JSON Working Group
Internet-Draft
Intended status: Standards Track
Expires: October 30, 2019

U. Carion

April 28, 2019

JSON Schema Language
draft-json-schema-language-00

Abstract

JavaScript Object Notation (JSON) Schema Language is a portable method for describing the format of JSON data and the errors associated with ill-formed data.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on October 30, 2019.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

JSON Schema Language is a schema language for JSON data. This document specifies:

- When a JSON object is a correct JSON Schema Language schema
- When a JSON document is valid with respect to a correct JSON Schema Language schema
- A standardized form of errors to produce when validating a JSON value

JSON Schema Language is centered around the question of validating a JSON value (an "instance") against a JSON object (a "schema"), within the context of a collection of other schemas (an "evaluation context").

2. Conventions

The keywords *MUST*, *MUST NOT*, *REQUIRED*, *SHALL*, *SHALL NOT*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL*, when they appear in this document, are to be interpreted as described in [RFC2119].

The terms "absolute-URI" and "URI-reference", when they appear in this document, are to be understood as they are defined in [RFC3986].

The term "JSON Pointer", when it appears in this document, is to be understood as it is defined in [RFC6901].

3. Terminology

- instance: A JSON value being validated.
- schema: A JSON object describing the form of valid instances.
- evaluation context: A collection of schemas which may refer to one another.
- validation error: A JSON object representing a reason why an instance is invalid.

4. Syntax

This section specifies when a JSON document is a correct schema.

4.1. Keywords

Some member names of a schema are reserved, and carry special meaning. These member names are called keywords. Correct schemas *MUST* satisfy the following requirements:

- "id": If a schema has a member named "id", its corresponding value *MUST* be a JSON string encoding an absolute-URI.
- "definitions": If a schema has a member named "definitions", its corresponding value *MUST* be a JSON object. The values of this object *MUST* all be correct schemas.
- "ref": If a schema has a member named "ref", its corresponding value *MUST* be a JSON string encoding a URI-reference.
- "type": If a schema has a member named "type", its corresponding value *MUST* be a JSON string encoding one of the values "null", "boolean", "number", or "string".

- "elements": If a schema has a member named "elements", its corresponding value *MUST* be a JSON object. This object *MUST* be a correct schema.

- "properties": If a schema has a member named "properties", its corresponding value *MUST* be a JSON object. The values of this object *MUST* all be correct schemas.

- "optionalProperties": If a schema has a member named "optionalProperties", its corresponding value *MUST* be a JSON object. The values of this object *MUST* all be correct schemas.

- "values": If a schema has a member named "values", its corresponding value *MUST* be a JSON object. This object *MUST* be a correct schema.

- "discriminator": If a schema has a member named "discriminator", its corresponding value *MUST* be a JSON object. This object *MUST* have exactly two members:

  * A member with the name "tag", whose corresponding value *MUST* be a JSON string.

  * A member with the name "mapping", whose corresponding value *MUST* be a JSON object. The values of this object *MUST* all be correct schemas.

4.2. Forms

Only certain combinations of schema keywords are correct. These valid combinations are called "forms". Correct schemas *MUST* fall into exactly one of the following forms:

- The "empty" form: the schema may have members with the name "id" and/or "definitions", but none of the other keywords listed in Section 4.1.

- The "ref" form: the schema may have members with the name "id", "definitions", and/or "ref", but none of the other keywords listed in Section 4.1.

- The "type" form: the schema may have members with the name "id", "definitions", and/or "type", but none of the other keywords listed in Section 4.1.

- The "elements" form: the schema may have members with the name "id", "definitions", and/or "elements", but none of the other keywords listed in Section 4.1.

- The "properties" form: the schema may have members with the name "id", "definitions", "properties", and/or "optionalProperties", but none of the other keywords listed in Section 4.1.

- The "values" form: the schema may have members with the name "id", "definitions", and/or "values", but none of the other keywords listed in Section 4.1.

- The "discriminator" form: the schema may have members with the name "id", "definitions", and/or "discriminator", but none of the other keywords listed in Section 4.1.

