE 1**

Write the RFC 6901 test document to a file (build the backslash and quote keys without typing escapes by hand). Resolve eighteen pointers (the twelve from the RFC and six edge cases: `/foo/-`, `/foo/01`, `/foo/2`, `/nope`, `/foo/bar` and one more of your choice) with at least three real libraries and a jq function you write yourself. Put the RFC's expected result next to each, print which library differs where, and make sure your jq function treats `""` and `"/"` differently.

View solution

## EXERCISE 2

Using the bookstore document, write 21 JSONPath queries (child, descendant, wildcard, index and negative index, union, slice with step and negative step, filters with and without parentheses, `&&`, `length()`, `match()`, `search()` and a comparison of two missing values), with the expected answer for each written by hand first. Run them in at least two RFC 9535 libraries and two older ones, score each library, list every query where a library differs, and check that the two RFC libraries agree with each other. Then write jq equivalents for six of them and check their output.

View solution

## EXERCISE 3

Run the RFC 6902 appendix examples plus extra cases of your own (24 in all) through `fast-json-patch` (default and with validation), json-p3 and Python's `jsonpatch`, and report which cases each gets wrong. Show what a failing three-operation patch leaves behind in each library's default mode. Run the RFC 7396 examples (18 cases) in a Node library, a Python library and a jq function you write, and show that a generated merge patch loses a `null`. Finally write a small HTTP server that accepts both patch media types, applies them to a copy, and call it with `curl` for a success, a failing patch, a wrong body shape, a wrong media type and bad JSON.

View solution

## CHAPTER 8 QUICK REFERENCE

- Pointer (RFC 6901): `/a/0/b`; escape `~` as `~0` and `/` as `~1` (decode `~1` first); `""` is the whole document, `"/"` is the key `""`; no leading zeros; `-` is "after the last" and never reads a value. `fast-json-patch` accepted `/foo/01`

- JSONPath (RFC 9535, Feb 2024): `$`, `.name`, `*`, `..`, `[3]`, `[-1]`, `[a,b]`, `[start:end:step]`, `[?@.x < 10]`, functions `length count match search value`; results are a nodelist, empty is valid; json-p3 and python-jsonpath scored 21/21, jsonpath-plus 13/21, jsonpath-ng 15/21
- jq has no JSONPath: use `select`, `..`, `paths`, `path(...)`; no slice step
- Patch (RFC 6902): array of `add remove replace move copy test`; `remove`/`replace` need an existing target; `add` over an existing member replaces it; `/-` appends; `test` gives preconditions; media type `application/json-patch+json`
- Libraries: jsonpatch, json-p3 and validated fast-json-patch matched the RFC on 24/24; fast-json-patch's default `applyPatch` accepted 3 invalid patches (pass `true`); apply patches to a **copy**: fast-json-patch (default), json-p3 and jsonpatch with `in_place=True` left a half-applied document after a failure
- Merge patch (RFC 7396): object patch merges, `null` deletes, arrays replace whole, non-object patch replaces the target; cannot set `null`, cannot edit one array element; `application/merge-patch+json`; Node, Python and jq function all matched 18/18; jq's `*` keeps nulls and is not merge patch
- Choose: Pointer for one place; an RFC 9535 JSONPath library for portable search; jq for terminal work; Patch for exact, ordered, checkable edits; Merge Patch for simple field updates without meaningful null; Schema for shape

**COMING NEXT**

Chapter 9 is JSON Lines and large files: the problem with a 2 GB array, JSON Lines and NDJSON (one value per line) with `jq -c`, `-s` and `--stream`, streaming parsers in Node and Python, measured memory and time on generated data, gzip, binary formats such as MessagePack and CBOR, and pagination as the API-side answer. Start with *Working with JSON 9*.

# Working with JSON

Chapter 9 · JSON Lines and Large Files

A JSON document is *one value*. To read the first record of a file that is one huge array, an ordinary parser must read, and build in memory, the whole file, because it cannot know the array is valid until it reaches the final `]`. That is fine for a 10 KB API response and a real problem for a database export. This chapter measures how bad it gets, then shows the fixes in the order you should reach for them: **JSON Lines** (one record per line), **streaming parsers** for arrays you cannot change, **gzip**, the **binary cousins** (MessagePack, CBOR, BSON), and, on the API side, **pagination**. The numbers come from one generated data set and one Windows machine; treat the ratios as the lesson and the absolute figures as examples.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Node 24.18.0 (**stream-json 3.7.0**, **@msgpack/msgpack 3.1.3**, **cbor-x 1.6.6**), Python 3.14.0 (**ijson 3.5.1** with the C backend, **msgpack 1.2.3**, **cbor2 6.1.5**, **pymongo 4.18.3**'s `bson`), **jq 1.8.2** and **gzip 1.13**, on Windows 11, all installed in a scratch folder. The data is 1,000,000 generated records (about 108 MB), made with a fixed random seed. Memory is the *peak working set* of the process, sampled every 5 ms by a small harness, and time is wall-clock for a single run: both are rough. The format descriptions were checked against jsonlines.org, the NDJSON specification and RFC 7464 (read through a fetch tool, so quotations are short). **Not run:** files of gigabytes (the 2 GB figures below are straight-line extrapolation, marked as such), Linux and macOS, other languages, Brotli and zstd, random-access gzip variants such as bgzip, and Protocol Buffers (Chapter 10).

## The Problem, Measured

The test is the simplest realistic job: count the records whose `status` is `"paid"` and add up their `amount`. Each record looks like this (one line of the data set):

```
{"id":1,"name":"Tomás83","status":"shipped","amount":277.58,"tags":[],"ts":"2026-12-13T0
```

The same records were written two ways: as one array (`data.json`, one record per line inside `[ ... ]`) and as JSON Lines (`data.jsonl`). Every program printed the same answer, 250,574 paid

records with a total of 62,748,185.57, so the programs did equal work. Peak memory for the "load everything" approach, at three sizes:

| RECORDS | FILE SIZE | NODE `JSON.PARSE` | PYTHON `JSON.LOAD` |
|---|---|---|---|
| 100,000 | 10.8 MB | 81 MB | 87 MB |
| 300,000 | 32.6 MB | 188 MB | 224 MB |
| 1,000,000 | 108.9 MB | 461 MB | 704 MB |

Memory grows with the file. At a million records Node used about 4 times the file size, Python about 6.5 times, and `jq` loading the same array used 1,172 MB, about 11 times (table below). If that ratio held, which I did not test, a 2 GB file would want somewhere around 9 GB in Node and 13 GB in Python. The same job reading one line at a time stayed at 50-57 MB in Node and 17 MB in Python at *every* size: memory depends on the size of one record, not of the file.

## JSON Lines: One Value per Line

**JSON Lines** (also called **NDJSON**, newline-delimited JSON) puts one complete JSON value on each line, separated by `\n`. Because every line is valid JSON by itself, a program can read a line, process it, throw it away, and move on; a writer can append a line without touching the rest of the file; `head`, `tail`, `wc -l` and `split` work on records; and a damaged line costs you one record, not the file.

| FORMAT | RULES (FROM THE SOURCES I READ) | EXTENSION / MEDIA TYPE |
|---|---|---|
| JSON Lines (jsonlines.org) | UTF-8, no BOM; each line a valid JSON value (any value, not only objects); terminator `\n`, with `\r\n` accepted because surrounding whitespace is ignored; a final terminator "strongly recommended but not required"; a blank line is *not* valid | `.jsonl` (also `.jsonl.gz`); `application/jsonl`, which the site says is "not yet standardized" |
| NDJSON (ndjson-spec) | writers end each text with `\n` (optionally `\r\n`); parsers must accept both; texts must not contain raw newlines; parsers may skip empty lines if that is documented and configurable | `.ndjson`; `application/x-ndjson` |
| JSON text sequences (RFC 7464, Feb 2015, | each text is prefixed with an ASCII Record Separator (0x1E) and followed by `\n`; a parser | `application/json-seq` |

| FORMAT | RULES (FROM THE SOURCES I READ) | EXTENSION / MEDIA TYPE |
| --- | --- | --- |
| Standards Track) | should carry on past a damaged text, and must drop a top-level number, `true`, `false` or `null` that is not followed by whitespace, because it may be truncated | |

In practice JSON Lines and NDJSON are the same thing apart from how they treat blank lines and the final newline, and the same file satisfies both if it has no blank lines and ends with `\n`. The RFC 7464 form is different: the record separator makes it safe to resume reading after damage, which a plain newline cannot promise, and `jq --seq` speaks it. Newlines inside a string value are never a problem: `JSON.stringify` and `json.dumps` write them as `\n` (two characters), so a compactly written record cannot contain a raw line break. Pretty-printed JSON, written with `indent`, is *not* JSON Lines.

## Reading JSON Lines in Python and Node

```
# Python: the file object yields one line at a time
import json, sys
n = s = 0
with open(sys.argv[1], encoding="utf-8") as f:
    for line in f:
        if not line.strip(): continue          # skip blank lines (a policy choice)
        r = json.loads(line)
        if r["status"] == "paid": n += 1; s += r["amount"]
print("paid", n, "sum", "%.2f" % s)
```

```
// Node: readline over a stream, one line at a time
import fs from "node:fs"; import readline from "node:readline";
let n = 0, s = 0;
const rl = readline.createInterface({ input: fs.createReadStream(process.argv[2], "utf8"
for await (const line of rl) {
  if (!line) continue;
  const r = JSON.parse(line);
  if (r.status === "paid") { n++; s += r.amount; }
}
console.log("paid", n, "sum", s.toFixed(2));
```

Both were measured with the million-record file: Node 1.3 s and 57 MB, Python 2.6 s and 17 MB. Both programs are short, but both hide decisions that real files test. I ran each case below:

| INPUT | WHAT HAPPENED |
|---|---|
| no final newline | both records parse |
| CRLF line endings and a blank line | both records parse (the trailing `\r` is whitespace to the parser); the blank line is something *you* must decide about |
| pretty-printed object | every line fails (3 errors in 3 lines): it is not JSON Lines |
| BOM at the start | line 1 fails: Python says "Unexpected UTF-8 BOM (decode using utf-8-sig)", Node says the token is not valid JSON; the format forbids a BOM, so open with `utf-8-sig` only if you must accept files you did not write |
| truncated last line | the earlier lines were good and the last fails: a copy that stopped half way is visible, as long as you do not swallow errors |
| one bad line in the middle | the other lines still parse. Decide per job: stop, or count and skip the bad line and report the count |
| **U+2028 inside a string** | 2 records in the file, but Python's `str.splitlines()` gave 3 pieces and two parse errors, and Node's `readline` gave 3 lines. Splitting on `\n` only, or iterating a Python file object, gave the correct 2 |

The last row is the one that surprises people. JSON allows the raw character U+2028 (LINE SEPARATOR) inside a string, and some line-splitting helpers treat it as a line break, cutting a valid record in two. Python's `for line in f` and a split on `\n` are safe; `splitlines()` is not. In Node I measured `readline` splitting on it, so for data that may contain U+2028 (text typed by users, for example) split on `\n` yourself. I tested one version of each and one character; other separators in the Unicode line-break set were not tried.

