Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: March 19, 2020

A. Wright, Ed.

H. Andrews, Ed.

B. Hutton, Ed.
Wellcome Sanger Institute

September 16, 2019

JSON Schema Validation: A Vocabulary for Structural Validation of JSON
draft-handrews-json-schema-validation-02

Abstract

JSON Schema (application/schema+json) has several purposes, one of which is JSON instance validation. This document specifies a vocabulary for JSON Schema to describe the meaning of JSON documents, provide hints for user interfaces working with JSON data, and to make assertions about what a valid document must look like.

Note to Readers

The issues list for this draft 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.

Status of This Memo

Table of Contents

1. Introduction

JSON Schema can be used to require that a given JSON document (an instance) satisfies a certain number of criteria. These criteria are asserted by using keywords described in this specification. In addition, a set of keywords is also defined to assist in interactive user interface instance generation.

This specification will use the concepts, syntax, and terminology defined by the JSON Schema core [json-schema] specification.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

This specification uses the term "container instance" to refer to both array and object instances. It uses the term "children instances" to refer to array elements or object member values.

Elements in an array value are said to be unique if no two elements of this array are equal [json-schema].

3. Overview

JSON Schema validation asserts constraints on the structure of instance data. An instance location that satisfies all asserted constraints is then annotated with any keywords that contain non-assertion information, such as descriptive metadata and usage hints. If all locations within the instance satisfy all asserted constraints, then the instance is said to be valid against the schema.

Each schema object is independently evaluated against each instance location to which it applies. This greatly simplifies the implementation requirements for validators by ensuring that they do not need to maintain state across the document-wide validation process.

This specification defines a set of assertion keywords, as well as a small vocabulary of metadata keywords that can be used to annotate the JSON instance with useful information. The Section 7 keyword is intended primarily as an annotation, but can optionally be used as an assertion. The Section 8 keywords are annotations for working with documents embedded as JSON strings.

4. Interoperability Considerations

4.1. Validation of String Instances

It should be noted that the nul character (\u0000) is valid in a JSON string. An instance to validate may contain a string value with this character, regardless of the ability of the underlying programming language to deal with such data.

4.2. Validation of Numeric Instances

The JSON specification allows numbers with arbitrary precision, and JSON Schema does not add any such bounds. This means that numeric instances processed by JSON Schema can be arbitrarily large and/or have an arbitrarily long decimal part, regardless of the ability of the underlying programming language to deal with such data.

4.3. Regular Expressions

Keywords that use regular expressions, or constrain the instance value to be a regular expression, are subject to the interoperability considerations for regular expressions in the JSON Schema Core [json-schema] specification.

5. Meta-Schema

The current URI for the default JSON Schema meta-schema is <http://json-schema.org/draft/2019-09/schema>. For schema author convenience, this meta-schema describes all vocabularies defined in this specification and the JSON Schema Core specification, as well as two former keywords which are reserved for a transitional period. Individual vocabulary and vocabulary meta-schema URIs are given for each section below. Certain vocabularies are optional to support, which is explained in detail in the relevant sections.

Updated vocabulary and meta-schema URIs MAY be published between specification drafts in order to correct errors. Implementations SHOULD consider URIs dated after this specification draft and before the next to indicate the same syntax and semantics as those listed here.

6. A Vocabulary for Structural Validation

Validation keywords in a schema impose requirements for successful validation of an instance. These keywords are all assertions without any annotation behavior.

Meta-schemas that do not use "$vocabulary" SHOULD be considered to require this vocabulary as if its URI were present with a value of true.

The current URI for this vocabulary, known as the Validation vocabulary, is: <https://json-schema.org/draft/2019-09/vocab/validation>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2019-09/meta/validation>.

6.1. Validation Keywords for Any Instance Type

6.1.1. type

The value of this keyword MUST be either a string or an array. If it is an array, elements of the array MUST be strings and MUST be unique.

String values MUST be one of the six primitive types ("null", "boolean", "object", "array", "number", or "string"), or "integer" which matches any number with a zero fractional part.

An instance validates if and only if the instance is in any of the sets listed for this keyword.

6.1.2. enum

The value of this keyword MUST be an array. This array SHOULD have at least one element. Elements in the array SHOULD be unique.

An instance validates successfully against this keyword if its value is equal to one of the elements in this keyword's array value.

Elements in the array might be of any type, including null.

6.1.3. const

The value of this keyword MAY be of any type, including null.

Use of this keyword is functionally equivalent to an "enum" (Section 6.1.2) with a single value.

