Workgroup: JSON Schema
Author: A. Wright, Ed.

JSON Schema: Use Cases and Requirements

Abstract

To foster development of JSON Schema, this document contains a list of use cases and requirements that may be used to inform its development and evolution.

Note to Readers

The issues list for this document can be found at https://github.com/json-schema-org/json-schema-spec/issues.

For additional information, see https://json-schema.org/.

To provide feedback, use this issue tracker, the communication methods listed on the homepage, or email the document editors.

Table of Contents

1. Scope and Motivation
2. Conventions and Terminology
3. Objectives
   - 3.1. Validation
   - 3.2. Annotation
   - 3.3. Internet
   - 3.4. Out of scope
4. Use Cases
   - 4.1. Structural validation
   - 4.2. Semantic annotation
   - 4.3. Domain-specific language
   - 4.4. A common vocabulary
   - 4.5. Model-Driven UI constraints
   - 4.6. Fuzzing, enumeration, and generation

4.7. Embedded database constraints

4.8. Partial validation

4.9. Machine-readable profiles of Web resources

4.10. Hypermedia

4.11. Results and Reporting

4.12. Extension points

4.13. External validation

4.14. Intra-document data consistency validation

4.15. Inter-database consistency validation

4.16. Linting

5. Security Considerations

6. Informative References

Author's Address

1. Scope and Motivation

JSON Schema is a JSON media type for defining the structure of JSON data. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data.

This document elaborates in detail what this means, and the specific use cases that shall be supported.

2. Conventions and Terminology

Objectives specify the class of problems that are in the scope of the specification.

Use Cases catalog a variety of personal objectives that users may have, due to various motivations and constraints, that the specification shall accommodate, but without proscribing a specific design or implementation.

Requirements list functional, non-functional, and quality requirements, the use cases they may be derived from/related to, and reference how each use case is implemented. Requirements are not detailed at this time but may be specified in the future.

3. Objectives

JSON Schema shall be built to support the following objectives, supporting expansion into uses not currently described by any use case, but which fall within the objectives.

3.1. Validation

The first objective of JSON Schema is to describe sets of JSON documents; specifically, to notate a language of JSON documents using a machine-readable, set-builder notation. This covers the key use case of validating input using a validator, as well as numerous other tools that depend on describing to each other which kinds of JSON documents are and are not acceptable.

3.2. Annotation

The second objective of JSON Schema is to map an input JSON document to an arbitrary output described by the user. Annotations may be combined with validation (in the same schema) to specify the domain of inputs for which an output is defined. This covers the key use case of documenting the meaning of properties and values in JSON documents, and other uses where the input document is being interpreted in some fashion.

3.3. Internet

JSON is a technology standardized as a part of a larger ecosystem of Internet technology, and likewise, JSON Schema may also specify its role in this ecosystem; for example, use in HTTP, or the meaning of media type parameters.

3.4. Out of scope

Use cases or requirements that do not advance these objectives are likely out of scope, and better suited in separate work that references JSON Schema.

For example, a method of describing an API interface would exceed the scope of JSON Schema, although JSON Schema may be used as a part of such a description, such as when the API is using JSON and needs a way to describe these JSON documents.

4. Use Cases

JSON Schema shall written to standardize these use cases, or shall accommodate implementations targeted at these uses.

4.1. Structural validation

Structural validation refers to the "structure" that a JSON document is supposed to follow, such as which properties must exist, what types of values are expected where, and what they must look like. Constraints cannot read values elsewhere in the document (e.g. to compare two values for equality), though constraints may depend on "where" the value is found (e.g. specific properties or array items must be a number).

More specifically, structural validation is any validation that can be performed with a context-free grammar that follows JSON semantics, such that any forms that are equal according to JSON will produce the same result.

Validators that support structural validation can entirely replace generic grammar languages such as ABNF [RFC5234], and will guarantee compliance with JSON semantics. Likewise, schemas whose validation rules are limited to structural validation can be executed using a deterministic pushdown automata, guaranteeing a result in proportional time without error.

The following features of JSON are structural validation:

JSON primitive type (string, object, etc)

Minimum or maximum in a linear range of values

Minimum or maximum lengths (number of characters, digits, items, or properties)

Literal/constant values or alternate/enumerated values

Pattern matching strings by a regular expression (including object keys)

Logic operators (union, intersection, difference)

Descent into object properties and array items (recursively)

Multiple forms that are value-equal according to JSON are not distinguishable under this use-case. This includes the ordering of properties in an object, character escapes in strings, and whitespace.

4.2. Semantic annotation

There is a need to annotate values within a JSON document: for machine readability, and for documentation purposes. Given a document and directions for annotating it, you should should be able to:

document the meaning of a property,

suggest a default value for new documents of a given type,

fill in missing values (in objects or arrays) with values of equivalent meaning,

generate a list of hyperlinks,

or declare relationships between data.

A schema may be used to describe a machine-readable profile [RFC6906] of JSON document. Even when two schemas identify the same set of JSON documents; depending on the context, a given JSON document may be a profile of one but not the other.