## jq and JSON Lines

`jq` already treats its input as a *stream of values* (Chapter 4), so a JSON Lines file is natural input. What matters is how you ask it to read. Measured on the million-record file, for the same "paid" job:

| COMMAND (JOB) | TIME | PEAK MEMORY |
|---|---|---|
| `jq '[.[]\|select(.status=="paid")]\|length' data.json` (whole array) | 6.3 s | 1,172 MB |
| `jq -s '...' data.jsonl` (`-s` slurps every line into one array) | 6.5 s | 1,171 MB |
| `jq -n 'reduce inputs as $r (0; ...)' data.jsonl` | 4.4 s | 6 MB |
| `jq -c 'select(.status=="paid")\|.id' data.jsonl` (one result per input line) | 8.2 s | 6 MB |
| `jq -c --stream -n 'fromstream(1\|truncate_stream(inputs))\|select(...)\|.id' data.json` (array, streamed) | 45.9 s | 6 MB |

`-s` throws away the benefit of JSON Lines: it builds the array in memory exactly as if the file had been one. Use `-n` with `inputs` (or `reduce` / `foreach` over `inputs`) when you need totals, and plain `jq -c` when each line maps to an output line. `--stream` turns a large array into path/leaf events, so memory stays small, but at 46 seconds against 4-8 it is the slow way: my advice is to convert the array to JSON Lines once (next section) and use the fast forms. A few more `jq` facts I checked:

- **Array to lines, and back:** `jq -c '.[]'` and `jq -s -c .` (or `jq -n -c '[inputs]'`). Both need the whole array in memory on one side, so they suit small files.
- **CRLF on Windows:** `jq -c` wrote `\r\n` after each line when I redirected to a file; `jq -b` wrote `\n`. Parsers accept either, but file sizes, checksums and strict tools differ, so use `-b` when you write JSON Lines on Windows.
- **A truncated last line:** `jq` printed the earlier results, then reported "Unfinished JSON term at EOF at line 3, column 6" and exited with status 5.
- `--seq` **changes input as well as output:** it writes RS before and LF after each value, which is RFC 7464 output. But I fed it a plain JSON Lines file and it printed "ignoring parse error: Unfinished abandoned text" and no values, exit status 0: with `--seq`, input must be RS-framed too. A silent exit status 0 is the danger.
- **Piping on this machine was much slower than a file argument:** `jq -n 'reduce inputs ...' data.jsonl` took 4.4 s, while `cat data.jsonl | jq ...` in Git Bash took 13.9 s and `gzip -dc data.jsonl.gz | jq ...` took 18.4 s (the last two ran in Git Bash). I did not find the cause and did not test PowerShell, cmd or Linux; if it matters to you, time both forms on your machine.
- `input_line_number` reports the line a value came from, which is handy for error messages.

## Streaming an Array You Cannot Change

Sometimes the file is a single array and somebody else made it. A **streaming (event) parser** reads tokens and hands you each array element as it completes, without building the whole document.

```
# Python: ijson (the C backend, yajl2_c, was the one in use)
import ijson, sys
n = s = 0
with open(sys.argv[1], "rb") as f:                    # binary mode
    for r in ijson.items(f, "item", use_float=True):   # "item" = each element of a top-
        if r["status"] == "paid": n += 1; s += r["amount"]
```

```
// Node: stream-json 3.x (the API changed between major versions: check its README)
import fs from "node:fs";
import { parser } from "stream-json";
import { streamArray } from "stream-json/streamers/stream-array.js";
import chain from "stream-chain";
let n = 0, s = 0;
const pipeline = chain([fs.createReadStream(process.argv[2]), parser(), streamArray()]);
pipeline.on("data", ({ value: r }) => { if (r.status === "paid") { n++; s += r.amount; }
pipeline.on("end", () => console.log("paid", n, "sum", s.toFixed(2)));
```

Two things to know. `ijson` returns numbers with a fractional part as `Decimal` by default (I printed the type: `Decimal`), exact but slower to add up and not mixable with floats; `use_float=True` gives ordinary floats, as I used. And streaming costs CPU: `stream-json` took 7.8 s against 1.2 s for plain `JSON.parse` on the same file, in exchange for 108 MB of memory instead of 461 MB. Streaming a file that fits comfortably in memory is not a win. Results on the million-record array:

| APPROACH | TIME | PEAK MEMORY |
|---|---|---|
| Node `JSON.parse`, whole array | 1.2 s | 461 MB |
| Node `readline` over JSON Lines | 1.3 s | 57 MB |
| Node `stream-json` over the array | 7.8 s | 108 MB |

| APPROACH | TIME | PEAK MEMORY |
|---|---|---|
| Python `json.load`, whole array | 2.7 s | 704 MB |
| Python line by line over JSON Lines | 2.6 s | 17 MB |
| Python `ijson` over the array | 2.2 s | 20 MB |

Converting once is the practical answer. Streaming the array with `ijson` and writing one compact line per record took 6.0 s and 20 MB, and the resulting file was byte-for-byte the same (same SHA-256) as the JSON Lines file the data generator wrote directly. When you control the producer, write JSON Lines from the start.

## Compression: gzip

JSON is repetitive text (the same keys on every record), and gzip is very good at that. Python's `gzip` module at three levels, on the 107.9 MB JSON Lines file, one run each:

| LEVEL | SIZE | % OF ORIGINAL | COMPRESS | DECOMPRESS |
|---|---|---|---|---|
| 1 | 31,027,868 | 28.8% | 0.27 s | 0.15 s |
| 6 (default) | 18,870,741 | 17.5% | 1.18 s | 0.11 s |
| 9 | 17,537,608 | 16.3% | 2.26 s | 0.12 s |

Compression is cheap to undo and can be read as a stream: Node `fs.createReadStream(...).pipe(zlib.createGunzip())` into `readline` took 1.9 s and 66 MB, and Python `gzip.open(path, "rt", encoding="utf-8")` took 2.9 s and 18 MB, both with the same answer as the uncompressed file. Two gzip members written one after the other decompress as a single stream (I checked), which is why `.jsonl.gz` files can be appended to. The price is that gzip is not seekable: to read the millionth line you decompress the first 999,999. (Formats that add an index, such as bgzip, exist; I did not run them.) For HTTP, the server and browser can compress for you with `Content-Encoding` (HTTP itself is covered in *API Types & Design* 2-4, which I did not re-check for this topic); you do not need to gzip a JSON response by hand.

## Binary Cousins: MessagePack, CBOR, BSON

MessagePack, CBOR and BSON are binary formats with the same data model as JSON (plus extras such as byte strings and, in BSON, dates). I encoded the same million records with each, timed Python's encoder and decoder, compressed each with gzip level 6, and checked that every decoded result equals the original records:

| FORMAT | BYTES | AFTER GZIP -6 | ENCODE | DECODE |
|---|---|---|---|---|
| JSON array, compact | 107,918,861 | 18,870,743 | 1.43 s | 1.78 s |
| JSON array, indent=2 | 164,171,274 | 20,464,364 | 1.65 s | 1.71 s |
| JSON Lines, compact | 107,918,860 | 18,870,741 | 3.92 s | 4.52 s |
| MessagePack | 84,963,287 | 18,335,396 | 0.61 s | 1.09 s |
| CBOR | 84,963,391 | 18,334,512 | 2.30 s | 2.11 s |
| BSON (one document per record, concatenated) | 119,598,105 | 21,341,959 | 1.40 s | 1.14 s |

Three lessons. **Gzip does most of the work.** Raw, MessagePack and CBOR are 21% smaller than compact JSON; after gzip, MessagePack is only 2.8% smaller than gzipped JSON, and BSON is larger than JSON both ways. **Pretty-printing is almost free once compressed** (164 MB raw but 20.5 MB gzipped), so do not minify for size if the transfer is compressed. **Speed differs by library**: the JSON Lines row includes splitting lines in my Python code, so it is not a fair judgment of the format. The binary formats are not human-readable, cannot be edited in a text editor or inspected with `curl` alone, and need a library on every side. BSON has no top-level array (a document is an object), which is why it is stored as one document per record. I also decoded the MessagePack and CBOR files, written by Python, in Node (`@msgpack/msgpack` and `cbor-x`) and the results equalled the JSON for 1,000 records. Not tested: integers beyond 64 bits, dates and byte strings (where these formats and JSON diverge), and encoders in languages other than Python and Node.

## Pagination: The API-Side Answer

A server should not send a million records in one response either. **Pagination** returns a page at a time, and there are two common designs. **Offset**: `?limit=100&offset=200`. **Cursor**: `?limit=100&after=<last id seen>`, with the server returning a `next` value. I wrote a small server of 1,000 items, newest first, and walked it with a client, once with nothing changing and once with 5 new items inserted right after the client fetched the first page:

| WALK | ITEMS RECEIVED | DISTINCT | DUPLICATES |
|---|---|---|---|
| offset, nothing changes | 1,000 | 1,000 | 0 |
| offset, 5 inserted after page 1 | 1,005 | 1,000 | **5** |
| cursor, nothing changes | 1,000 | 1,000 | 0 |
| cursor, 5 inserted after page 1 | 1,000 | 1,000 | 0 |

With offset paging, the insert pushed the last five items of page 1 onto page 2, so the client saw them twice. (The mirror problem, a *delete* making a client skip items, follows from the same arithmetic but I did not run it.) The cursor walk saw each original item once; the five new items were not returned because the cursor was already past them, which is usually what you want for an export and not what you want for a live feed. Cursor paging also avoids the server counting past a large offset, but I did not measure database cost: my server holds an array. The server's design (what `next` is called, whether the cursor is a bare id) is mine; real APIs choose their own and often use an opaque string.

A server can also *stream* JSON Lines in a single response. My `/stream` endpoint sent `Content-Type: application/x-ndjson` and wrote one line every 200 ms; a Node client reading `fetch`'s body stream parsed the first line at 222 ms and the fifth at 1,031 ms, rather than waiting for the end. That is the main reason NDJSON turns up in log and event APIs. The client has to keep any partial line in a buffer until the next newline arrives (the leftover after the last line was 0 bytes), because a network chunk can end in the middle of a record.

## Choosing

| SITUATION | REACH FOR |
|---|---|
| Small file or API response | one JSON document and an ordinary parser: it is simplest |
| Exports, logs, data that grows by appending | JSON Lines, written one record per line, compact, UTF-8, `\n`, no BOM |
| Someone else's huge array | stream it once (ijson, stream-json) and write JSON Lines |
| Totals or filters in the terminal | `jq -n` with `inputs`, or `jq -c`; avoid `-s` and avoid `--stream` unless you must |

| SITUATION | REACH FOR |
| --- | --- |
| Storage or transfer size | gzip first; reach for binary formats only when speed or typed values matter more than readability |
| An API with many records | pagination (cursor for exports); streaming NDJSON for event feeds |

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My first `stream-json` run "finished" in 0.35 s using 39 MB.** That looked like a spectacular result; it was a crash (exit status 1), because I had guessed the module path from an older version. After I fixed the path, a `.pipe()` call failed because version 3 changed the API. The real run took 7.8 s and 108 MB. **How to avoid it:** print the exit status and the result next to every timing, and read the library's own README for its current API.