An instance validates successfully against this keyword if its value is equal to the value of the keyword.

6.2. Validation Keywords for Numeric Instances (number and integer)

6.2.1. multipleOf

The value of "multipleOf" MUST be a number, strictly greater than 0.

A numeric instance is valid only if division by this keyword's value results in an integer.

6.2.2. maximum

The value of "maximum" MUST be a number, representing an inclusive upper limit for a numeric instance.

If the instance is a number, then this keyword validates only if the instance is less than or exactly equal to "maximum".

6.2.3. exclusiveMaximum

The value of "exclusiveMaximum" MUST be number, representing an exclusive upper limit for a numeric instance.

If the instance is a number, then the instance is valid only if it has a value strictly less than (not equal to) "exclusiveMaximum".

6.2.4. minimum

The value of "minimum" MUST be a number, representing an inclusive lower limit for a numeric instance.

If the instance is a number, then this keyword validates only if the instance is greater than or exactly equal to "minimum".

6.2.5. exclusiveMinimum

The value of "exclusiveMinimum" MUST be number, representing an exclusive lower limit for a numeric instance.

If the instance is a number, then the instance is valid only if it has a value strictly greater than (not equal to) "exclusiveMinimum".

6.3. Validation Keywords for Strings

6.3.1. maxLength

The value of this keyword MUST be a non-negative integer.

A string instance is valid against this keyword if its length is less than, or equal to, the value of this keyword.

The length of a string instance is defined as the number of its characters as defined by RFC 8259 [RFC8259].

6.3.2. minLength

The value of this keyword MUST be a non-negative integer.

A string instance is valid against this keyword if its length is greater than, or equal to, the value of this keyword.

The length of a string instance is defined as the number of its characters as defined by RFC 8259 [RFC8259].

Omitting this keyword has the same behavior as a value of 0.

6.3.3. pattern

The value of this keyword MUST be a string. This string SHOULD be a valid regular expression, according to the ECMA 262 regular expression dialect.

A string instance is considered valid if the regular expression matches the instance successfully. Recall: regular expressions are not implicitly anchored.

6.4. Validation Keywords for Arrays

6.4.1. maxItems

The value of this keyword MUST be a non-negative integer.

An array instance is valid against "maxItems" if its size is less than, or equal to, the value of this keyword.

6.4.2. minItems

The value of this keyword MUST be a non-negative integer.

An array instance is valid against "minItems" if its size is greater than, or equal to, the value of this keyword.

Omitting this keyword has the same behavior as a value of 0.

6.4.3. uniqueItems

The value of this keyword MUST be a boolean.

If this keyword has boolean value false, the instance validates successfully. If it has boolean value true, the instance validates successfully if all of its elements are unique.

Omitting this keyword has the same behavior as a value of false.

6.4.4. maxContains

The value of this keyword MUST be a non-negative integer.

An array instance is valid against "maxContains" if the number of elements that are valid against the schema for "contains" [json-schema] is less than, or equal to, the value of this keyword.

If "contains" is not present within the same schema object, then this keyword has no effect.

6.4.5. minContains

The value of this keyword MUST be a non-negative integer.

An array instance is valid against "minContains" if the number of elements that are valid against the schema for "contains" [json-schema] is greater than, or equal to, the value of this keyword.

A value of 0 is allowed, but is only useful for setting a range of occurrences from 0 to the value of "maxContains". A value of 0 with no "maxContains" causes "contains" to always pass validation.

If "contains" is not present within the same schema object, then this keyword has no effect.

Omitting this keyword has the same behavior as a value of 1.

6.5. Validation Keywords for Objects

6.5.1. maxProperties

The value of this keyword MUST be a non-negative integer.

An object instance is valid against "maxProperties" if its number of properties is less than, or equal to, the value of this keyword.

6.5.2. minProperties

The value of this keyword MUST be a non-negative integer.

An object instance is valid against "minProperties" if its number of properties is greater than, or equal to, the value of this keyword.

Omitting this keyword has the same behavior as a value of 0.

6.5.3. required

The value of this keyword MUST be an array. Elements of this array, if any, MUST be strings, and MUST be unique.

An object instance is valid against this keyword if every item in the array is the name of a property in the instance.

Omitting this keyword has the same behavior as an empty array.

6.5.4. dependentRequired

The value of this keyword MUST be an object. Properties in this object, if any, MUST be arrays. Elements in each array, if any, MUST be strings, and MUST be unique.

This keyword specifies properties that are required if a specific other property is present. Their requirement is dependent on the presence of the other property.

