Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: September 20, 2018

A. Wright, Ed.

H. Andrews, Ed.
Cloudflare, Inc.

G. Luff

March 19, 2018

JSON Schema Validation: A Vocabulary for Structural Validation of JSON
draft-handrews-json-schema-validation-01

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 <http://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

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 September 20, 2018.

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 applies schemas to locations within the instance, and asserts constraints on the structure of the data at each location. 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.

3.1. Applicability

Validation begins by applying the root schema to the complete instance document. From there, various keywords are used to determine which additional subschemas are applied to either the current location, or a child location. These keywords also define whether and how subschema assertion results are modified and/or combined. Such keywords do not assert conditions on their own. Rather, they control how assertions are applied and evaluated.

The keywords in the boolean logic (Section 6.7) and conditional (Section 6.6) sections of this specification apply subschemas to the same location as the parent schema. The former group defines boolean operations on the subschema assertion results, while the latter evaluates one subschema and uses its assertion results to determine which of two other subschemas to apply as well.

Several keywords determine which subschemas are applied to array items, object property values, and object property names. They are: "items", "additionalItems", "contains", "properties", "patternProperties", "additionalProperties", and "propertyNames". The "contains" keyword only requires its subschema to be valid against at least one child instance, while the other keywords require that all subschemas are valid against all child instances to which they apply.

3.1.1. Keyword Independence

Validation keywords typically operate independently, without affecting each other’s outcomes.

For schema author convenience, there are some exceptions among the keywords that control subschema applicability:

- "additionalProperties", whose behavior is defined in terms of "properties" and "patternProperties"; and

- "additionalItems", whose behavior is defined in terms of "items".

3.2. Assertions

Validation is a process of checking assertions. Each assertion adds constraints that an instance must satisfy in order to successfully validate.

Assertion keywords that are absent never restrict validation. In some cases, this no-op behavior is identical to a keyword that exists with certain values, and these values are noted where known.

All of the keywords in the general (Section 6.1), numeric (Section 6.2), and string (Section 6.3) sections are assertions, as well as "minItems", "maxItems", "uniqueItems", "minProperties", "maxProperties", and "required". Additionally, "dependencies" is shorthand for a combination of conditional and assertion keywords.

The "format", "contentType", and "contentEncoding" keywords can also be implemented as assertions, although that functionality is an optional part of this specification, and the keywords convey additional non-assertion information.

3.2.1. Assertions and Instance Primitive Types

Most validation assertions only constrain values within a certain primitive type. When the type of the instance is not of the type targeted by the keyword, the instance is considered to conform to the assertion.

For example, the "maxLength" keyword will only restrict certain strings (that are too long) from being valid. If the instance is a number, boolean, null, array, or object, then it is valid against this assertion.

3.3. Annotations

In addition to assertions, this specification provides a small vocabulary of metadata keywords that can be used to annotate the JSON instance with useful information. The Section 7 and Section 8 keywords are also useful as annotations as well as being optional assertions, as they convey additional usage guidance for the instance data.

A schema that is applicable to a particular location in the instance, against which the instance location is valid, attaches its annotations to that location in the instance. Since many subschemas can be applicable to any single location, annotation keywords need to specify any unusual handling of multiple applicable occurrences of the keyword with different values. The default behavior is simply to collect all values.

Additional vocabularies SHOULD make use of this mechanism for applying their own annotations to instances.

3.3.1. Annotations and Validation Outcomes

Annotations are collected whenever an instance is valid against a schema object, and all of that schema object's parent schemas.

In particular, annotations in a subschema contained within a "not", at any depth, including any number of intervening additional "not" subschemas, MUST be ignored. If the instance was valid against the "not" subschema, then by definition it is not valid against the schema that contains the "not", so the "not" subschema's annotations are not used.

Similarly, annotations within a failing branch of a "oneOf", "anyOf", "then", or "else" MUST be ignored even when the instance successfully validates against the complete schema document.

3.3.2. Annotations and Short-Circuit Validation

Annotation keywords MUST be applied to all possible sub-instances. Even if such application can be short-circuited when only assertion evaluation is needed. For instance, the "contains" keyword need only be checked for assertions until at least one array item proves valid. However, when working with annotations, all items in the array must be evaluated to determine all items with which the annotations should be associated.

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

Two validation keywords, "pattern" and "patternProperties", use regular expressions to express constraints, and the "regex" value for the "format" keyword constrains the instance value to be a regular

expression. These regular expressions SHOULD be valid according to the ECMA 262 [ecma262] regular expression dialect.

Furthermore, given the high disparity in regular expression constructs support, schema authors SHOULD limit themselves to the following regular expression tokens:

- individual Unicode characters, as defined by the JSON specification [RFC7159];
- simple character classes ([abc]), range character classes ([a-z]);
- complemented character classes ([^abc], [^a-z]);
- simple quantifiers: "+" (one or more), "*" (zero or more), "?" (zero or one), and their lazy versions ("+?", "*?", "??");
- range quantifiers: "{x}" (exactly x occurrences), "{x,y}" (at least x, at most y, occurrences), {x,} (x occurrences or more), and their lazy versions;
- the beginning-of-input ("^") and end-of-input ("$") anchors;
- simple grouping ("(...)") and alternation ("|").