**2. My first "streaming" conversion reported 444 MB.** The conversion itself was fine; the same script then read both files whole to compare them. After I split the check into a separate script, the conversion measured 20 MB. **How to avoid it:** measure one thing per program, and do the verification somewhere that is not being measured.

**3. The tool I used to write my test files turned my escape sequence for U+2028 into the real character.** The scripts then contained an invisible line separator and could not be read in the solution file. I replaced the characters by building the escape from `chr(92)`. **How to avoid it:** after writing a file that must contain a backslash escape, search it for the raw character.

**4. I assumed `jq --seq` only changed the output.** Fed a plain JSON Lines file, it printed a warning and no values, with exit status 0. **How to avoid it:** test the flag on the exact input you will give it, and check that values came out, not just that the exit status was 0.

**5. I expected `cat file | jq` to cost the same as `jq file`.** On this machine it took three times as long. **How to avoid it:** when a pipeline is slow, time the pieces separately before blaming the program.

## Exercises

**EXERCISE 1**

Generate 1,000,000 records (fixed random seed) as both a JSON array and a JSON Lines file. Write programs that count the records with `status` "paid" and add their `amount` in Node (whole file, `readline`, `stream-json`), Python (`json.load`, line by line, `ijson`) and `jq` (whole array, `-s`, `-n`

with `inputs`, `--stream`). Write a harness that records wall time, peak memory and exit status for each, and run the whole-versus-lines pair at 100,000, 300,000 and 1,000,000 records. Check that every program gives the same answer and say what the harness cannot tell you.

View solution

**EXERCISE 2**

Write JSON Lines readers in Python and Node and test them on: no final newline, CRLF with a blank line, a pretty-printed object, a BOM, a truncated last line, a bad line in the middle, and a record containing U+2028. Report what each reader does and fix any reader that splits a valid record. Then, with `jq`, show the CRLF behaviour with and without `-b`, the output of `--seq`, what `--seq` does with a plain JSON Lines input, and what a truncated last line does to the output and the exit status. Finally convert the million-record array to JSON Lines with `ijson` in constant memory and prove the result identical to the directly generated file with a checksum.

View solution

**EXERCISE 3**

Compress the JSON Lines file with gzip at levels 1, 6 and 9 and report size and time; show that two gzip members back to back decompress as one; read the compressed file as a stream in Node, Python and `jq`. Encode the records as compact JSON, indented JSON, JSON Lines, MessagePack, CBOR and BSON, compress each, check that every one decodes to the original, and decode the MessagePack and CBOR files in Node. Finally write a small HTTP server with an offset endpoint, a cursor endpoint, an insert endpoint and an NDJSON streaming endpoint, and show how many duplicates each paging style gives when items are inserted mid-walk and when each streamed line arrives.

View solution

## CHAPTER 9 QUICK REFERENCE

- One JSON document = one value: parsers load all of it. Measured on 108 MB (1M records): Node 461 MB, Python 704 MB, jq 1,172 MB. Line by line: Node 57 MB, Python 17 MB, `jq -n` + `inputs` 6 MB. Memory grew with file size for whole-file loads and stayed flat for lines
- JSON Lines / NDJSON: one compact JSON value per line, UTF-8, no BOM, `\n` (CRLF tolerated), no blank lines in JSON Lines; `.jsonl` / `.ndjson`; `application/x-ndjson` (NDJSON) and

`application/jsonl` ("not yet standardized"); RFC 7464 `application/json-seq` uses a 0x1E prefix

- Reading: Python `for line in f` and Node `readline`; do **not** use `splitlines()` (U+2028 broke a valid record); Node's `readline` also split on U+2028 in my test; decide what to do with blank lines and bad lines and count them
- jq: `-c` per line; `-n` + `inputs`/`reduce` for totals; avoid `-s` (1,171 MB) and `--stream` (46 s vs 4-8 s); `-b` avoids CRLF on Windows; `--seq` needs RS-framed input; a truncated last line gives exit status 5
- Giant array: stream with `ijson` (`use_float=True`) or `stream-json` (slower: 7.8 s vs 1.2 s), then write JSON Lines; the converted file matched the generated one byte for byte
- gzip -6 shrank JSON Lines to 17.5%; levels 1/6/9 gave 28.8% / 17.5% / 16.3%; streams can be read directly; concatenated members are valid; not seekable
- MessagePack/CBOR: 21% smaller raw, about 3% smaller after gzip; BSON larger than JSON; pretty-printing is nearly free once gzipped (20.5 MB vs 18.9 MB)
- Pagination: offset gave 5 duplicates when 5 items were inserted mid-walk; cursor gave none; NDJSON streams deliver each line as it is sent, if the client buffers partial lines

**COMING NEXT**

Chapter 10 is JSON and its neighbours: JSON against YAML, TOML, JSON5 and JSONC, XML, CSV and Protocol Buffers for configuration, exchange and storage; what each adds or removes (comments, types, schemas, size, speed); converting between them with real tools; YAML's "Norway problem" shown by running a parser; and when JSON is the wrong choice. Start with *Working with JSON 10*.

# Working with JSON

Chapter 10 · JSON and Its Neighbours

JSON is rarely the only choice. Configuration files are often YAML or TOML, data feeds are sometimes XML or CSV, services between companies use Protocol Buffers, and editors accept "JSON with comments". Each format fixes something JSON lacks and gives something up for it. This chapter does not argue from reputation: it feeds the *same* awkward inputs to real parsers and reports what came out, converts one document through each format and counts what was lost, and measures size and speed on 100,000 records. The most useful result is a list of places where a format *quietly changes your data*.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

Python 3.14.0 with **PyYAML 6.0.3** (with libyaml), **ruamel.yaml 0.19.1**, the standard-library **tomllib** with **tomli_w 1.2.0**, **json5 0.17.3**, **xmltodict 1.0.4**, the standard `csv` module and **protobuf 7.36.2** (the `upb` backend) with **grpcio-tools 1.84.0**; Node 24.18.0 with **yaml 2.9.1**, **js-yaml 5.4.3**, **smol-toml 1.9.1**, **json5 2.2.3**, **jsonc-parser 3.3.1** and **fast-xml-parser 5.11.2**. Windows 11, all in a scratch folder. **Not run:** the `yq` command-line tool, Excel's handling of CSV, TOML 1.1, YAML anchors and merge keys, YAML's unsafe loaders, XML namespaces, schemas and XPath, Protocol Buffers schema evolution, and other languages' parsers. A different version of a library may behave differently, especially for YAML, where the behaviour depends on which YAML version a library implements.

## At a Glance

| FORMAT | COMMENTS | `NULL` | TYPED VALUES | BUILT FOR |
|---|---|---|---|---|
| JSON | no | yes | string, number, boolean, null, array, object | data exchange between programs |
| YAML | yes | yes (`null`, `~`, empty) | depends on the loader's rules (see below) | human-edited configuration |
| TOML | yes | **no** | strings, integers, floats, booleans, dates and times, arrays, tables | human-edited configuration |
| JSON5 / JSONC | yes | yes | as JSON (JSON5 adds hex, `Infinity`, `NaN`) | hand-edited JSON files |

| FORMAT | COMMENTS | `NULL` | TYPED VALUES | BUILT FOR |
|---|---|---|---|---|
| XML | yes | no standard way | **none**: all text | documents, and long-lived enterprise interfaces |
| CSV | no | no | **none**: all text | flat tables, spreadsheets |
| Protocol Buffers | in the schema file only | no (absent field) | strict, from a schema | compact, fast, schema-checked messages |

## YAML: The Friendly Format with Sharp Edges

YAML uses indentation instead of braces, allows comments and is easy to read. The catch is that a YAML value with no quotes is *guessed*: the loader decides whether `NO` is a string, a boolean or something else. The guess depends on which version of the YAML specification the library follows (1.1 or 1.2) and on the library. I wrote one small YAML file full of unquoted values and loaded it with four loaders. A selection:

| UNQUOTED VALUE | PYYAML 6.0.3 | RUAMEL.YAML 0.19.1 | NODE `YAML` 2.9.1 | `YAML` WITH `VERSION: "1.1"` |
|---|---|---|---|---|
| `norway: NO` | boolean **false** | string | string | boolean **false** |
| `on_word: on`, `yes_word: yes` | boolean true | string | string | boolean true |
| `quoted_no: "NO"` | string | string | string | string |
| `version: 1.10` | float **1.1** | float 1.1 | number 1.1 | number 1.1 |
| `zip: 01234` | integer **668** (octal) | integer 1234 | number 1234 | number 668 |
| `octal_old: 014` | integer 12 | integer 14 | number 14 | number 12 |
| `octal_new: 0o14` | string | integer 12 | number 12 | string |
| `sexagesimal: 22:22` | integer **1342** | string | string | number 1342 |
| `thousands: 1_000` | integer 1000 | integer 1000 | string | number 1000 |
| `exp: 1e3` | string | float 1000.0 | number 1000 | number 1000 |

| UNQUOTED VALUE | PYYAML 6.0.3 | RUAMEL.YAML 0.19.1 | NODE `YAML` 2.9.1 | `YAML` WITH `VERSION: "1.1"` |
|---|---|---|---|---|
| `date: 2026-10-12` | a `date` object | a `date` object | string | a `Date` object |
| `tilde: ~`, `empty:` | null | null | null | null |

The first row is the famous **Norway problem**: a list of country codes containing `NO` turns Norway into `false` in a YAML 1.1 loader (PyYAML, and Node's `yaml` when told to use 1.1). The other rows are the same disease: `1.10` becomes 1.1 in *every* loader I ran, so a version number must be quoted; `01234` is a zip code in one loader and the number 668 in another; `22:22` is a time of day or the integer 1342. I did not find a case where quoting a value (`"NO"`, `"1.10"`) was read as anything but a string. When PyYAML *writes* a document it does the quoting for you: `yaml.safe_dump` wrote the string `"1.10"` as `'1.10'` and the string `"2026-10-12"` as `'2026-10-12'`, and both came back as strings.

### "YAML is a superset of JSON": Only Sometimes

The YAML 1.2 specification is designed so that a JSON document is also a YAML document. I fed JSON texts to the same loaders to see how true that is in practice:

| JSON INPUT | `JSON.LOADS` / `JSON.PARSE` | PYYAML | RUAMEL.YAML | NODE `YAML` / JS-YAML |
|---|---|---|---|---|
| ordinary object, escaped slash, `u00e9` escape | fine | fine | fine | fine |
| a tab between tokens | fine | **error** (ScannerError) | fine | fine |
| a surrogate-pair escape for an emoji | one emoji | two **lone surrogates** | two lone surrogates | one emoji |
| duplicate key `{"a":1,"a":2}` | last wins: 2 | last wins: 2, silently | **error** | **error** (both) |
| `1E400` | infinity | the *string* "1E400" | infinity | `yaml`: infinity; js-yaml: the string "1E400" |
| `1e3` | 1000.0 | the *string* "1e3" | 1000.0 | 1000 |

A YAML 1.2 loader reads ordinary JSON, but PyYAML (a 1.1 implementation) changed the meaning of `1e3` and broke a valid JSON number into a string, and mangled an emoji. So "just load my JSON with a YAML parser" is only safe with a loader that targets 1.2, and even then the duplicate-key behaviour differs from a JSON parser. (The Node `JSON.parse` line for `1E400` printed `null` because I displayed it through `JSON.stringify`, which prints infinity as `null`.) One more difference from JSON: a plain value containing a colon and a space, `colon_str: a: b`, is an *error* in the loaders I tried (PyYAML: "mapping values are not allowed here"; Node `yaml`: "Nested mappings are not allowed in compact mappings"), so it must be quoted.