Validation succeeds if, for each name that appears in both the instance and as a name within this keyword's value, every item in the corresponding array is also the name of a property in the instance.

Omitting this keyword has the same behavior as an empty object.

7. A Vocabulary for Semantic Content With "format"

7.1. Foreword

Structural validation alone may be insufficient to allow an application to correctly utilize certain values. The "format" annotation keyword is defined to allow schema authors to convey semantic information for a fixed subset of values which are accurately described by authoritative resources, be they RFCs or other external specifications.

Implementations MAY treat "format" as an assertion in addition to an annotation, and attempt to validate the value's conformance to the specified semantics. See the Implementation Requirements below for details.

The value of this keyword is called a format attribute. It MUST be a string. A format attribute can generally only validate a given set of instance types. If the type of the instance to validate is not in this set, validation for this format attribute and instance SHOULD succeed. All format attributes defined in this section apply to strings, but a format attribute can be specified to apply to any

instance types defined in the data model defined in the core JSON Schema. [json-schema] [[CREF1: Note that the "type" keyword in this specification defines an "integer" type which is not part of the data model. Therefore a format attribute can be limited to numbers, but not specifically to integers. However, a numeric format can be used alongside the "type" keyword with a value of "integer", or could be explicitly defined to always pass if the number is not an integer, which produces essentially the same behavior as only applying to integers.]]

Meta-schemas that do not use "$vocabulary" SHOULD be considered to utilize this vocabulary as if its URI were present with a value of false. See the Implementation Requirements below for details.

The current URI for this vocabulary, known as the Format vocabulary, is: <https://json-schema.org/draft/2019-09/vocab/format>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2019-09/meta/format>.

7.2. Implementation Requirements

The "format" keyword functions as an annotation, and optionally as an assertion. [[CREF2: This is due to the keyword's history, and is not in line with current keyword design principles.]] In order to manage this ambiguity, the "format" keyword is defined in its own separate vocabulary, as noted above. The true or false value of the vocabulary declaration governs the implementation requirements necessary to process a schema that uses "format", and the behaviors on which schema authors can rely.

7.2.1. As an annotation

The value of format MUST be collected as an annotation, if the implementation supports annotation collection. This enables application-level validation when schema validation is unavailable or inadequate.

This requirement is not affected by the boolean value of the vocabulary declaration, nor by the configuration of "format"'s assertion behavior described in the next section. [[CREF3: Requiring annotation collection even when the vocabulary is declared with a value of false is atypical, but necessary to ensure that the best practice of performing application-level validation is possible even when assertion evaluation is not implemented. Since "format" has always been a part of this specification, requiring implementations to be aware of it even with a false vocabulary declaration is deemed to not be a burden.]]

7.2.2. As an assertion

Regardless of the boolean value of the vocabulary declaration, an implementation that can evaluate "format" as an assertion MUST provide options to enable and disable such evaluation. The assertion evaluation behavior when the option is not explicitly specified depends on the vocabulary declaration's boolean value.

When implementing this entire specification, this vocabulary MUST be supported with a value of false (but see details below), and MAY be supported with a value of true.

When the vocabulary is declared with a value of false, an implementation:

- MUST NOT evaluate "format" as an assertion unless it is explicitly configured to do so;
- SHOULD provide an implementation-specific best effort validation for each format attribute defined below;
- MAY choose to implement validation of any or all format attributes as a no-op by always producing a validation result of true;
- SHOULD document its level of support for validation.

[[CREF4: This matches the current reality of implementations, which provide widely varying levels of validation, including no validation at all, for some or all format attributes. It is also designed to encourage relying only on the annotation behavior and performing semantic validation in the application, which is the recommended best practice.]]

When the vocabulary is declared with a value of true, an implementation that supports this form of the vocabulary:

- MUST evaluate "format" as an assertion unless it is explicitly configured not to do so;
- MUST implement syntactic validation for all format attributes defined in this specification, and for any additional format attributes that it recognizes, such that there exist possible instance values of the correct type that will fail validation.

The requirement for minimal validation of format attributes is intentionally vague and permissive, due to the complexity involved in many of the attributes. Note in particular that the requirement is limited to syntactic checking; it is not to be expected that an

implementation would send an email, attempt to connect to a URL, or otherwise check the existence of an entity identified by a format instance. [[CREF5: The expectation is that for simple formats such as date-time, syntactic validation will be thorough. For a complex format such as email addresses, which are the amalgamation of various standards and numerous adjustments over time, with obscure and/or obsolete rules that may or may not be restricted by other applications making use of the value, a minimal validation is sufficient. For example, an instance string that does not contain an "@" is clearly not a valid email address, and an "email" or "hostname" containing characters outside of 7-bit ASCII is likewise clearly invalid.]]