4.3. Domain-specific language

Developers may write an application that uses a JSON Schema internally as a domain-specific language, so that the schema is only used inside a single application by a single party. By using a declarative language, the application requirements can be optimized better than a human could do.

Application authors may use a schema to define custom hooks and processing for the JSON (without need for standardizing the customization).

The only interoperability consideration here is that updates to the validator library must not change the validation result of any JSON documents, unless the developer specifically opts into such a breaking change (e.g. by upgrading the library to a new major version number).

4.4. A common vocabulary

A development team maintains two similar applications, but for different platforms, in different languages. The application downloads and reads from a common repository of JSON documents. They want to make sure that both applications accept or reject JSON with identical behavior, so they write a single JSON Schema and deploy it to both applications.

The only interoperability consideration here is that the two applications, given the same schema, must both be reasonably expected to support the same behavior and operate in the same manner.

4.5. Model-Driven UI constraints

When a server declares constraints that a submission must meet, there is a need for the user interface to receive these constraints to provide model-driven validation of permissible values, making the form more accessible to the user.

For example, if a value is specified to be a date, the form field where the value is specified can provide immediate feedback if an invalid date is entered, and even offer a date picker to help the user input a correct value.

Weak interoperability requirements can hamper the user experience within this use case. If the schema is ambiguous in any way, or is up to the discretion of the customer's user-agent, some customers may have a difficult time submitting a correct request.

4.6. Fuzzing, enumeration, and generation

Security applications need to generate examples of JSON documents within the valid set, and outside the valid set.

4.7. Embedded database constraints

A database that uses JSON may need a way to declare, enforce, or guarantee certain constraints on the JSON document being stored. The database may also use JSON Schema as a way to annotate certain fields as having a special meaning for uniqueness or indexing purposes.

4.8. Partial validation

Due to technical limitations, some JSON parsers may only be able to understand a subset of the JSON value space, and it makes sense to validate the value read by the application, instead of the JSON document provided to the JSON parser. For example:

- Many JSON parsers cast the arbitrary-precision decimal numbers to an IEEE floating point, validating the number after it has lost some precision.
- Some programming languages cannot distinguish between an ordered array of values and a key-value map; or an empty array is identical to an empty object.
- Other validators may have a limited amount of memory and cannot support assertions more complicated than a deterministic context-free grammar.

Users of these validators need a way to determine if the missing functionality is essential to correctly validating input, and if not, get a validation result that would be correct but-for the unimplemented functionality.

This should work both ways; an application should be able to use a third-party schema and understand if the assertions go beyond what the environment supports; and an application should be able to publish a schema that is compatible with the subset of the value space that it supports. Consider if these are separate use cases.

4.9. Machine-readable profiles of Web resources

A Web server that offers a JSON document should be able to link to a profile document that describes the meaning of the data in a machine-readable form.

4.10. Hypermedia

Generic user-agents must be able to make use of the schema as it evolves, including Web browsers, spiders, and automated tooling. It should support loose coupling (like an HTML homepage); so a schema should be able to change, add, and remove features with minimal breakage for compatible clients.

4.11. Results and Reporting

The party that is providing the schema and input may not be the same party that is performing the validation; in this case, there should be a standard way to abstract away the validator interface, and report the results of a validation operation (validation result, annotations, and errors).

4.12. Extension points

Not every feature needs to be supported by every implementation. To accommodate a wide variety of niche audiences, it should be possible to specify features that are optional to implement. This includes standardized features that are optional to implement, bespoke or user-defined features that are not standardized, and new features added to future publications of the specification. For forward compatibility, implementations that do not support optional extensions must degrade in a predictable fashion.

4.13. External validation

Authors may embed resources of other media types, such as text documents, or base64 or hex-encoded binary documents; and may wish to pass off validation of these documents to another software tool.

4.14. Intra-document data consistency validation

A JSON document may carry relational data that must be internally consistent. Example constraints include:

- One-to-one calculations, like that children's birthdates come after their parent's birthdates;
- One-to-many calculations, like a reference to an anchor points to an anchor defined somewhere in the same document.

4.15. Inter-database consistency validation

A JSON document may carry relational data that must be verified against outside data sources. Example constraints include:

- References to a user ID points to a user in a central users database.
- A supplied email address has been verified by the user.

4.16. Linting

Sometimes it's desirable to require formatting that does not impact the application-level meaning of the document, but instead specifies requirements purely for aesthetic or compatibility reasons.

For example, for security reasons, an application may want to verify a JSON document does not contain the string "`</script`" but is written instead with a character escape such as "`<\/script`". This would ensure that, if the JSON were to be embedded in a `<script>` tag, it could not close the tag and be interpreted as HTML.

This is not a standard part of JSON Schema because it may violate the semantics of JSON, by adding an ability to distinguish between encodings which are supposed to be equal to receiving applications.

5. Security Considerations

This document does not make any normative requirements.

6. Informative References

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6906] Wilde, E., "The 'profile' Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013, <https://www.rfc-editor.org/info/rfc6906>.

Author's Address

Austin Wright (editor)
Email: aaa@bzfx.net