> **RULES THAT FOLLOW FROM THE RESULTS**
>
> Quote every string that is not obviously a word. Use a YAML 1.2 loader if you have a choice, and know which one you have. Prefer `safe_load`-style functions (I ran only the safe loaders; the unsafe ones are not covered here). Never use unquoted version numbers, zip codes, country codes, phone numbers or times.

## TOML: Configuration Without Guessing

TOML is aimed at configuration files. Values have explicit types (a quoted string is a string, `2026-10-12` is a date) so there is much less guessing. I parsed a small TOML file in Python and Node:

```
# a comment
title = "demo"
when = 2026-10-12T08:30:00Z
day = 2026-10-12
mixed = [1, "two", 3.0]
hex = 0xFF
big = 1_000
[owner]
name = "Ada"
```

- **Types:** `when` came back as a Python `datetime` with a UTC offset, `day` as a `date`, `hex` as 255 and `big` as 1000. In Node, `smol-toml` returned `TomlDate` objects (a subclass of `Date`) for both.
- **Dates are not JSON:** `json.dumps` on the parsed data raised "Object of type datetime is not JSON serializable". The usual fix, `default=str`, wrote `"2026-10-12 08:30:00+00:00"` with a

space instead of a `T`, which is not the ISO form most JSON consumers expect. Convert dates with `.isoformat()` deliberately.

- **No `null`:** `n = null` is an error in both libraries, and `tomli_w` refused to write a Python `None` ("Object of type 'NoneType' is not TOML serializable"). A JSON document with `null` values cannot go to TOML without deciding what an absent value means.
- **Mistakes are caught:** a duplicate key (`a = 1` then `a = 2`) and redefining `a` as a table both raised `TOMLDecodeError`.
- **`nan` is allowed:** `x = nan` parsed to a NaN float in Python, so a TOML file can contain a value JSON cannot represent.

## JSON5 and JSONC: JSON You Can Comment

**JSON5** is a superset of JSON that allows comments, trailing commas, single-quoted and unquoted keys, hex numbers, `+1`, `.5`, `Infinity` and `NaN`. **JSONC** ("JSON with Comments") is just JSON plus comments and, depending on the tool, trailing commas. I ran thirteen inputs through `JSON.parse`, JSON5 2.2.3 and Microsoft's `jsonc-parser` 3.3.1:

| INPUT | `JSON.PARSE` | JSON5 2.2.3 | JSONC-PARSER |
|---|---|---|---|
| line comment, block comment | error | ok | ok |
| trailing comma | error | ok | error by default, ok with `allowTrailingComma` |
| single quotes, unquoted key, hex `0x1F`, `+1`, `.5`, `Infinity`, `NaN`, multi-line string | error | ok (hex gives 31) | errors |
| `undefined` | error | error | errors |
| ordinary JSON | ok | ok | ok |

The Python `json5` package agreed on the cases I tried (comments, trailing comma, unquoted key, `Infinity`). Two cautions. **Comments do not survive:** parsing a commented file and writing it back with `JSON5.stringify` produced a file with the comment gone, and with unquoted keys and single quotes (`{ port: 8080, tags: [ 'a', ], }`), which is *not* valid JSON. And **a JSON5 or JSONC file is not JSON**: `JSON.parse`, `jq` and most servers reject it, so strip

or convert it before handing it to anything that expects JSON. (I did not test which tools or editors accept comments in which file names; check the documentation of the tool that reads your file.)

## XML: Everything Is Text

XML has attributes, mixed content, namespaces and schemas, and it is still the format of many long-lived interfaces. For data, its limit is that there are no number, boolean or list types: everything is text, and a list is only "an element that repeats". Converting a document to XML and back with `xmltodict` lost types and one list:

| FIELD | BEFORE | AFTER JSON, XML, JSON |
|---|---|---|
| `debug` | boolean `false` | the string `"false"` |
| `port`, `ratio`, each server's `port` | numbers | strings |
| `tags` | `["web"]` (a list of one) | the string `"web"`: **a one-item list is indistinguishable from a single value** |
| `owner.email` | `null` | null again here, but an empty string and `null` both became `None`: they cannot be told apart |

That is 6 differences in a 9-field document. The Node library `fast-xml-parser` showed the same list ambiguity (two `<s>` elements gave an array, one gave a plain string) and, by default, **turned the text `1.10` into the number 1.1 and `01234` into 1234**, the YAML problem again. A node with both an attribute and text came out as `{"@id": "7", "#text": "hi"}` in `xmltodict`; the names `@id` and `#text` are that library's convention, not a standard. Any JSON-to-XML mapping is a set of choices, so two tools will give two different answers.

## CSV: Flat and Untyped

CSV is a table of text. It is the right answer for flat data that goes to a spreadsheet. I wrote one record with awkward values using Python's `csv` module and read it back:

| FIELD | WRITTEN | READ BACK |
|---|---|---|
| `id` 1, `price` 19.99 | numbers | strings `'1'` and `'19.99'` |
| `flag` true | boolean | the string `'True'` (Python's spelling, not JSON's `true`) |
| `tags` a list | a list | the string `"['a', 'b']"` (Python's repr, not JSON) |
| `n` null, `empty` "" | different | both the empty string |
| `zip` "01234", a note with a comma and quotes, a value with a newline | strings | identical (the module quoted them) |

The module handled commas, quotes and embedded newlines correctly by quoting, and kept the leading zero, because it never converts anything. The damage is in types, nulls and nesting: you must write the schema yourself. In my size test below the nested `tags` list went into one cell as compact JSON, and a reader had to convert each column back. (What a *spreadsheet* does to `01234`, long numbers or dates is a separate problem and I did not test it.)

## Protocol Buffers: A Schema and Bytes

Protocol Buffers ("protobuf") describes messages in a `.proto` schema file and encodes them as compact binary; the schema is needed to read the bytes. (Protobuf in a real service is the subject of the course *gRPC*.) I used this schema:

```
syntax = "proto3";
message Record {
  int64 id = 1;
  string name = 2;
  string status = 3;
  double amount = 4;
  repeated string tags = 5;
  string ts = 6;
}
```

For the first record, the JSON line is 100 bytes and the protobuf message is 52 bytes. Three things I observed that do not show in a size table:

- **Default values vanish.** The first record has `id` 0. In proto3 a zero is the default and is not stored: it is absent from the bytes, and the library's own JSON conversion (`MessageToJson`) left `"id"` out entirely. A reader cannot tell "zero" from "not provided".
- **int64 is a string in protobuf's JSON mapping.** `MessageToJson` wrote the id $2^{53}+1$ as `"9007199254740993"` (a string), so JavaScript does not lose the digits (Chapter 1) but your JSON schema must expect a string.
- **The bytes are unreadable without the schema**, and you must keep schema versions in step between sender and receiver. I did not test schema evolution.

## Round Trip: One Document Through Each Format

I took one 9-field configuration document (including a string `"1.10"`, a boolean, a number, a one-item list, a nested object, a list of objects and a `null`), wrote it with each format's own library, read it back and compared it with the original:

| FORMAT (LIBRARY) | RESULT |
|---|---|
| YAML (PyYAML dump, then load) | identical (the dumper quoted `'1.10'` and the date string) |
| TOML (tomli_w, tomllib) | **failed** on the `null`; identical once the null field was removed |
| XML (xmltodict) | 6 differences: booleans and numbers became strings, a one-item list became a string |

The lesson is the same in each row: a format round-trips what it can *represent*. If your data has `null`, TOML cannot hold it; if it has numbers and lists, XML cannot tell you which is which unless something else (a schema) does.

## Size and Speed on 100,000 Records

The first 100,000 records of the Chapter 9 data set (about 10.7 MB as JSON) were written and read back in each format, with each result checked against the original (the same records came back for every row except plain XML, where all values are strings). Single run, one machine:

| FORMAT | BYTES | AFTER GZIP -6 | WRITE | READ |
|---|---|---|---|---|
| JSON array | 10,690,640 | 1,883,172 | 0.11 s | 0.12 s |
| JSON Lines | 10,690,639 | 1,883,169 | 0.28 s | 0.25 s |
| YAML (PyYAML, pure Python) | 10,840,549 | 1,901,699 | 14.43 s | **28.04 s** |
| YAML (PyYAML with libyaml) | 10,840,549 | 1,901,699 | 4.19 s | 13.95 s |
| TOML | 12,690,175 | 1,937,666 | 1.04 s | 2.28 s |
| CSV (tags as a JSON cell) | 6,140,443 | 1,684,128 | 0.28 s | 0.31 s |
| XML | 14,664,873 | 2,001,254 | 1.55 s | 1.27 s with `xmltodict`; 0.18 s just building the element tree with ElementTree |
| Protocol Buffers (length-prefixed) | 6,041,804 | 1,859,122 | 0.16 s | 0.21 s |

What the numbers say. **YAML was about 230 times slower to read than JSON** in the pure-Python loader and about 115 times slower with libyaml, for almost the same number of bytes: fine for a config file, wrong for a data feed. **After gzip, every format landed between 1.68 MB (CSV) and 2.00 MB (XML), within about 11% of JSON's 1.88 MB**, so size is rarely a reason to leave JSON if the transfer is compressed. CSV and Protobuf are smaller raw because they do not repeat the field names on every record. The speed ranking is partly a statement about these *libraries* (the Python JSON parser is written in C; TOML's is pure Python), not only the formats, and the CSV and Protobuf rows include my own conversion code. The XML "Read" column mixes two different jobs, as labelled.

## Choosing and When JSON Is the Wrong Choice

| YOU NEED | REACH FOR |
|---|---|
| Data between programs, web APIs, browser code | JSON |

| YOU NEED | REACH FOR |
|---|---|
| A configuration file people edit by hand, with comments | TOML (flat, typed) or YAML (nested, but quote every string and know your loader); JSONC if your tooling already reads it |
| Hand-edited JSON in a tool that supports it | JSONC or JSON5, converted to JSON before use elsewhere |
| A table for a spreadsheet | CSV, with the types written down somewhere |
| A document with mixed text and markup, or an existing XML interface | XML |
| High volume, strict schema, many languages, small messages | Protocol Buffers (see *gRPC*), or MessagePack/CBOR (Chapter 9) if you have no schema |
| Exact money, 64-bit ids, dates, binary data | JSON works if you represent them as strings (Chapters 1 and 2); a typed format helps if you cannot |

JSON is the wrong choice when you need comments or a human-edited file with no ceremony, when data is naturally a flat table, when you need typed binary data at high volume, or when the other end only speaks something else. It is not the wrong choice just because it is plain: after gzip it was within about 11% of the others on size, and it was the fastest to read in these tests.

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. I wrote a YAML test value, `colon_str: a: b`, and "fixed" it before running it.** I had assumed it would be a parse error and quoted it without checking. Running it afterwards confirmed the error in both loaders (it is now a row of evidence in the chapter). **How to avoid it:** when you think a line is invalid, run it first; a fix applied to an unverified guess is still a guess.