It is RECOMMENDED that implementations use a common parsing library for each format, or a well-known regular expression. Implementations SHOULD clearly document how and to what degree each format attribute is validated.

The standard core and validation meta-schema (Section 5) includes this vocabulary in its "$vocabulary" keyword with a value of false, since by default implementations are not required to support this keyword as an assertion. Supporting the format vocabulary with a value of true is understood to greatly increase code size and in some cases execution time, and will not be appropriate for all implementations.

7.2.3. Custom format attributes

Implementations MAY support custom format attributes. Save for agreement between parties, schema authors SHALL NOT expect a peer implementation to support such custom format attributes. An implementation MUST NOT fail validation or cease processing due to an unknown format attribute. When treating "format" as an annotation, implementations SHOULD collect both known and unknown format attribute values.

Vocabularies do not support specifically declaring different value sets for keywords. Due to this limitation, and the historically uneven implementation of this keyword, it is RECOMMENDED to define additional keywords in a custom vocabulary rather than additional format attributes if interoperability is desired.

7.3. Defined Formats

7.3.1. Dates, Times, and Duration

These attributes apply to string instances.

Date and time format names are derived from RFC 3339, section 5.6 [RFC3339]. The duration format is from the ISO 8601 ABNF as given in Appendix A of RFC 3339.

Implementations supporting formats SHOULD implement support for the following attributes:

date-time: A string instance is valid against this attribute if it is a valid representation according to the "date-time" production.

date: A string instance is valid against this attribute if it is a valid representation according to the "full-date" production.

time: A string instance is valid against this attribute if it is a valid representation according to the "full-time" production.

duration: A string instance is valid against this attribute if it is a valid representation according to the "duration" production.

Implementations MAY support additional attributes using the other production names defined in that section. If "full-date" or "full-time" are implemented, the corresponding short form ("date" or "time" respectively) MUST be implemented, and MUST behave identically. Implementations SHOULD NOT define extension attributes with any name matching an RFC 3339 production unless it validates according to the rules of that production. [[CREF6: There is not currently consensus on the need for supporting all RFC 3339 formats, so this approach of reserving the namespace will encourage experimentation without committing to the entire set. Either the format implementation requirements will become more flexible in general, or these will likely either be promoted to fully specified attributes or dropped.]]

7.3.2. Email Addresses

These attributes apply to string instances.

A string instance is valid against these attributes if it is a valid Internet email address as follows:

email: As defined by RFC 5322, section 3.4.1 [RFC5322].

idn-email: As defined by RFC 6531 [RFC6531]

Note that all strings valid against the "email" attribute are also valid against the "idn-email" attribute.

7.3.3. Hostnames

These attributes apply to string instances.

A string instance is valid against these attributes if it is a valid representation for an Internet hostname as follows:

hostname: As defined by RFC 1123, section 2.1 [RFC1123], including host names produced using the Punycode algorithm specified in RFC 5891, section 4.4 [RFC5891].

idn-hostname: As defined by either RFC 1123 as for hostname, or an internationalized hostname as defined by RFC 5890, section 2.3.2.3 [RFC5890].

Note that all strings valid against the "hostname" attribute are also valid against the "idn-hostname" attribute.

7.3.4. IP Addresses

These attributes apply to string instances.

A string instance is valid against these attributes if it is a valid representation of an IP address as follows:

ipv4: An IPv4 address according to the "dotted-quad" ABNF syntax as defined in RFC 2673, section 3.2 [RFC2673].

ipv6: An IPv6 address as defined in RFC 4291, section 2.2 [RFC4291].

7.3.5. Resource Identifiers

These attributes apply to string instances.

uri: A string instance is valid against this attribute if it is a valid URI, according to [RFC3986].

uri-reference: A string instance is valid against this attribute if it is a valid URI Reference (either a URI or a relative-reference), according to [RFC3986].

iri: A string instance is valid against this attribute if it is a valid IRI, according to [RFC3987].

iri-reference: A string instance is valid against this attribute if it is a valid IRI Reference (either an IRI or a relative-reference), according to [RFC3987].

uuid: A string instance is valid against this attribute if it is a valid string representation of a UUID, according to [RFC4122].

Note that all valid URIs are valid IRIs, and all valid URI References are also valid IRI References.