Finally, implementations MUST NOT take regular expressions to be anchored, neither at the beginning nor at the end. This means, for instance, the pattern "es" matches "expression".

5. Meta-Schema

The current URI for the JSON Schema Validation is <http://json-schema.org/draft-07/schema#>.

6. Validation Keywords

Validation keywords in a schema impose requirements for successful validation of an instance.

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 value, including null.

6.1.3. const

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

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 7159 [RFC7159].

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 7159 [RFC7159].

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. items

The value of "items" MUST be either a valid JSON Schema or an array of valid JSON Schemas.

This keyword determines how child instances validate for arrays, and does not directly validate the immediate instance itself.

If "items" is a schema, validation succeeds if all elements in the array successfully validate against that schema.

If "items" is an array of schemas, validation succeeds if each element of the instance validates against the schema at the same position, if any.

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

6.4.2. additionalItems

The value of "additionalItems" MUST be a valid JSON Schema.

This keyword determines how child instances validate for arrays, and does not directly validate the immediate instance itself.

If "items" is an array of schemas, validation succeeds if every instance element at a position greater than the size of "items" validates against "additionalItems".

Otherwise, "additionalItems" MUST be ignored, as the "items" schema (possibly the default value of an empty schema) is applied to all elements.

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

6.4.3. 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.

6.4.4. 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.

6.4.5. 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.

6.4.6. contains

The value of this keyword MUST be a valid JSON Schema.

An array instance is valid against "contains" if at least one of its elements is valid against the given schema.

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. properties

The value of "properties" MUST be an object. Each value of this object MUST be a valid JSON Schema.

This keyword determines how child instances validate for objects, and does not directly validate the immediate instance itself.

Validation succeeds if, for each name that appears in both the instance and as a name within this keyword’s value, the child instance for that name successfully validates against the corresponding schema.

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

6.5.5. patternProperties

The value of "patternProperties" MUST be an object. Each property name of this object SHOULD be a valid regular expression, according to the ECMA 262 regular expression dialect. Each property value of this object MUST be a valid JSON Schema.

This keyword determines how child instances validate for objects, and does not directly validate the immediate instance itself. Validation of the primitive instance type against this keyword always succeeds.

Validation succeeds if, for each instance name that matches any regular expressions that appear as a property name in this keyword’s value, the child instance for that name successfully validates against each schema that corresponds to a matching regular expression.

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

6.5.6. additionalProperties

The value of "additionalProperties" MUST be a valid JSON Schema.

This keyword determines how child instances validate for objects, and does not directly validate the immediate instance itself.

Validation with "additionalProperties" applies only to the child values of instance names that do not match any names in "properties", and do not match any regular expression in "patternProperties".

For all such properties, validation succeeds if the child instance validates against the "additionalProperties" schema.

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

6.5.7. dependencies

[[CREF1: This keyword may be split into two, with the variation that uses an array of property names rather than a subschema getting a new name. The dual behavior is confusing and relatively difficult to implement. In the previous draft, we proposed dropping the keyword altogether, or dropping one of its forms, but we received feedback in support of keeping it. See issues #442 and #528 at <https://github.com/json-schema-org/json-schema-spec/issues> for further discussion. Further feedback is encouraged.]]

This keyword specifies rules that are evaluated if the instance is an object and contains a certain property.

This keyword's value MUST be an object. Each property specifies a dependency. Each dependency value MUST be an array or a valid JSON Schema.

If the dependency value is a subschema, and the dependency key is a property in the instance, the entire instance must validate against the dependency value.

If the dependency value is an array, each element in the array, if any, MUST be a string, and MUST be unique. If the dependency key is a property in the instance, each of the items in the dependency value must be a property that exists in the instance.

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

6.5.8. propertyNames

The value of "propertyNames" MUST be a valid JSON Schema.

If the instance is an object, this keyword validates if every property name in the instance validates against the provided schema. Note the property name that the schema is testing will always be a string.

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

6.6. Keywords for Applying Subschemas Conditionally

These keywords work together to implement conditional application of a subschema based on the outcome of another subschema.

These keywords MUST NOT interact with each other across subschema boundaries. In other words, an "if" in one branch of an "allOf" MUST NOT have an impact on a "then" or "else" in another branch.

There is no default behavior for any of these keywords when they are not present. In particular, they MUST NOT be treated as if present with an empty schema, and when "if" is not present, both "then" and "else" MUST be entirely ignored.

6.6.1. if

This keyword’s value MUST be a valid JSON Schema.

This validation outcome of this keyword’s subschema has no direct effect on the overall validation result. Rather, it controls which of the "then" or "else" keywords are evaluated.

Instances that successfully validate against this keyword’s subschema MUST also be valid against the subschema value of the "then" keyword, if present.

Instances that fail to validate against this keyword’s subschema MUST also be valid against the subschema value of the "else" keyword, if present.