**2. My first Node YAML test crashed at import.** I wrote `import jsyaml from "js-yaml"`, but version 5 has no default export. **How to avoid it:** after a major-version install, check how the package exports its functions before writing a test around it.

**3. A column of my YAML table said `null` for `1E400`, and it was wrong.** The Node `yaml` loader returned infinity; my display went through `JSON.stringify`, which prints infinity as `null`. **How to avoid it:** do not judge a value through a function that changes it; print the type and the raw value.

**4. I expected the first record's `id` to appear in protobuf's JSON output.** The value was 0, so it was omitted as a default. **How to avoid it:** when you compare formats, include values that are zero, empty or false; they are where the formats disagree most.

## Exercises

### EXERCISE 1

Write a YAML file of at least twenty unquoted values (country codes, `yes` / `on`, `1.10`, `01234`, octal and hex forms, `22:22`, `1_000`, `1e3`, dates, `~`, an empty value, `.inf` and `.nan`). Load it with PyYAML, ruamel.yaml, the Node `yaml` package (default and `version: "1.1"`) and js-yaml, and print a table of the type and value each gives. Then feed ten JSON texts (including a tab between tokens, a surrogate-pair escape, a duplicate key, `1E400` and `1e3`) to `json.loads` / `JSON.parse` and the YAML loaders, and report where "YAML is a superset of JSON" fails.

View solution

### EXERCISE 2

Parse a TOML file with a datetime, a date, a mixed-type array, hex and underscored numbers in Python and Node, and show what `json.dumps` does with them (with and without `default=str`). Try a `null`, a duplicate key and `nan` in TOML. Run thirteen inputs through `JSON.parse`, JSON5 and `jsonc-parser` (with and without `allowTrailingComma`) and show what a JSON5 round trip does to comments. Show the CSV and XML losses described in the chapter (a one-item list, the empty string versus `null`, `1.10` and `01234` in `fast-xml-parser`).

View solution

### EXERCISE 3

Take a nine-field configuration document with a `null`, a one-item list and a string `"1.10"`; write and re-read it as YAML, TOML and XML and list every difference. Then write and read 100,000 records as JSON, JSON Lines, YAML (pure Python and libyaml), TOML, CSV, XML and Protocol Buffers (write the `.proto` file, compile it with `grpc_tools.protoc`), checking each round trip, and report bytes, gzip size, write time and read time. Show the first record as a JSON line and as protobuf bytes in hex, and what protobuf's JSON mapping does with a zero `id` and an `id` of $2^{53}+1$.

View solution

### CHAPTER 10 QUICK REFERENCE

- YAML guesses types: `NO` is false in a 1.1 loader (PyYAML), `1.10` became 1.1 in every loader, `01234` was 668 or 1234, `22:22` was 1342 or a string; **quote strings** and know whether your loader is YAML 1.1 or 1.2
- "YAML is a JSON superset" held for 1.2 loaders only on my cases: PyYAML errored on a tab, mangled a surrogate pair, and read `1e3` as a string; duplicate keys error in ruamel, `yaml` and js-yaml but not PyYAML or `json.loads`; `a: b` as a plain value is an error
- TOML: comments, typed dates, no `null`, duplicate keys rejected, `nan` allowed; parsed dates are not JSON-serialisable (`default=str` gives a space, not `T`)
- JSON5 (comments, trailing commas, hex, `NaN`) and JSONC (comments) are not JSON; `JSON5.stringify` drops comments and writes unquoted keys
- XML and CSV have no types: booleans, numbers and one-item lists were lost (6 differences in a 9-field document); empty string and `null` merge; `fast-xml-parser` turned `1.10` into 1.1 and `01234` into 1234
- Protobuf: schema needed; 52 bytes against 100 for the same record; zero values are omitted; int64 is a string in its JSON mapping
- 100,000 records: YAML read 28 s (pure Python) or 14 s (libyaml) against 0.12 s for JSON; after gzip -6 every format was within about 11% of JSON's 1.88 MB (CSV 1.68, XML 2.00)

**COMING NEXT**

Chapter 11 is JSON security and robustness: untrusted input, depth and size limits, duplicate keys and how different parsers disagree, prototype pollution in JavaScript, `eval` and other unsafe loading, injection when JSON is built by string concatenation, JSON in HTML, and what a safe API boundary looks like. Start with *Working with JSON 11*.

# Working with JSON

Chapter 11 · Security and Parsing Pitfalls

JSON cannot run code: it has no functions, no references and no includes. That makes it much safer than many formats, and it also makes people careless. Almost every JSON security problem is really about *what the program does with the data*: it trusts a number it cannot represent, lets two components read the same text differently, merges an attacker's keys into an object, builds JSON out of strings, or accepts a document of any size and depth. This chapter shows each problem happening, then the fix, using the tools from the earlier chapters. All demonstrations are harmless (they print a message or set a variable) and run only against my own scratch files and a server on `127.0.0.1`.

**EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**

Node 24.18.0 (with **secure-json-parse 4.1.1**), Python 3.14.0 (PyYAML 6.0.3), **jq 1.8.2**, **curl 8.12.1** and headless **Chrome 154** on Windows 11, in a scratch folder. The jq security history comes from jq's `NEWS` file as summarised by a fetch tool, so check the file itself before quoting a CVE number. **Not run:** other languages (Java, Go, PHP, C#, Ruby), other JavaScript engines, other browsers, lodash and other merge libraries (I wrote my own merge function), the historical "JSON hijacking" attack (explained, not reproduced), real exploitation of any service, and fuzzing. Limits such as "depth about 16,900" come from one run on one machine and are not constants.

## 1. Numbers Beyond $2^{53}$: Use Strings for IDs

The JSON grammar has no limit on a number's size; the program that reads it does. I fed the same text, `{"id":9007199254740993,"price":0.1}`, to three tools:

| TOOL | RESULT FOR `ID` 9007199254740993 |
|---|---|
| Node `JSON.parse` | **9007199254740992**, and re-serialising wrote that wrong value back |
| Python `json.loads` | 9007199254740993 (exact) |
| jq 1.8.2, value passed through unchanged | 9007199254740993 (the original text is kept) |
| jq 1.8.2, `.id+0` | **9007199254740992** (arithmetic converts to a double) |

| TOOL | RESULT FOR `ID` 9007199254740993 |
| --- | --- |
| jq 1.8.2, `.id\|tostring` | "9007199254740993" (exact) |

Two different records can have ids that differ only in the last digit; after Node reads them they are the same record. For the security angle: an access check on "id 9007199254740993" in one component and a lookup on "id 9007199254740992" in another is a bug an attacker can aim at. **Send identifiers (and money, and anything with more than 15 significant digits) as strings** (Chapters 1 and 2 explain the precision limit and the `rawJSON` alternative).

## 2. Duplicate Keys: Two Readers, Two Answers

The standard says object names "SHOULD be unique" and does not say what happens otherwise. All three tools I ran (Node, Python, jq) take the **last** value: for `{"role":"user","role":"admin","n":1,"n":2}` every one of them gave `role` "admin" and `n` 2. I did not find a first-wins parser among the tools I ran, but other parsers exist that behave differently (I did not test any), and the attack needs only *one* pair of components that disagree: a gateway that checks the first `role` and a backend that obeys the last. To show the shape of the problem I wrote a small first-wins reader myself (labelled as a simulation in the output):

```
DOC = '{"role":"user","role":"admin","tags":[{"a":1,"a":2}]}'

def strict(pairs):                      # reject duplicates instead of choosing
    seen = set()
    for k, _ in pairs:
        if k in seen: raise ValueError("duplicate key: " + repr(k))
        seen.add(k)
    return dict(pairs)

json.loads(DOC)                         # {'role': 'admin', ...}   last wins
json.loads(DOC, object_pairs_hook=strict)  # ValueError
```

The strict hook caught a duplicate nested inside an array (`tags[0].a`) as well. Note that the hook runs from the inside out, so it reported the inner duplicate `'a'` before it reached the outer `role`. Parsing and then re-serialising (`json.dumps(json.loads(doc))`) produces a document with one `role`, which removes the ambiguity for whatever reads it next, but only if

the security check happens *after* that step. Node's `JSON.parse` has no option to reject duplicates: in section 9 I use a small scanner for it.

## 3. Prototype Pollution in JavaScript

Every JavaScript object inherits from `Object.prototype`. If an attacker can set a property on that shared prototype, *every* object in the program gains it. The route is a key named `__proto__` (or `constructor.prototype`) in JSON that a program merges into an object. I tested the pieces separately with the payload `{"name":"x","__proto__":{"isAdmin":true}}`:

| STEP | RESULT |
|---|---|
| `JSON.parse(payload)` alone | **Safe.** `__proto__` becomes an ordinary own property (`Object.keys` shows it); `({}).isAdmin` stays `undefined` |
| my naive recursive `mergeBad(defaults, parsed)` | **Polluted:** afterwards `({}).isAdmin` is `true` for a brand-new object |
| same merge with payload `{"constructor":{"prototype":{"polluted":1}}}` | polluted: `({}).polluted` is 1 |
| my `mergeSafe` (skips `__proto__`, `constructor`, `prototype`, uses `Object.keys` and `Object.hasOwn`) | both payloads: nothing polluted |
| `new Map(Object.entries(parsed))` | keys `a` and `__proto__` are just map keys; nothing affected |
| `Object.assign(Object.create(null), parsed)` | a dictionary with no prototype: `'isAdmin' in dict` is false |
| `secure-json-parse` 4.1.1 | by default throws `SyntaxError: Object contains forbidden prototype property`; with `protoAction: "remove"` returns `{"name":"x"}` |
| `Object.assign({}, parsed)` | no global pollution, but the new object's prototype became the attacker's object: `t1.isAdmin` was `true` |
| `{...parsed}` and `structuredClone(parsed)` | safe: `__proto__` stays an own key, `isAdmin` is `undefined` |

```
// the vulnerable pattern: a recursive merge that trusts every key
function mergeBad(target, src) {
  for (const k in src) {
    if (src[k] && typeof src[k] === "object") { if (!target[k]) target[k] = {}; mergeBad
    else target[k] = src[k];
  }
  return target;
}
// for k = "__proto__", target[k] is Object.prototype itself, so the recursion writes in

// the fix: skip the three dangerous names and only touch own properties
function mergeSafe(target, src) {
  for (const k of Object.keys(src)) {
    if (k === "__proto__" || k === "constructor" || k === "prototype") continue;
    if (src[k] && typeof src[k] === "object" && !Array.isArray(src[k])) {
      if (!Object.hasOwn(target, k) || typeof target[k] !== "object") target[k] = {};
      mergeSafe(target[k], src[k]);
    } else target[k] = src[k];
  }
  return target;
}
```

The lesson is that the parser was not the problem; the merge was. The same is true of "clone", "set by path" and "extend" helpers, and published packages have had exactly this bug. (I wrote my own functions and did not test any library.) Use a `Map` or a null-prototype object for data keyed by user-supplied names, reject or remove the three names at the boundary, and keep untrusted data out of generic merge helpers.