Note also that the "uuid" format is for plain UUIDs, not UUIDs in URNs. An example is "f81d4fae-7dec-11d0-a765-00a0c91e6bf6". For UUIDs as URNs, use the "uri" format, with a "pattern" regular expression of "^urn:uuid:" to indicate the URI scheme and URN namespace.

7.3.6. uri-template

This attribute applies to string instances.

A string instance is valid against this attribute if it is a valid URI Template (of any level), according to [RFC6570].

Note that URI Templates may be used for IRIs; there is no separate IRI Template specification.

7.3.7. JSON Pointers

These attributes apply to string instances.

json-pointer: A string instance is valid against this attribute if it is a valid JSON string representation of a JSON Pointer, according to RFC 6901, section 5 [RFC6901].

relative-json-pointer: A string instance is valid against this attribute if it is a valid Relative JSON Pointer [relative-json-pointer].

To allow for both absolute and relative JSON Pointers, use "anyOf" or "oneOf" to indicate support for either format.

7.3.8. regex

This attribute applies to string instances.

A regular expression, which SHOULD be valid according to the ECMA 262 [ecma262] regular expression dialect.

Implementations that validate formats MUST accept at least the subset of ECMA 262 defined in the Regular Expressions (Section 4.3) section of this specification, and SHOULD accept all valid ECMA 262 expressions.

8. A Vocabulary for the Contents of String-Encoded Data

8.1. Foreword

Annotations defined in this section indicate that an instance contains non-JSON data encoded in a JSON string.

These properties provide additional information required to interpret JSON data as rich multimedia documents. They describe the type of content, how it is encoded, and/or how it may be validated. They do not function as validation assertions; a malformed string-encoded document MUST NOT cause the containing instance to be considered invalid.

Meta-schemas that do not use "$vocabulary" SHOULD be considered to require this vocabulary as if its URI were present with a value of true.

The current URI for this vocabulary, known as the Content vocabulary, is: <https://json-schema.org/draft/2019-09/vocab/content>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2019-09/meta/content>.

8.2. Implementation Requirements

Due to security and performance concerns, as well as the open-ended nature of possible content types, implementations MUST NOT automatically decode, parse, and/or validate the string contents by default. This additionally supports the use case of embedded documents intended for processing by a different consumer than that which processed the containing document.

All keywords in this section apply only to strings, and have no effect on other data types.

Implementations MAY offer the ability to decode, parse, and/or validate the string contents automatically. However, it MUST NOT perform these operations by default, and MUST provide the validation result of each string-encoded document separately from the enclosing document. This process SHOULD be equivalent to fully evaluating the instance against the original schema, followed by using the annotations to decode, parse, and/or validate each string-encoded

document. [[CREF7: For now, the exact mechanism of performing and returning parsed data and/or validation results from such an automatic decoding, parsing, and validating feature is left unspecified. Should such a feature prove popular, it may be specified more thoroughly in a future draft.]]

See also the Security Considerations (Section 10) sections for possible vulnerabilities introduced by automatically processing the instance string according to these keywords.

8.3. contentEncoding

If the instance value is a string, this property defines that the string SHOULD be interpreted as binary data and decoded using the encoding named by this property.

Possible values for this property are listed in RFC 2045, Sec 6.1 [RFC2045] and RFC 4648 [RFC4648]. For "base64", which is defined in both RFCs, the definition in RFC 4648, which removes line length limitations, SHOULD be used, as various other specifications have mandated different lengths. Note that line lengths within a string can be constrained using the "pattern" (Section 6.3.3) keyword.

If this keyword is absent, but "contentMediaType" is present, this indicates that the media type could be encoded into UTF-8 like any other JSON string value, and does not require additional decoding.

The value of this property MUST be a string.

8.4. contentMediaType

If the instance is a string, this property indicates the media type of the contents of the string. If "contentEncoding" is present, this property describes the decoded string.

The value of this property MUST be a string, which MUST be a media type, as defined by RFC 2046 [RFC2046].

8.5. contentSchema

If the instance is a string, and if "contentMediaType" is present, this property contains a schema which describes the structure of the string.

This keyword MAY be used with any media type that can be mapped into JSON Schema's data model.

The value of this property SHOULD be ignored if "contentMediaType" is not present.

8.6. Example

Here is an example schema, illustrating the use of "contentEncoding" and "contentMediaType":

```
{
    "type": "string",
    "contentEncoding": "base64",
    "contentMediaType": "image/png"
}
```

Instances described by this schema are expected to be strings, and their values should be interpretable as base64-encoded PNG images.