4.3. Additional restrictions to prevent ambiguity

To prevent ambiguous or unsatisfiable schemas during evaluation (see Section 5.3), there are two additional constraints that all JSON documents must satisfy to be a valid schema:

1. If a schema both "properties" and "optionalProperties" members, the "properties" and "optionalProperties" values *MUST NOT* share any member names in common.

   Without this restriction, it could be ambiguous whether a property is required or not.

2. If a schema has a "discriminator" member, all of the values of "mapping" within "discriminator" *MUST* be of the "properties" form described in Section 4.2. Furthermore, these schemas within "mapping" *MUST NOT* have a member in "properties" or "optionalProperties" whose name equals that of "tag"'s within "discriminator".

   Without this restriction, it could be possible for a schema to require that an instance be simultaneously an object and not an object. Additionally, schemas might also give contradictory requirements by describing the same instance member through both "tag" and "properties".

To illustrate the first restriction, the following JSON document is not a valid schema, as "foo" appears both in "properties" and "optionalProperties":

```
{
  "properties": { "foo": {} },
  "optionalProperties": { "foo": {} }
}
```

To illsturate the second restriction, the following JSON document is not a valid schema because one of the members of "mapping" is not of the "properties" form:

```
{
  "discriminator": {
    "tag": "foo",
    "mapping": {
      "a": { "elements": {} }
    }
  }
}
```

Finally, the following JSON document is not a valid schema because one of the members of "mapping" has a "properties" member whose value equals that of "tag"'s:

```
{
  "discriminator": {
    "tag": "foo",
    "mapping": {
      "a": { "properties": { "foo": { "type": "number" } } }
    }
  }
}
```

4.4. Evaluation context and reference resolution

An evaluation context is a collection of schemas which may refer to one another. An evaluation context is correct if:

- All of its constituent schemas are correct,
- No two constituent schemas have the same "id" value, and
- No more than one schema lacks an "id" value.

If a schema is correct and it has a member named "ref", then this member is said to be a reference. The reference of a correct schema *MUST* be resolvable. Reference resolution is defined as follows:

1. By Section 4.1, a schema may be contained by another schema. Reference resolution uses the "root" of a schema to determine a base URI. The "root" of a given schema is the immediate element of an evaluation context which contains the given schema. All schemas are, for this definition, considered to contain themselves.

2. By Section 4.1, the value of the reference must be a URI-reference. This URI-reference is resolved using the process described in [RFC3986] to produce a resolved URI. If the root of a schema has a member named "id", then that member's corresponding value shall be used as the base URI for the URI resolution process; otherwise, no base URI is used.

   If the URI-reference cannot be resolved, then the reference is unresolvable.

3. Take the URI from (2), and remove its fragment part, if present.

4. Find the element of the evaluation context which has a member named "id" and whose value equals the URI from (3). If there does not exist such a schema, then the reference is unresolvable.

5. If URI from (2) has no fragment, then the reference resolves to the schema from (4).

6. Otherwise, the schema from (4) must have a member named "definitions"; if it does not, then the reference is unresolvable. Furthermore, the "definitions" value must have a member whose name equals the fragment of the URI from (2); if it does not, then the reference is unresolvable. If it does have such a member, then the reference resolves to this member's value.

For example, if an evaluation context contains two schemas:

```
{
  "id": "http://example.com",
  "ref": "/foo#a"
}

{
  "id": "http://example.com/foo",
  "definitions": {
    "a": {
      "ref": "#"
    },
    "b": {
      "id": "http://example.com/bar",
      "ref": "#"
    }
  }
}
```

Then the reference with value "/foo#a" refers to the "a" definition of the schema with ID "http://example.com/foo". Both of the references with value "#" refer the root schema with ID "http://example.com/foo". The "id" keyword of the "b" definition is irrelevant, as it occurs outside of a root schema.

5. Semantics

This section specifies when an instance is valid against a correct schema, within the context of an evaluation context. This section also specifies a standardized form of errors to produce when validating an instance.

5.1. Configuration

Users will have different desired behavior with respect to unspecified members in a schema or instance. Two distinct sets of semantics (one for schemas, another for instances), determine whether unspecified members are acceptable.