## 4. Never `eval` JSON, and the Other Formats That Run Code

Before `JSON.parse` existed, people turned JSON into data with `eval`. I fed both approaches a document with a call hidden in a value:

| INPUT | `JSON.PARSE` / `JSON.LOADS` | `EVAL` |
|---|---|---|
| `{"a":1,"b":(globalThis.RAN = "code ran", 42)}` in Node | SyntaxError | accepted; `RAN` was set to "code ran" |

| INPUT | `JSON.PARSE` / `JSON.LOADS` | `EVAL` |
|---|---|---|
| `{"a": 1, "b": __import__("builtins").print("...") or 42}` in Python | JSONDecodeError | accepted; the print ran |

Honest JSON works with `eval` too, which is why the habit lasted. A real attacker's value would not print a message. The same class of bug lives in other formats' loaders, and I ran two harmless demonstrations (each makes the loader call `print`):

- **`pickle.loads`** called the function named inside the data (`__reduce__` returned `print` and an argument): "code ran during pickle.loads". Never unpickle data you did not make.
- **PyYAML:** `yaml.load(..., Loader=yaml.UnsafeLoader)` on the text `!!python/object/apply:builtins.print ["..."]` called `print`. `yaml.safe_load` refused the same text with a `ConstructorError`. The same text inside a JSON *string* is only data.

So the safe habit is: JSON goes through `JSON.parse` or `json.loads`, YAML through `safe_load`, and nothing that can instantiate arbitrary types through untrusted input. JSON's lack of features is the protection; keep it that way.

## 5. Depth and Size: Denial of Service

A parser needs memory for what it reads, and some code recurses once per nesting level. Chapter 9 measured the memory side; here is the depth side. I generated nested arrays (`[[[[...]]]]`) and tried each operation:

| OPERATION | RESULT |
|---|---|
| Node `JSON.parse` | depth 5,000,000 parsed fine (about 1 second) |
| Node `JSON.stringify` / `structuredClone` on a nested object | failed with `RangeError` (call stack) above about depth 4,767 / 1,831 |
| a depth-100,000 document parsed by Node, then `JSON.stringify` or `structuredClone` on the result | both `RangeError`: the parse succeeded, and the *next* step crashed |
| Python `json.loads` (recursion limit 1000) | fine to about depth 16,900; `RecursionError` at 100,000 |
| Python `json.dumps` | fine to about depth 15,500 |

| OPERATION | RESULT |
| --- | --- |
| Python `copy.deepcopy` of a depth-5,000 parse result | `RecursionError` |
| jq 1.8.2 parsing | depth 10,000 fine; 10,001: "parse error: Exceeds depth limit for parsing" |

The "about" figures came from a single binary search each and move with the stack size and the program's state, so treat them as orders of magnitude. The point is that different layers fail at different depths, and the failures are exceptions, not security holes, but an uncaught one in a request handler is an easy way to knock a service over. Cap the **body size** before reading it all, cap the **depth** (64 is generous for almost any API), and catch the errors.

jq has had its own memory-safety fixes. As summarised from jq's `NEWS`: 1.8.0 fixed CVE-2024-23337, a signed integer overflow in array and object internals, by limiting them to $2^{29}$ elements; 1.8.2 fixed a number of issues including CVE-2026-40164 (crafted keys that collide in the hash table, causing high CPU use from about 100 KB of JSON; it now randomises the hash seed) and capped recursion depth in several functions. On 1.8.2 I tried `jq -n '[] | .[999999999999] = 1'` and got "Array index too large" instead of an allocation. If you process untrusted JSON with `jq`, run a current version (`jq --version`).

## 6. Injection: Never Build JSON with String Concatenation

```
def bad2(name):  return '{"role":"user","user":"' + name + '"}'
def good(name):  return json.dumps({"role": "user", "user": name})

attack = 'bob","role":"admin'
json.loads(bad2(attack))     # {'role': 'admin', 'user': 'bob'}      injected a second "
json.loads(good(attack))     # {'role': 'user', 'user': 'bob","role":"admin'}   just a s
```

The name contained a quote, a comma and a new key, and with string concatenation it became JSON structure. Two things I observed that are easy to miss. First, the attack only worked when the injected key came **after** the real one: in the template that put the name first (`{"user":"NAME","role":"user"}`) the later real `role` overwrote the injected one, because parsers take the last duplicate (section 2). So the order of fields decided whether the attack worked, which is no defence. Second, the same concatenation breaks on innocent input: a name

with a plain double quote made the parser fail ("Expecting ',' delimiter"), and a name containing a backslash followed by `t` (`C:\temp`) was silently turned into a tab. `JSON.stringify` and `json.dumps` escape correctly; use them for every value.

### JSON inside HTML

A common pattern puts JSON in a `<script>` block. `JSON.stringify` does *not* escape `<`, so a value containing `</script>` ends the script block early and the rest is read as HTML:

```
const data = { comment: "</script><script>alert(1)</script>" };
"<script>const data = " + JSON.stringify(data) + ";</script>"        // the string conta

const safe = JSON.stringify(data).replace(/</g, "\u003c").replace(/>/g, "\u003e").replac
// "comment":"\u003c/script\u003e..."  still valid JSON, and it parses back to the same
```

The escaped form contains no raw `<`, so the browser cannot find a closing tag in it. (I checked the strings and the round trip, not the browser rendering of this particular page. The next section does run a browser.) Since ES2019 a raw U+2028 inside a JSON string is also legal JavaScript, so it no longer needs special handling in a script block; I tested that in Node only.

### `Content-Type`: a Browser Does What the Header Says

I wrote a tiny server with two routes that echo a query parameter inside a JSON object, `{"q":"..."}`. One labels the response `text/html`, the other `application/json; charset=utf-8` with `X-Content-Type-Options: nosniff`. I opened both in headless Chrome 154 with `q=<script>document.title='RAN'</script>` and dumped the page:

| ROUTE | WHAT CHROME MADE OF IT |
|---|---|
| `text/html` | the script **ran**: the page title became "RAN" (reflected XSS from a JSON body) |
| `application/json` with `nosniff` | shown as text: the dump contained `<script>` inside a `<pre>`; nothing ran |

So: always send JSON as `application/json`, and add `X-Content-Type-Options: nosniff`. (I did not test what Chrome does with `application/json` *without* nosniff, or any other browser.) The old "**JSON hijacking**" attack is a related historical case: a site returned a top-level array of private data, and a hostile page loaded it with a `<script src>` tag and overrode the browser's

array constructor to read it. That relied on old browser behaviour; I did not reproduce it. Modern guidance is the same as above (correct content type, authentication that does not rely on cookies alone, and an object at the top level rather than a bare array), but check current OWASP guidance for your situation.

## 7. Sensitive Data in Logs

JSON is easy to log whole, and that puts passwords and tokens in log files. Redaction by key name is simple (a replacer function in JavaScript, `walk` in jq) and I ran both. It also has limits, which the test exposed:

```
const SENSITIVE = /^(password|passwd|secret|token|access_token|refresh_token|authorizati
const redact = (v) => JSON.parse(JSON.stringify(v, (k, val) => (SENSITIVE.test(k) ? "[RE
```

- It removed `password`, `profile.ssn`, `items[0].token` and the `Authorization` header at any depth.
- It **missed** a free-text field (`"note": "my password is hunter2"`) and a key I had not listed (`"pw"`): both still contained the secret. A list of names protects only the names on the list; an allow-list (log only the fields you choose) is safer.
- Error messages can leak too: Node's parser error for the text `password=hunter2` was `Unexpected token 'p', "password=hunter2" is not valid JSON`, which quotes the input. Do not log a parse error together with the body that caused it.

## 8. A Front Gate for Untrusted JSON

Putting the limits together, here is a small gate I wrote for Node: check the content type, the byte size, then scan the text once for depth and duplicate keys, then call `JSON.parse`. The scanner ignores braces inside strings, handles escaped quotes, and decodes keys before comparing them (so `"a"` and `"\u0061"` count as the same key).

```
export function safeParse(text, contentType = "application/json") {
  if (!/^application\/(.+\+)?json(\s*;.*)?$/i.test(contentType)) throw new Error("unsupp
  if (Buffer.byteLength(text) > 1_000_000) throw new Error("body too large (limit 100000
  scan(text);              // throws on depth > 64 or a duplicate key in any one object
```

```
    return JSON.parse(text);
  }
```

| INPUT | GATE |
| --- | --- |
| ordinary document; same key in two different objects; braces and an escaped quote inside strings | accepted |
| `{"role":"user","role":"admin"}`; a nested duplicate; `"a"` versus its escaped form | rejected: duplicate key |
| depth 64 / depth 65 / depth 100,000 | accepted / rejected / rejected |
| a 1,000,001-byte string | rejected: too large |
| content types `application/json`, with charset, `application/vnd.api+json` / `text/html`, `text/plain` | accepted / rejected |

Behind a small HTTP server the same gate returned 200 for good JSON, 400 for a duplicate key, for depth 100 and for malformed JSON, 415 for `text/plain`, and 413 for a 2 MB body (the status codes are my choices). The gate is deliberately small: it does not validate the *shape* (use JSON Schema, Chapters 6 and 7, after the gate), it does not stop a flood of requests, and it is a scanner I wrote, not a hardened library: I tested the cases in the table and no others. In production, also put a size limit in the web server so an oversized body never reaches your code.

## Checklist

| RISK | DEFENCE (EACH SHOWN ABOVE) |
| --- | --- |
| big numbers lose digits | ids, money and long numbers as strings |
| duplicate keys parsed differently | reject them at the boundary, or re-serialise before checking |
| prototype pollution | no generic merge of untrusted data; skip `__proto__` / `constructor` / `prototype`; `Map` or null-prototype objects; `secure-json-parse` |
| code execution | only `JSON.parse` / `json.loads`; `safe_load` for YAML; never `eval`, `pickle` or unsafe loaders on untrusted data |

| RISK | DEFENCE (EACH SHOWN ABOVE) |
|---|---|
| depth and size | cap bytes before reading, cap depth, catch errors, keep jq current |
| injection | build JSON only with `JSON.stringify` / `json.dumps`; escape `<` in script blocks |
| content type | `application/json` plus `nosniff`; object at the top level |
| logs | allow-list fields; do not log bodies with parse errors |

**MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My first injection demo failed, and I had expected it to work.** The injected `"role":"admin"` came before the real `"role":"user"`, and last-wins parsing kept the real one. Only when I put the attacker's field last did it succeed. **How to avoid it:** when an attack demo does nothing, find out why before changing it; here the reason (duplicate-key order) became part of the lesson.

**2. Headless Chrome started from Git Bash hung** past two and a half minutes without output. I stopped it and ran Chrome from PowerShell with a timeout that kills the process, which worked first time. **How to avoid it:** give every browser launch a hard timeout, and clean up the profile processes it leaves.

**3. I guessed a security-advisory URL for the jq CVE and the page was about a different CVE.** The fetch tool said so and I did not use it as evidence for CVE-2024-23337; I took the facts from jq's `NEWS` file instead. **How to avoid it:** do not guess identifiers in URLs; start from the project's own changelog.

**4. My first log-redaction example said a parse error "can quote your input" but the error I triggered did not.** It was a trailing-comma error that quotes only a position. I changed the input to `password=hunter2`, which does quote it. **How to avoid it:** check the claim against the exact case you print, not a similar one.