Another example:

```
{
    "type": "string",
    "contentMediaType": "text/html"
}
```

Instances described by this schema are expected to be strings containing HTML, using whatever character set the JSON string was decoded into. Per section 8.1 of RFC 8259 [RFC8259], outside of an entirely closed system, this MUST be UTF-8.

This example describes a JWT that is MACed using the HMAC SHA-256 algorithm, and requires the "iss" and "exp" fields in its claim set.

```
{
    "type": "string",
    "contentMediaType": "application/jwt",
    "contentSchema": {
        "type": "array",
        "minItems": 2,
        "items": [
            {
                "const": {
                    "typ": "JWT",
                    "alg": "HS256"
                }
            },
            {
                "type": "object",
                "required": ["iss", "exp"],
                "properties": {
                    "iss": {"type": "string"},
                    "exp": {"type": "integer"}
                }
            }
        ]
    }
}
```

Note that "contentEncoding" does not appear. While the "application/jwt" media type makes use of base64url encoding, that is defined by the media type, which determines how the JWT string is decoded into a list of two JSON data structures: first the header, and then the payload. Since the JWT media type ensures that the JWT can be represented in a JSON string, there is no need for further encoding or decoding.

9. A Vocabulary for Basic Meta-Data Annotations

These general-purpose annotation keywords provide commonly used information for documentation and user interface display purposes. They are not intended to form a comprehensive set of features. Rather, additional vocabularies can be defined for more complex annotation-based applications.

Meta-schemas that do not use "$vocabulary" SHOULD be considered to require this vocabulary as if its URI were present with a value of true.

The current URI for this vocabulary, known as the Meta-Data vocabulary, is: <https://json-schema.org/draft/2019-09/vocab/meta-data>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2019-09/meta/meta-data>.

9.1. "title" and "description"

The value of both of these keywords MUST be a string.

Both of these keywords can be used to decorate a user interface with information about the data produced by this user interface. A title will preferably be short, whereas a description will provide explanation about the purpose of the instance described by this schema.

9.2. "default"

There are no restrictions placed on the value of this keyword. When multiple occurrences of this keyword are applicable to a single sub-instance, implementations SHOULD remove duplicates.

This keyword can be used to supply a default JSON value associated with a particular schema. It is RECOMMENDED that a default value be valid against the associated schema.

9.3. "deprecated"

The value of this keyword MUST be a boolean. When multiple occurrences of this keyword are applicable to a single sub-instance, applications SHOULD consider the instance location to be deprecated if any occurrence specifies a true value.

If "deprecated" has a value of boolean true, it indicates that applications SHOULD refrain from usage of the declared property. It MAY mean the property is going to be removed in the future.

A root schema containing "deprecated" with a value of true indicates that the entire resource being described MAY be removed in the future.

When the "deprecated" keyword is applied to an item in an array by means of "items", if "items" is a single schema, the deprecation relates to the whole array, while if "items" is an array of schemas, the deprecation relates to the corrosponding item according to the subschemas position.

Omitting this keyword has the same behavior as a value of false.

9.4. "readOnly" and "writeOnly"

The value of these keywords MUST be a boolean. When multiple occurrences of these keywords are applicable to a single sub-instance, the resulting behavior SHOULD be as for a true value if any occurrence specifies a true value, and SHOULD be as for a false value otherwise.

If "readOnly" has a value of boolean true, it indicates that the value of the instance is managed exclusively by the owning authority, and attempts by an application to modify the value of this property are expected to be ignored or rejected by that owning authority.

An instance document that is marked as "readOnly for the entire document MAY be ignored if sent to the owning authority, or MAY result in an error, at the authority's discretion.

If "writeOnly" has a value of boolean true, it indicates that the value is never present when the instance is retrieved from the owning authority. It can be present when sent to the owning authority to update or create the document (or the resource it represents), but it will not be included in any updated or newly created version of the instance.

An instance document that is marked as "writeOnly" for the entire document MAY be returned as a blank document of some sort, or MAY produce an error upon retrieval, or have the retrieval request ignored, at the authority's discretion.

For example, "readOnly" would be used to mark a database-generated serial number as read-only, while "writeOnly" would be used to mark a password input field.

These keywords can be used to assist in user interface instance generation. In particular, an application MAY choose to use a widget that hides input values as they are typed for write-only fields.

Omitting these keywords has the same behavior as values of false.