5.1.1. Strict schema semantics

When evaluation is using strict schema semantics, then a correct schema *MUST NOT* contain members whose names are outside the list of keywords described in Section 4.1. When evaluation is not using strict schema semantics, then a correct schema *MAY* contain members whose names are outside this list.

Implementations *MAY* allow users to choose whether to use strict schema semantics. Implementations *SHOULD* document whether they use strict schema semantics by default.

5.1.2. Strict instance semantics

See Section 5.3.5 for how strict instance semantics affects whether an instance is valid with respect to a schema.

Implementations *MAY* allow users to choose whether to use strict instance semantics. Implementations *SHOULD* document whether they use strict instance semantics by default.

5.2. Errors

To facilitate consistent validation error handling, this document specifies a standard error format. Implementations *SHOULD* support producing errors in this standard form.

The standard error format is a JSON array. The order of the elements of this array is not specified. The elements of this array are JSON objects with up to three members:

- A member with the name "instancePath", whose value is a JSON string containing a JSON Pointer. This JSON Pointer will point to the part of the instance that was rejected.

- A member with the name "schemaPath", whose value is a JSON string containing a JSON Pointer. This JSON Pointer will point to the part of the schema that rejected the instance.

- A member with the name "schemaURI", whose value is an absolute-URI. This URI will be the "id" value of the root schema of the schema that rejected the instance. See Section 4.4 for a definition of a schema's root. If the root schema lacks an "id" value, then the "schemaURI" member shall be omitted.

The values for "instancePath" and "schemaPath" depend on the form of the schema, and are described in detail in Section 5.3.

5.3. Evaluation

Whether an instance is valid against a schema depends upon the form of the schema. This section describes how each form validates instances.

5.3.1. Empty form

If a schema is of the "empty" form, then it accepts all instances.

5.3.2. Ref form

The "ref" form is meant to enable schema re-use.

If a schema is of the "ref" form, then it accepts an instance if and only if the schema which the "ref" member resolves to accepts the instance. The standard errors to produce are the same as those that the referent schema produces. The resolution of a "ref" member is described in Section 4.4.

For example, if we evaluate the instance:

"example"

Against the schema:

```
{
  "ref": "http://example.com"
}
```

Within an evaluating context containing the schema:

```
{
  "id": "http://example.com",
  "type": "number"
}
```

Then the standard errors are:

```
[
  {
    "instancePath": "",
    "schemaPath": "/type",
    "schemaURI": "http://example.com"
  }
]
```

See Section 5.3.3 for how the "type" member produces errors, as the errors in the example above compose upon "type" errors.

5.3.3. Type form

The "type" form is meant to describe the primitive data types of JSON.

If a schema is of the "type" form, then:

- If the value of the "type" member is "null", then the instance is accepted if it equals "null".
- If the value of the "type" member is "boolean", then the instance is accepted if it equals "true" or "false".
- If the value of the "type" member is "number", then the instance is accepted if it is a JSON number.
- If the value of the "type" member is "string", then the instance is accepted if it is a JSON string.

If the instance is not accepted, then the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the "type" member.

For example, if we evaluate the instance:

"example"

Against the schema:

{ "type": "number" }

Then the standard errors are:

[{ "instancePath": "", "schemaPath": "/type" }]

5.3.4. Elements form

The "elements" form is meant to describe JSON arrays representing homogeneous data. When a schema is of the "elements" form, it validates:

- That the instance is an array, and
- That all of the elements of the array are of the same type.

If a schema is of the "elements" form, then:

1. If the instance is not a JSON array, then the instance is rejected. The standard error shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the "elements" member.

2. Otherwise, the instance is accepted if each element of the instance is accepted by the value of the "elements" member. The standard error shall be the concatenation of the standard errors from evaluating each element of the instance against the value of the "elements" member.