**5. I expected `JSON.parse` to fail at some depth.** It parsed 5 million levels; the failures came later, in `JSON.stringify`, `structuredClone` and Python's `deepcopy`. **How to avoid it:** test the whole path the data takes, not only the parse.

## Exercises

**EXERCISE 1**

Show the number problem in Node, Python and jq (including the effect of `.id+0` and `tostring`). Show how Node, Python and jq treat duplicate keys, write a strict `object_pairs_hook` and a simulated first-wins reader, and show how they disagree. Then build JSON by string concatenation in Python and Node with a name that adds a second `role`: show it working with the attacker's field after the real one, failing with it before, and breaking on a plain quote and on a backslash, and compare with `json.dumps` / `JSON.stringify`. Show the `</script>` problem and the escaped form.

View solution

### EXERCISE 2

In Node, show that `JSON.parse` of a `__proto__` payload is harmless, write a vulnerable recursive merge and show it polluting `Object.prototype` through both `__proto__` and `constructor.prototype` (remove the pollution afterwards), then fix it and test the fix. Try a `Map`, a null-prototype object, `secure-json-parse`, `Object.assign`, spread and `structuredClone` and report each. Then show `eval` running code that `JSON.parse` rejects, in Node and Python, and (with a harmless `print`) a pickle and an unsafe YAML load running code that `safe_load` refuses.

View solution

### EXERCISE 3

Measure depth limits for parse, stringify, clone and copy in Node, Python and jq, including a parse that succeeds followed by an operation that fails, and try jq's huge-index assignment. Write the front gate (size, depth, duplicate keys, content type), test it on twelve inputs, put it behind a small HTTP server and call it with `curl` for each status code. Serve the same JSON with `text/html` and with `application/json` plus `nosniff` and open both in headless Chrome with a script in the data. Write a redaction function and show what it misses.

View solution

## CHAPTER 11 QUICK REFERENCE

- Ids as strings: Node read 9007199254740993 as 9007199254740992; jq `.id+0` does too, while Python and a jq pass-through keep it
- Duplicate keys: Node, Python and jq all keep the **last**; the standard leaves it open, so reject duplicates at the boundary (Python `object_pairs_hook`, a scanner in Node) or re-serialise before checking

- Prototype pollution comes from *merging* untrusted keys, not from `JSON.parse`; skip `__proto__`, `constructor`, `prototype`; use `Map` or `Object.create(null)`; `secure-json-parse` rejects them; `Object.assign` changed the new object's prototype
- Never `eval` JSON; `pickle.loads` and `yaml.load` with an unsafe loader ran code, `yaml.safe_load` refused
- Limits: Node parse took depth 5,000,000 but `stringify` failed about 4,767 and `structuredClone` about 1,831; Python `loads` about 16,900; jq stops at 10,001; cap body size and depth and catch the errors; keep jq current (1.8.0 limited arrays to $2^{29}$ elements)
- Build JSON only with `JSON.stringify` / `json.dumps`; concatenation let a name add a `role` key (the attacker's field must come last to win); escape `<` as `\u003c` inside script blocks
- Send `application/json` plus `X-Content-Type-Options: nosniff`: Chrome ran a script from a `text/html` response and showed the `application/json` one as text
- Logs: redact by allow-list; key-name redaction missed free text and an unlisted key; parse errors can quote the input

**COMING NEXT**

Chapter 12 is the capstone: a JSON data pipeline. You take a real public JSON dump, validate it with a JSON Schema, clean and reshape it with `jq`, stream the result as JSON Lines with a Node or Python script, patch one record with JSON Patch, report counts and write a short README, using what Chapters 1 to 11 built. Start with *Working with JSON 12*.

# Working with JSON

Chapter 12 · Capstone: A JSON Data Pipeline

This chapter uses everything in the course on one real job. You download a public JSON feed, check it against a JSON Schema, clean it into JSON Lines with `jq`, stream through it with a script to produce a report, check that report with a second tool, change one record with a JSON Patch, and prove you can run the whole thing again and get the same files. The data is the USGS earthquake feed for the past 30 days. Unlike the earlier chapters, the data is not mine or generated: it has real surprises in it, and the pipeline found one that I did not expect.

> **EVERY EXAMPLE HERE WAS RUN, EXCEPT WHERE MARKED**
>
> Node 24.18.0 (**Ajv 8.20.0**, **ajv-formats 3.0.1**), Python 3.14.0 (**jsonschema 4.26.0**, **jsonpatch 1.35**), **jq 1.8.2**, **curl 8.12.1**, Windows 11. All counts below are from one snapshot taken on 2026-10-12 and will not match a download you make later. **Not run:** a second download, any other feed, Linux or macOS, other validators or patch libraries, and a larger window such as a year of data. The licence statement below comes from USGS policy pages found by a web search, not from a page written for this one feed; see the Licence section.

## The Plan

| STEP | WHAT | TOOL | CHAPTERS USED |
|---|---|---|---|
| 1 | Fetch one snapshot; record size, hash, clock, count | curl, jq | 4, 11 |
| 2 | Validate every feature against a schema; cross-check with a second validator; run a control | Ajv, jsonschema | 6, 7 |
| 3 | Keep earthquakes, flatten to one record per line | jq | 4, 5, 9 |
| 4 | Stream the lines, validate each record, count, write a report | Node | 2, 9 |
| 5 | Recompute the numbers a different way and compare | jq | 5 |
| 6 | Apply a JSON Patch to one record, with checks, without touching the original | jsonpatch, jsonschema | 3, 7, 8 |
| 7 | Run steps 2 to 6 twice and compare checksums | sha256 | 9, 11 |

## Step 1: The Data and Its Licence

The USGS publishes earthquake summaries as GeoJSON (a JSON format for map data), refreshed every minute, at `https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_month.geojson`. The page that documents the feed says each feature has an `id`, a `geometry` (a Point with longitude, latitude and depth) and `properties` such as magnitude, place, time and status. I made **one** request, with a descriptive `User-Agent`, and kept the file:

| ITEM | VALUE |
|---|---|
| Response | HTTP 200, `Content-Type: application/json; charset=utf-8`, `X-Content-Type-Options: nosniff` (Chapter 11), `Cache-Control: public, max-age=60` |
| Size | 7,173,808 bytes |
| SHA-256 | `37f31144591d663c...` (kept in full in the README) |
| Feed clock | generated 2026-10-12T19:05:41Z; `metadata.count` 10,105, and the file holds 10,105 features (checked) |

**Licence.** A capstone should record where data came from and what you may do with it. The USGS pages I could read through a search say that most of its information resides in the public domain and may be used without restriction, with a request for credit (the suggested form is a line such as "Source: U.S. Geological Survey"), and that some third-party material on its sites is copyrighted. A direct fetch of the main USGS copyright page failed (HTTP 403), so I did not read that page myself, and the README says so. **Before you reuse the data, read the current USGS terms.** The snapshot is not stored with the course; the README records how to fetch another.

## Step 2: Validate Before You Trust

A feed you did not write can change without warning. The schema describes only the parts this pipeline depends on, and stays open to new fields, because a feed that adds a property should not break us. An excerpt (Chapters 6 and 7 explain each keyword):

```
  "geometry": {
    "type": "object",
    "required": ["type", "coordinates"],
```

```
    "properties": {
      "type": { "const": "Point" },
      "coordinates": {
        "type": "array",
        "prefixItems": [
          { "type": "number", "minimum": -180, "maximum": 180 },   // longitude
          { "type": "number", "minimum": -90,  "maximum": 90 },    // latitude
          { "type": "number" }                                     // depth in km (negativ
        ],
        "items": false,
        "minItems": 3
      }
    }
  },
  "properties": {
    "properties": {
      "mag":    { "type": "number", "minimum": -2, "maximum": 10 },
      "place":  { "type": "string", "minLength": 1 },
      "time":   { "type": "integer", "minimum": 0 },
      "status": { "enum": ["automatic", "reviewed"] },
      "tsunami": { "enum": [0, 1] },
      "url":    { "type": "string", "format": "uri" }
    }
  }
```

I compiled the schema once with Ajv (strict mode on, `ajv-formats` for `format: "uri"`) and checked each feature separately so every bad record can be named. Results on the snapshot:

| CHECK | RESULT |
| --- | --- |
| Ajv, per feature | 10,105 features: **10,104 valid, 1 invalid**; the one error is "/properties/place must be string" on `aka2026snusmn` (a null place) |
| Python `jsonschema`, same schema | 1 invalid, the same id: the two validators agree. (`jsonschema` ignores `format` unless given a format checker, a Chapter 6 finding) |
| `metadata.count` equals the number of features | true; a schema cannot express this, so the script checks it |

| CHECK | RESULT |
| --- | --- |
| Control: eight copies of one real feature, six deliberately broken (mag as a string, latitude 95, two coordinates, an unknown status, a float timestamp, a non-URI url) | the untouched copy and the copy with an extra property passed; all six broken copies were rejected |

The control matters: a validator that says "valid" to everything would also have produced "10,104 valid". Seeing it reject known-bad copies is what makes the one real failure believable.

## Step 3: Clean and Reshape with jq

The raw feature nests the useful fields two levels deep, stores times as milliseconds, and mixes real earthquakes with explosions and quarry blasts. The `jq` program (Chapter 5) turns each earthquake into one flat record:

```
def iso: (. / 1000 | floor | todate);
def region: if contains(", ") then (split(", ") | last) else . end;
.features[]
| select(.properties.type == "earthquake")
| .properties as $p
| ($p.place // "(unknown)") as $place
| { id: .id, time: ($p.time | iso), updated: ($p.updated | iso),
    mag: $p.mag, mag_type: $p.magType, place: $place, region: ($place | region),
    lon: .geometry.coordinates[0], lat: .geometry.coordinates[1], depth_km: .geometry.co
    status: $p.status, tsunami: ($p.tsunami == 1), sig: $p.sig, net: $p.net, url: $p.url
```

I ran it as `jq -b -c -f 03_clean.jq quakes.geojson > quakes.jsonl`. `-c` gives one record per line and `-b` keeps Windows from writing CRLF (Chapter 9). Checks on the output:

- **9,891 records written; 214 features dropped** by the type filter (72 explosions, 117 quarry blasts, 22 ice quakes, 2 mining explosions, 1 landslide). 9,891 + 214 = 10,105, so nothing went missing.
- The file contains no carriage-return bytes (0, counted with `tr`, after I had to fix my first check; see the mistakes).
- **The unexpected result:** the one feature with a null place is a *quarry blast*, so the type filter had already dropped it, and my "(unknown)" rule for null places never fired (0 records). I had written the rule to handle the invalid record Step 2 found; on this snapshot it is dead code. It

would matter for a feed with a null place on an earthquake, so I kept it and wrote that down rather than claim it was exercised.

- **The `region` rule is crude.** It takes the text after the last comma of `place`. USGS writes "CA" for California but "Alaska" in full, so the biggest regions are "Alaska" (3,323 records) and "CA" (2,828), and 137 places with no comma (for example "Mid-Indian Ridge") keep their whole text. A real analysis would map abbreviations to names.