9.5. "examples"

The value of this keyword MUST be an array. There are no restrictions placed on the values within the array. When multiple occurrences of this keyword are applicable to a single sub-instance, implementations MUST provide a flat array of all values rather than an array of arrays.

This keyword can be used to provide sample JSON values associated with a particular schema, for the purpose of illustrating usage. It is RECOMMENDED that these values be valid against the associated schema.

Implementations MAY use the value(s) of "default", if present, as an additional example. If "examples" is absent, "default" MAY still be used in this manner.

10. Security Considerations

JSON Schema validation defines a vocabulary for JSON Schema core and concerns all the security considerations listed there.

JSON Schema validation allows the use of Regular Expressions, which have numerous different (often incompatible) implementations. Some implementations allow the embedding of arbitrary code, which is outside the scope of JSON Schema and MUST NOT be permitted. Regular expressions can often also be crafted to be extremely expensive to compute (with so-called "catastrophic backtracking"), resulting in a denial-of-service attack.

Implementations that support validating or otherwise evaluating instance string data based on "contentEncoding" and/or "contentMediaType" are at risk of evaluating data in an unsafe way based on misleading information. Applications can mitigate this risk by only performing such processing when a relationship between the schema and instance is established (e.g., they share the same authority).

Processing a media type or encoding is subject to the security considerations of that media type or encoding. For example, the security considerations of RFC 4329 Scripting Media Types [RFC4329] apply when processing JavaScript or ECMAScript encoded within a JSON string.

11. References

11.1. Normative References

[ecma262] "ECMA 262 specification", <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.

[json-schema] Wright, A. and H. Andrews, "JSON Schema: A Media Type for Describing JSON Documents", draft-handrews-json-schema-02 (work in progress), November 2017.

[relative-json-pointer]
Luff, G. and H. Andrews, "Relative JSON Pointers", draft-handrews-relative-json-pointer-01 (work in progress), November 2017.

[RFC1123] Braden, R., Ed., "Requirements for Internet Hosts - Application and Support", STD 3, RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/info/rfc1123>.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <https://www.rfc-editor.org/info/rfc2045>.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[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>.

[RFC2673] Crawford, M., "Binary Labels in the Domain Name System", RFC 2673, DOI 10.17487/RFC2673, August 1999, <https://www.rfc-editor.org/info/rfc2673>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[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>.

[RFC3987] Duerst, M. and M. Suignard, "Internationalized Resource Identifiers (IRIs)", RFC 3987, DOI 10.17487/RFC3987, January 2005, <https://www.rfc-editor.org/info/rfc3987>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[RFC5891] Klensin, J., "Internationalized Domain Names in Applications (IDNA): Protocol", RFC 5891, DOI 10.17487/RFC5891, August 2010, <https://www.rfc-editor.org/info/rfc5891>.

[RFC6531] Yao, J. and W. Mao, "SMTP Extension for Internationalized Email", RFC 6531, DOI 10.17487/RFC6531, February 2012, <https://www.rfc-editor.org/info/rfc6531>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/info/rfc6570>.

[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>.

11.2. Informative References

[RFC4329] Hoehrmann, B., "Scripting Media Types", RFC 4329, DOI 10.17487/RFC4329, April 2006, <https://www.rfc-editor.org/info/rfc4329>.

Appendix A. Keywords Moved from Validation to Core

Several keywords have been moved from this document into the Core Specification [json-schema] as of this draft, in some cases with re-naming or other changes. This affects the following former validation keywords:

"definitions" Renamed to "$defs" to match "$ref" and be shorter to type. Schema vocabulary authors SHOULD NOT define a "definitions" keyword with different behavior in order to avoid invalidating schemas that still use the older name. While "definitions" is absent in the single-vocabulary meta-schemas referenced by this document, it remains present in the default meta-schema, and implementations SHOULD assume that "$defs" and "definitions" have the same behavior when that meta-schema is used.

"allOf", "anyOf", "oneOf", "not", "if", "then", "else",
"items", "additionalItems",
"contains", "propertyNames",
"properties", "patternProperties", "additionalProperties"
All of these keywords apply subschemas to the instance and combine their results, without asserting any conditions of their own. Without assertion keywords, these applicators can only cause assertion failures by using the false boolean schema, or by inverting the result of the true boolean schema (or equivalent schema objects). For this reason, they are better defined as a generic mechanism on which validation, hyper-schema, and extension vocabularies can all be based.