For example, if we have the schema:

```
{
  "elements": {
    "type": "number"
  }
}
```

Then if we evaluate the instance:

"example"

Against this schema, the standard errors are:

[{ "instancePath": "", "schemaPath": "/elements" }]

If instead we evaluate the instance:

```
[1, 2, "foo", 3, "bar"]
```

The standard errors are:

```
[
  { "instancePath": "/2", "schemaPath": "/elements/type" },
  { "instancePath": "/4", "schemaPath": "/elements/type" }
]
```

5.3.5. Properties form

The "properties" form is meant to describe JSON objects being used in a fashion similar to structs in C-like languages. When a schema is of the "properties" form, it validates:

- That the instance is an object,
- That the instance has a set of required properties, each satisfying their own respective schema, and
- That the instance may have a set of optional properties that, if present in the instance, satisfy their own respective schema.

If a schema is of the "properties" form, then:

1. If the instance is not a JSON object, then the instance is rejected.

   The standard error for this case has an "instancePath" pointing to the instance. If the schema has a "properties" member, then the "schemaPath" of the error shall point to the "properties" member. Otherwise, "schemaPath" shall point to the "optionalProperties" member.

2. If the instance is a JSON object, and the schema has a "properties" member, then for each member name of the "properties" of the schema, a member of the same name must appear in the instance. Otherwise, the instance is rejected.

   The standard error for this case has an "instancePath" pointing to the instance, and a "schemaPath" pointing to the member of "properties" whose name lacks a counterpart in the instance.

3. If the instance is a JSON object, then for each member of the instance, find a member of the same name in the "properties" or "optionalProperties" of the schema.

* If no such member in the "properties" or "optionalProperties" exists, and validation is using strict instance semantics, then the instance is rejected.

  The standard error for this case has an "instancePath" pointing to the member of the instance lacking a counterpart in the schema, and a "schemaPath" pointing to the schema.

* If such a member in the "properties" or "optionalProperties" does exist, then the value of the member from the instance must be accepted by the value of the corresponding member from the schema. Otherwise, the instance is rejected.

  The standard error for this case is the concatenation of the errors from evaluating the member of the instance against the member of the schema.

An instance may have errors arising from both (2) and (3). In this case, the standard errors should be concatenated together.

For example, if we have the schema:

```
{
  "properties": {
    "a": { "type": "string" },
    "b": { "type": "string" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "string" }
  }
}
```

Then if we evaluate the instance:

```
"example"
```

Against this schema, then the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/properties" }]
```

If instead we evalute the instance:

```
{ "b": 3, "c": 3, "e": 3 }
```

The standard errors, using strict instance semantics, are:

```
[
  { "instancePath": "",
    "schemaPath": "/properties/a" },
  { "instancePath": "/b",
    "schemaPath": "/properties/b/type" },
  { "instancePath": "/c",
    "schemaPath": "/optionalProperties/c/type" },
  { "instancePath": "/e",
    "schemaPath": "" }
]
```

If we the same instance were evaluated, but without strict instance semantics, the final element of the above array of errors would not be present.

5.3.6. Values form

The "values" form is meant to describe JSON objects being used as an associative array mapping arbitrary strings to values all of the same type. When a schema is of the "properties" form, it validates:

- That the instance is an object, and
- That the values of the instance all satisfy the same schema.

If a schema is of the "values" form, then:

1. If the instance is not a JSON object, then the instance is rejected. The standard error shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the "values" member.
2. Otherwise, the instance is accepted if the value of each member of the instance is accepted by the value of the "values" member. The standard error shall be the concatenation of the standard errors from evaluating the value of each member of the instance against the value of the "values" member.

For example, if we have the schema:

```
{
  "values": {
    "type": "number"
  }
}
```

Then if we evaluate the instance:

```
"example"
```

Against this schema, the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/values" }]
```

If instead we evaluate the instance:

```
{ "a": 1, "b": 2, "c": "foo", "d": 3, "e": "bar" }
```

The standard errors are:

```
[
  { "instancePath": "/c", "schemaPath": "/values/type" },
  { "instancePath": "/e", "schemaPath": "/values/type" }
]
```

5.3.7. Discriminator form

The "discriminator" form is meant to describe JSON objects being used in a fashion similar to a discriminated union construct in C-like languages. When a schema is of the "disciminator" type, it validates:

- That the instance is an object,
- That the instance has a particular "disciminator" property,
- That this "discriminator" value is a string within a set of valid values, and
- That the instance satisfies another schema, where this other schema is chosen based on the value of the "discriminator" property.