## Step 4: Stream the Lines and Make the Report

The cleaned file is JSON Lines, so a Node script reads it one line at a time (Chapter 9), validates each record against a second, *closed* schema (`additionalProperties: false`: this is our own output, so an unknown field is a bug), and tallies. Excerpt:

```
const rl = readline.createInterface({ input: fs.createReadStream(input, "utf8"), crlfDel
for await (const line of rl) {
  report.lines++;
  if (!line.trim()) { report.blank_lines++; continue; }
  let r; try { r = JSON.parse(line); } catch { report.bad_json_lines++; continue; }   //
  if (!check(r)) { report.schema_invalid_records++; continue; }
  report.valid_records++;
  report.by_magnitude[bucket(r.mag)]++;
  if (r.mag >= 5) strong.write(line + "\n");                                    // se
  // ... status, tsunami, depth range, regions, days, strongest ...
}
```

| REPORT ITEM | VALUE ON THIS SNAPSHOT |
|---|---|
| Lines / valid / invalid / bad JSON / blank | 9,891 / 9,891 / 0 / 0 / 0; no duplicate ids |
| Magnitude: under 2 / 2-3 / 3-4 / 4-5 / 5-6 / 6+ | 6,898 / 1,713 / 303 / 808 / 161 / 8 (the 4-5 count is larger than the 3-4 count; I did not investigate why) |
| Status | 1,799 automatic, 8,092 reviewed |
| Tsunami flag set | 5 |
| Depth range | -3.43 km to 665.057 km |

| REPORT ITEM | VALUE ON THIS SNAPSHOT |
|---|---|
| Strongest | M 7.7, 12 km WSW of Pitaloza Arriba, Panama, 2026-10-09T17:56:06Z |
| `strong.jsonl` (mag 5 and over) | 169 lines = 161 + 8 |
| Resource use (Chapter 9 harness) | about 0.4 s, 58 MB peak memory |

The script also checks its own arithmetic: valid + invalid + bad + blank equals the number of lines, and the magnitude buckets sum to the valid count (both true).

## Step 5: Check the Numbers a Different Way

A report is only as good as its method, so I recomputed the key numbers with `jq` straight from the *original* GeoJSON, not from `quakes.jsonl`, and compared them with `report.json`: number of earthquakes, the six magnitude buckets, status counts, the tsunami count and the strongest event's id. They agree. As a control I changed one number in the jq output on purpose and the comparison reported a disagreement. Two methods, two inputs, same answer: the cleaning step did not lose or alter any of the things counted.

## Step 6: Patch One Record, Safely

Suppose a reviewer confirms record `aka2026ufzwkj`, so it should become `reviewed` and its magnitude 1.5. This is a teaching edit on a copy, not a real seismic correction. A JSON Patch (Chapter 8) states it exactly, with preconditions:

```
[
  { "op": "test",    "path": "/id",     "value": "aka2026ufzwkj" },
  { "op": "test",    "path": "/status", "value": "automatic" },
  { "op": "replace", "path": "/status", "value": "reviewed" },
  { "op": "replace", "path": "/mag",    "value": 1.5 }
]
```

The script streams the JSON Lines file, applies the patch to the one record whose id matches (the library applies it to a copy), checks the *patched* record against the closed record schema, and only then writes the new file; the original is never opened for writing. Three patches, three outcomes:

| PATCH | RESULT |
| --- | --- |
| the good patch above | applied; new file written; **exactly one line differs** from the original (9,891 lines in both); status counts moved from 1,799 automatic and 8,092 reviewed to 1,798 and 8,093 |
| a `test` that expects `reviewed` on a record that is `automatic` | failed with `JsonPatchTestFailed`; **no output file**; the `replace /mag 9.9` after it never ran |
| `add /note` (a valid patch operation) | the patch applied, but the patched record failed the closed schema ("Additional properties are not allowed ('note' was unexpected)"); **no output file** |

The third row is why the schema check comes *after* the patch: JSON Patch knows nothing about your schema, so a patch can be perfectly legal and still produce a record your pipeline rejects. The original `quakes.jsonl` had the same SHA-256 afterwards.

## Step 7: Repeatable

I ran steps 2 to 6 twice from the saved snapshot in two fresh folders. The four data files (`quakes.jsonl`, `report.json`, `strong.jsonl`, `quakes_patched.jsonl`) and the five step logs all had identical SHA-256 values in both runs. That is a modest claim: both runs read the same snapshot, so it shows the *scripts* are deterministic. It says nothing about a new download, which will differ because the feed changes every minute. That is why Step 1 records the hash of the snapshot.

## The README

A pipeline is only reusable if someone else can tell what it did. Mine states: the source and URL; the snapshot time, size, count and SHA-256; the licence and how I learned it (with the caveat); a table of the seven steps with their results on this snapshot; the headline numbers; and the known limits (the crude `region` rule, the dead "(unknown)" rule, that the patch is a classroom edit, and the exact tool versions). The full file is in the solution to Exercise 3.

> **MISTAKES WHILE PREPARING THIS CHAPTER**

**1. My check for carriage returns said all 9,891 lines had one, and the file had none.** The shell test I used was wrong in this environment; counting the bytes with Python showed 0 CR and 9,891 LF. I replaced the check with `tr -cd` and an octal code, and tested it on a string that I knew contained a CRLF (it counted 1). **How to avoid it:** test a checker on input where you know the answer before believing it on input where you do not.

**2. I expected the null-place feature to be an earthquake.** My script stopped (it was set to stop on error) when it could not find that record in the cleaned file, and the reason was that it is a quarry blast. **How to avoid it:** when a rule exists for a case you have seen, check that the case reaches the rule; a count of how often a rule fired would have told me at once.

**3. I could not read the main USGS licence page.** The fetch tool returned an empty page, then a direct request returned HTTP 403, and two other USGS pages had no licence text. I used search results instead and wrote that into the README. **How to avoid it:** say exactly which source a legal or licence claim came from, and say when it is second-hand.

**4. The editing tool turned a backslash-r in a script into a real carriage-return character** (the same escape problem as in earlier chapters), which made the line unreadable in the saved file. I rewrote it with the backslash built from `chr(92)`. **How to avoid it:** after writing a script that must contain a backslash escape, print that line and look for raw control characters.

**5. My first progress echo printed "content-type: Content-Type: ..." with the label twice.** A cosmetic slip, but in a record of what a server said it is the kind of thing that makes a reader doubt the rest. **How to avoid it:** read the output of a reporting script once as a stranger would.

## Exercises

**EXERCISE 1**

Download one snapshot of the USGS `all_month` feed with a single request, and record its size, SHA-256, feed clock and the count check. Write a JSON Schema (2020-12) for the fields the pipeline needs, validate every feature with Ajv, cross-check the verdicts with a second validator in Python, and write a control that breaks copies of a real feature in six ways and shows all six are rejected. Name every invalid record and explain each error.

View solution

**EXERCISE 2**

Write a `jq` program that keeps earthquakes and flattens each to one record (ISO times, a boolean tsunami flag, split coordinates, a region), write the records as JSON Lines without CRLF, and prove nothing was lost (kept + dropped = total). Then write a Node script that streams the file, validates each record against a closed schema, and writes a report and a second JSON Lines file of the magnitude 5 and over events. Recompute the key numbers with `jq` from the original file, compare, and show that a deliberately changed number is caught. Report whether each of your cleaning rules ever fired.

View solution

**EXERCISE 3**

Write a JSON Patch with two `test` operations and two `replace` operations for one record, and a script that streams the JSON Lines file, patches only the record with that id, validates the patched record against the closed schema, and writes a new file only if everything succeeds. Run a good patch, a patch whose `test` fails and a legal patch that breaks the schema, and show which files exist afterwards, that exactly one line differs, and that the original is unchanged. Run steps 2 to 6 twice and compare checksums. Finish with a README that records the source, snapshot hash, licence and its caveat, the steps with results, and the known limits.

View solution

## Course Complete: What You Can Now Do

| CHAPTER | YOU CAN NOW | ONE FACT TO REMEMBER |
|---|---|---|
| 1 JSON Syntax | read and write exact JSON; know where parsers differ | JSON has six types and a short forbidden list; parsers disagree on edge cases |
| 2 JSON in JavaScript | use `parse`, `stringify`, revivers and replacers without surprises | numbers beyond $2^{53}$ lose digits in `JSON.parse` |
| 3 JSON in Python | control encodings, types and numbers in the `json` module | open files with an explicit encoding |
| 4 jq Basics | query JSON from the command line, with correct quoting | exit statuses and shell quoting are half of the work |
| 5 jq Reshaping | group, reduce, format and reuse programs | test a jq feature on your version before relying on it |

| CHAPTER | YOU CAN NOW | ONE FACT TO REMEMBER |
|---|---|---|
| 6 Schema Basics | describe data with JSON Schema 2020-12 | `format` is ignored unless you ask for it |
| 7 Validating | compile once, choose combinators, read errors | compiling per document was about 24,000 times slower |
| 8 Pointer, Path, Patch | address, search and change parts of a document | apply a patch to a copy; test your library's default mode |
| 9 Lines and Scale | handle large files with JSON Lines and streaming | one line at a time kept memory flat at 17-57 MB; whole-file loads grew with the file |
| 10 Other Formats | choose between JSON and its neighbours | quote YAML strings; TOML has no null |
| 11 Security and Pitfalls | defend an API boundary | build JSON with the library, never with string concatenation |
| 12 Capstone | chain it all into a checked, repeatable pipeline | validate before, cross-check after, record what you did |

**WHERE TO GO NEXT**

*API Types & Design* shows where JSON travels (REST, HTTP, status codes, webhooks); *API Testing & Tooling* covers `curl` and other tools you used here in more depth; *GraphQL* is another way to ask for JSON; and *gRPC* covers Protocol Buffers in a working service (Chapter 10 compared the formats). If `jq` or JSON Schema deserve a course of their own, that would be a separate course.

## CHAPTER 12 QUICK REFERENCE

- Pipeline: fetch once and record size, hash, clock and count; validate per feature with a compiled schema and a control; clean with `jq -b -c` into JSON Lines; stream, validate and report in a script; recompute key numbers with a second tool; patch with checks and write a new file only on success; run twice and compare checksums
- Snapshot used: USGS `all_month` GeoJSON, 2026-10-12T19:05:41Z, 7,173,808 bytes, 10,105 features; 10,104 valid, 1 invalid (null place, a quarry blast); 9,891 earthquakes kept, 214 dropped
- Report: magnitude buckets 6,898 / 1,713 / 303 / 808 / 161 / 8; 1,799 automatic and 8,092 reviewed; 5 tsunami flags; strongest M 7.7 (Panama); 58 MB peak; jq and the Node report agree

- Patch: good patch changed exactly 1 line; a failed `test` and a schema-breaking `add` each wrote no file; the original hash was unchanged
- Honest limits: the "(unknown)" rule never fired; the region rule gives "CA" next to "Alaska"; the licence was read second-hand; repeatability was shown on one snapshot only

**COURSE COMPLETE**

That is the end of *Working with JSON*. You can read, write, query, reshape, validate, address, patch, stream, convert and defend JSON, and you have run each of those against real tools. Start any chapter again from *Working with JSON 1*, or move on to *API Types & Design*.