"dependencies" This keyword had two different modes of behavior, which made it relatively challenging to implement and reason about. The schema form has been moved to Core and renamed to "dependentSchemas", as part of the applicator vocabulary. It is analogous to "properties", except that instead of applying its subschema to the property value, it applies it to the object containing the property. The property name array form is retained here and renamed to "dependentRequired", as it is an assertion which is a shortcut for the conditional use of the "required" assertion keyword.

Appendix B. Acknowledgments

Thanks to Gary Court, Francis Galiegue, Kris Zyp, and Geraint Luff for their work on the initial drafts of JSON Schema.

Thanks to Jason Desrosiers, Daniel Perrett, Erik Wilde, Ben Hutton, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave

Finlay, and Denis Laxalde for their submissions and patches to the document.

Appendix C. ChangeLog

[[CREF8: This section to be removed before leaving Internet-Draft status.]]

draft-handrews-json-schema-validation-02

* Grouped keywords into formal vocabuarlies
* Update "format" implementation requirements in terms of vocabularies
* By default, "format" MUST NOT be validated, although validation can be enabled
* A vocabulary declaration can be used to require "format" validation
* Moved "definitions" to the core spec as "$defs"
* Moved applicator keywords to the core spec
* Renamed the array form of "dependencies" to "dependentRequired", moved the schema form to the core spec
* Specified all "content*" keywords as annotations, not assertions
* Added "contentSchema" to allow applying a schema to a string-encoded document
* Also allow RFC 4648 encodings in "contentEncoding"
* Added "minContains" and "maxContains"
* Update RFC reference for "hostname" and "idn-hostname"
* Add "uuid" and "duration" formats

draft-handrews-json-schema-validation-01

* This draft is purely a clarification with no functional changes
* Provided the general principle behind ignoring annotations under "not" and similar cases

* Clarified "if"/"then"/"else" validation interactions

* Clarified "if"/"then"/"else" behavior for annotation

* Minor formatting and cross-referencing improvements

draft-handrews-json-schema-validation-00

* Added "if"/"then"/"else"

* Classify keywords as assertions or annotations per the core spec

* Warn of possibly removing "dependencies" in the future

* Grouped validation keywords into sub-sections for readability

* Moved "readOnly" from hyper-schema to validation meta-data

* Added "writeOnly"

* Added string-encoded media section, with former hyper-schema "media" keywords

* Restored "regex" format (removal was unintentional)

* Added "date" and "time" formats, and reserved additional RFC 3339 format names

* I18N formats: "iri", "iri-reference", "idn-hostname", "idn-email"

* Clarify that "json-pointer" format means string encoding, not URI fragment

* Fixed typo that inverted the meaning of "minimum" and "exclusiveMinimum"

* Move format syntax references into Normative References

* JSON is a normative requirement

draft-wright-json-schema-validation-01

* Standardized on hyphenated format names with full words ("uriref" becomes "uri-reference")

* Add the formats "uri-template" and "json-pointer"

* Changed "exclusiveMaximum"/"exclusiveMinimum" from boolean modifiers of "maximum"/"minimum" to independent numeric fields.

* Split the additionalItems/items into two sections

* Reworked properties/patternProperties/additionalProperties definition

* Added "examples" keyword

* Added "contains" keyword

* Allow empty "required" and "dependencies" arrays

* Fixed "type" reference to primitive types

* Added "const" keyword

* Added "propertyNames" keyword

draft-wright-json-schema-validation-00

* Added additional security considerations

* Removed reference to "latest version" meta-schema, use numbered version instead

* Rephrased many keyword definitions for brevity

* Added "uriref" format that also allows relative URI references

draft-fge-json-schema-validation-00

* Initial draft.

* Salvaged from draft v3.

* Redefine the "required" keyword.

* Remove "extends", "disallow"

* Add "anyOf", "allOf", "oneOf", "not", "definitions", "minProperties", "maxProperties".

* "dependencies" member values can no longer be single strings; at least one element is required in a property dependency array.

* Rename "divisibleBy" to "multipleOf".

* "type" arrays can no longer have schemas; remove "any" as a possible value.

* Rework the "format" section; make support optional.

* "format": remove attributes "phone", "style", "color"; rename "ip-address" to "ipv4"; add references for all attributes.

* Provide algorithms to calculate schema(s) for array/object instances.

* Add interoperability considerations.

Authors' Addresses

Austin Wright (editor)

EMail: aaa@bzfx.net

Henry Andrews (editor)

EMail: andrews_henry@yahoo.com

Ben Hutton (editor)
Wellcome Sanger Institute

EMail: bh7@sanger.ac.uk
URI: https://jsonschema.dev