If a schema is of the "disciminator" form, then:

1. If the instance is not a JSON object, then the instance is rejected. The standard error shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the "discriminator" member.
2. If the instance is a JSON object and lacks a member whose name equals the "tag" value of the "discriminator" of the schema, then the instance is rejected.

The standard error to produce in this case has an "instancePath" pointing to the instance, and a "schemaPath" pointing to the "tag" member of the "disciminator" member of the schema.

3. If the instance is a JSON object and has a member whose name equals the "tag" value of the "discriminator" of the schema, but that member's value is not a string, then the instance is rejected.

   The standard error to produce in this case has an "instancePath" pointing to the member of the instance corresponding to "tag", and a "schemaPath" pointing to the "tag" member of the discriminator.

4. If the instance is a JSON object and has a member whose name equals the "tag" value of the "discriminator" of the schema and whose value is a string, but that member's value is not equal to any of the member names in the "mapping" of the "discriminator", then the instance is rejected.

   The standard error to produce in this case has an "instancePath" pointing to the member of the instance corresponding to "tag", and a "schemaPath" pointing to the "mapping" member of the "discriminator" member of the schema.

5. If the instance is a JSON object and has a member whose name equals the "tag" value of the "discriminator" of the schema, and that member's value is equal to one of the member names in the "mapping" of the "discriminator", then the instance must satisfy this corresponding schema in "mapping". Otherwise, the instance is rejected.

   The standard errors to produce in this case are those produced by evaluating the instance against the schema within the "mapping".

For example, if we have the schema:

```
{
  "discriminator": {
    "tag": "version",
    "mapping": {
      "v1": {
        "properties": {
          "a": { "type": "number" }
        }
      },
      "v2": {
        "properties": {
          "a": { "type": "string" }
        }
      }
    }
  }
}
```

Then if we evaluate the instance:

```
"example"
```

Against this schema, the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/discriminator" }]
```

If we instead evaluate the instance:

```
{}
```

Then the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/discriminator/tag" }]
```

If we instead evaluate the instance:

```
{ "version": 1 }
```

Then the standard errors are:

```
[{ "instancePath": "/version", "schemaPath": "/discriminator/tag" }]
```

If we instead evaluate the instance:

```
{
  "version": "v3"
}
```

Then the standard errors are:

```
[
  { "instancePath": "/version",
    "schemaPath": "/discriminator/mapping" }
]
```

Finally, if the instance evaluated were:

```
{
  "version": "v2",
  "a": 3
}
```

Then the standard errors are:

```
[
  {
    "instancePath": "/a",
    "schemaPath": "/discriminator/mapping/v2/properties/a/type"
  }
]
```

6. IANA Considerations

No IANA considerations.

7. Security Considerations

Implementations of JSON Schema Language will necessarily be manipulating JSON data. Therefore, the security considerations of [RFC8259] are all relevant here.

Implementations which evaluate user-inputted schemas *SHOULD* implement mechanisms to detect, and abort, circular references which might cause a naive implementation to go into an infinite loop. Without such mechanisms, implementations may be vulnerable to denial-of-service attacks.

Implementations of JSON Schema Language *SHOULD NOT* naively attempt to fetch and evaluate schemas when they are referred to using the "ref" keyword. Doing so could lead to denial of service. Instead, implementations should only fetch schemas through secure channels, and should only fetch and evaluate schemas from trusted sources.

8. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/info/rfc6901>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

Appendix A. Acknowledgments

Thanks to Gary Court, Francis Galiegue, Kris Zyp, Geraint Luff, Jason Desrosiers, Daniel Perrett, Erik Wilde, Ben Hutton, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave Finlay, Denis Laxalde, Henry Andrews, and Austin Wright for their work on the initial drafts of JSON Schema, which inspired JSON Schema Language.

Author's Address

Ulysse Carion

Email: ulyssecarion@gmail.com