Engineer

JSON Schema

a language for describing the shape of JSON data

What it is

JSON Schema is a language for describing what a piece of JSON has to look like, and it is written in JSON itself. A description in it, called a schema, sets out the rules: this value is an object, it must have these properties, this one is a whole number of at least one, that one a string of at most 200 characters. A program called a validator takes a schema and an instance, the data being checked, and answers whether the instance conforms, and if it does not, where and why. The project writes its name with a capital S, JSON Schema, while a single document written in the language is simply a schema, in lower case.

The specification is maintained by the JSON Schema project at json-schema.org. The current version is 2020-12 and the one before it 2019-09; earlier versions were numbered drafts, such as draft-04 and draft-07, and older tools and articles still refer to them by those names. The text comes in two parts. The core document defines how a schema is processed, how one schema refers to another and the media type `application/schema+json`, and a second document, on validation, defines the keywords that check values. The current documents for 2020-12 are both dated 16 June 2022.

Each keyword adds one constraint. `type` names one of six basic types, string, number, boolean, array, object or null, or `integer`, which accepts only numbers whose fractional part is zero. Numbers take `minimum`, `maximum` and `multipleOf`; strings take `minLength`, `maxLength` and `pattern`, a regular expression; arrays take `minItems`, `maxItems` and `uniqueItems`; objects take `properties`, `required` and `additionalProperties`. Keywords such as `allOf`, `anyOf`, `oneOf` and `not` combine schemas, and `$ref` points at a schema defined elsewhere, often under `$defs` in the same document, so a shape written once can be used in many places. Anything a schema does not mention is allowed. The empty schema, `{}`, accepts every instance there is.

A schema can also say which version of the language it is written in. The `$schema` keyword holds the address of a meta-schema, a schema that describes valid schemas for that version, and the specification page notes that the meta-schema validates itself. For 2020-12 that address is `https://json-schema.org/draft/2020-12/schema` .

Use the concept in a decision

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

Explore our services Get in touch →

Test the assumptions behind the term

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

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

Why it matters

JSON says nothing about meaning. Any well-formed document parses, whether or not the program reading it can do anything with it. Without a written description, every program that reads a document keeps its own idea of what valid looks like, spread across conditions in its code, and those ideas drift apart until a request fails in production. A schema puts the rules in one place, in a form that people and programs can both read, and it can be stored, versioned, reviewed and published like any other file.

The specification states its intent broadly: JSON Schema is meant for the validation and documentation of JSON data, for hyperlink navigation and for interaction control, and the validation vocabulary is also meant to give the screens that show the data hints about it. In practice that means one description can reject bad input at the edge of a service, document an API, drive a form, and check that the JSON a language model returns has the fields the next step expects.

It has also become the common ground that other specifications build on. OpenAPI 3.1 defines its Schema Object as a superset of JSON Schema 2020-12, so the request and response bodies in an API description are JSON Schema with a few additions. MCP, the protocol through which AI agents discover and call tools, describes each tool’s parameters with an `inputSchema` that is a JSON

Schema, and may describe its result with an `outputSchema` that clients should validate structured results against. A shared description can therefore serve a validator, an API document and an agent.

What it looks like in practice

The schema below describes the input of a tool that books a delivery slot. It names the version it is written in, lists four properties with their rules, says which three must be present, and refuses any property it does not list:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/tools/book-slot.json",
  "title": "Book a delivery slot",
  "type": "object",
  "properties": {
    "order": { "type": "string", "pattern": "^[0-9]{4,8}$" },
    "date": { "type": "string", "format": "date" },
    "pallets": { "type": "integer", "minimum": 1, "maximum": 30 },
    "note": { "type": ["string", "null"], "maxLength": 200 }
  },
  "required": ["order", "date", "pallets"],
  "additionalProperties": false
}
```

An instance that breaks the rules:

```
{ "order": "4471", "date": "27/10/2026", "pallets": 0, "pallet": 2 }
```

The instance fails on two counts. `pallets` is below the minimum of 1, and `pallet`, a typo for `pallets`, is a property the schema does not list. The missing `note` is no fault, because `note` is not in `required`; had it been sent as `null`, that would pass too, since its type allows null. The date is the instructive one. The `date` format asks for the year first, and `27/10/2026` puts the day first, but a validator running with its defaults accepts it, because the specification makes `format` an annotation that is checked only when that checking is switched on. Switch that checking on in a validator that checks dates, and the same instance fails on three counts.

A tool described this way is what this site's page on turning existing software into tools for AI agents is about. It describes each operation an agent should have, such as booking a slot or releasing an order, offered as a tool with typed inputs, errors an agent can act on and idempotent calls, so that a retry never books twice. The description is an OpenAPI document generated from the code, and an MCP server offers the same operations. The page's argument is that an agent trusts a tool's description even when it has gone out of date, and a hand-written description can drift from the implementation. The descriptions there are therefore generated from the code outward, through the database, the API, the description and the client, with a check at every step that fails the build when two of them disagree.

Where it goes wrong

A listed property that is never required. `properties` checks only the properties that are present, so an object that leaves all of them out passes. Presence is a separate keyword, and leaving `required` out means nothing is required at all. Write `required` beside `properties` every time, and keep a test case for each required property with that property removed.

Typos that pass as extra properties. Without `additionalProperties`, any property the schema does not list is allowed, so `pallet` goes through and the real `pallets` is quietly missing. Setting it to `false` closes the object, but the keyword sees only the `properties` and `patternProperties` in its own schema object. A schema that builds on another through `allOf` or `$ref` and then closes itself this way rejects the very properties it inherited. Version 2020-12 has `unevaluatedProperties` for that case, since it also counts the properties checked through `allOf`, `$ref` and the other keywords that apply schemas in place. Use `additionalProperties: false` on a schema that stands alone and `unevaluatedProperties: false` on one that is composed.

`format` **that checks nothing.** By default `format` is an annotation and not an assertion: the specification requires a validator that can check formats to keep that checking switched off unless it is asked for, and even when it is switched on, a validator may let any format pass unchecked. OpenAPI 3.1 follows the same rule, and adds that how well formats are checked varies between implementations. Switch format checking on in the validator you use, prove it with a test case that should fail, and where a format matters but cannot be checked, add a `pattern` or a check in code.

Patterns that match part of a value. A `pattern` is not anchored, so `[0-9]{4}` accepts `order 4471b` because four digits appear somewhere inside it; the specification's own example is that `es` matches `expression`. Start the pattern with `^` and end it with `$` when the whole value has to match. Patterns follow the ECMA-262 dialect, and the specification recommends a small set of constructs for patterns that should behave alike everywhere, so keep to simple classes, quantifiers and anchors.

Missing, null and empty treated as one. A property that is absent, a property set to `null` and an empty string are three different instances. `"type": "string"` rejects `null`, and a property not listed in `required` may be absent. A client that sends `null` to mean no value, against a schema written for absence, fails on every such request. Decide per property what each of the three means and write it down: `["string", "null"]` when null carries meaning, absence from `required` when the property is optional, and `minLength: 1` when an empty string is not a value.

No `$schema`, or two versions mixed. When the root of a schema has no `$schema`, the specification leaves the behaviour to the implementation, and each validator applies whichever version it defaults to. The versions differ in places that matter. In 2019-09 an array given to `items` described a tuple; in 2020-12 that job belongs to `prefixItems`, and `items` took over the role of `additionalItems`. Put `$schema` in every root, follow the release notes when moving between versions, and validate every schema against its meta-schema in CI.

Further reading

- JSON Schema Specification — the project's index of the current version, its meta-schemas and the release notes for every earlier draft.
- JSON Schema: A Media Type for Describing JSON Documents — Core 2020-12: how schemas are processed, `$schema`, `$ref` and `$defs`, and the keywords that combine schemas.
- JSON Schema Validation: A Vocabulary for Structural Validation of JSON — the keywords for each type of value, `required`, and the rules for `format`.
- JSON Schema 2020-12 Release Notes — what changed since 2019-09, with `prefixItems` and `$dynamicRef` side by side with what they replaced.
- OpenAPI Specification v3.1.1 — the Schema Object as a superset of JSON Schema 2020-12, and `format` as an annotation by default.
- Model Context Protocol: Tools — how a tool declares its parameters and its structured output in JSON Schema.

Glossary

Choose for your actual constraints

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

See what it costs Get in touch →

Make the terminology practical

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

Explore our services

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

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

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

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