If annotations (Section 3.3) are being collected, they are collected from this keyword’s subschema in the usual way, including when the keyword is present without either "then" or "else".

6.6.2. then

This keyword's value MUST be a valid JSON Schema.

When "if" is present, and the instance successfully validates against its subschema, then valiation succeeds against this keyword if the instance also successfully validates against this keyword's subschema.

This keyword has no effect when "if" is absent, or when the instance fails to validate against its subschema. Implementations MUST NOT evaluate the instance against this keyword, for either validation or annotation collection purposes, in such cases.

6.6.3. else

This keyword's value MUST be a valid JSON Schema.

When "if" is present, and the instance fails to validate against its subschema, then valiation succeeds against this keyword if the instance successfully validates against this keyword's subschema.

This keyword has no effect when "if" is absent, or when the instance successfully validates against its subschema. Implementations MUST NOT evaluate the instance against this keyword, for either validation or annotation collection purposes, in such cases.

6.7. Keywords for Applying Subschemas With Boolean Logic

6.7.1. allOf

This keyword's value MUST be a non-empty array. Each item of the array MUST be a valid JSON Schema.

An instance validates successfully against this keyword if it validates successfully against all schemas defined by this keyword's value.

6.7.2. anyOf

This keyword's value MUST be a non-empty array. Each item of the array MUST be a valid JSON Schema.

An instance validates successfully against this keyword if it validates successfully against at least one schema defined by this keyword's value.

6.7.3. oneOf

This keyword's value MUST be a non-empty array. Each item of the array MUST be a valid JSON Schema.

An instance validates successfully against this keyword if it validates successfully against exactly one schema defined by this keyword's value.

6.7.4. not

This keyword's value MUST be a valid JSON Schema.

An instance is valid against this keyword if it fails to validate successfully against the schema defined by this keyword.

7. Semantic Validation With "format"

7.1. Foreword

Structural validation alone may be insufficient to validate that an instance meets all the requirements of an application. The "format" keyword is defined to allow interoperable semantic validation for a fixed subset of values which are accurately described by authoritative resources, be they RFCs or other external specifications.

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.

7.2. Implementation Requirements

The "format" keyword functions as both an annotation (Section 3.3) and as an assertion (Section 3.2). While no special effort is required to implement it as an annotation conveying semantic meaning, implementing validation is non-trivial.

Implementations MAY support the "format" keyword as a validation assertion. Should they choose to do so:

- they SHOULD implement validation for attributes defined below;
- they SHOULD offer an option to disable validation for this keyword.

Implementations MAY add custom format attributes. Save for agreement between parties, schema authors SHALL NOT expect a peer implementation to support this keyword and/or custom format attributes.

7.3. Defined Formats

7.3.1. Dates and Times

These attributes apply to string instances.

Date and time format names are derived from RFC 3339, section 5.6 [RFC3339].

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.

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. [[CREF2: 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 1034, section 3.1 [RFC1034], including host names produced using the Punycode algorithm specified in RFC 5891, section 4.4 [RFC5891].

idn-hostname: As defined by either RFC 1034 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].

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

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. String-Encoding Non-JSON Data

8.1. Foreword

Properties defined in this section indicate that an instance contains non-JSON data encoded in a JSON string. They describe the type of content and how it is encoded.

These properties provide additional information required to interpret JSON data as rich multimedia documents.

8.2. Implementation Requirements

The content keywords function as both annotations (Section 3.3) and as assertions (Section 3.2). While no special effort is required to implement them as annotations conveying how applications can interpret the data in the string, implementing validation of conformance to the media type and encoding is non-trivial.

Implementations MAY support the "contentMediaType" and "contentEncoding" keywords as validation assertions. Should they choose to do so, they SHOULD offer an option to disable validation for 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. RFC 2045, Sec 6.1 [RFC2045] lists the possible values for this property.

The value of this property MUST be a string.

The value of this property SHOULD be ignored if the instance described is not a string.

8.4. contentMediaType

The value of this property must be a media type, as defined by RFC 2046 [RFC2046]. This property defines the media type of instances which this schema defines.

The value of this property MUST be a string.

The value of this property SHOULD be ignored if the instance described is not a string.

If the "contentEncoding" property is not present, but the instance value is a string, then the value of this property SHOULD specify a text document type, and the character set SHOULD be the character set into which the JSON string value was decoded (for which the default is Unicode).

8.5. 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 should 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 should be strings containing HTML, using whatever character set the JSON string was decoded into (default is Unicode).

9. Schema Re-Use With "definitions"

The "definitions" keywords provides a standardized location for schema authors to inline re-usable JSON Schemas into a more general schema. The keyword does not directly affect the validation result.

This keyword’s value MUST be an object. Each member value of this object MUST be a valid JSON Schema.

As an example, here is a schema describing an array of positive integers, where the positive integer constraint is a subschema in "definitions":

```
{
    "type": "array",
    "items": { "$ref": "#/definitions/positiveInteger" },
    "definitions": {
        "positiveInteger": {
            "type": "integer",
            "exclusiveMinimum": 0
        }
    }
}
```

10. Schema Annotations

Schema validation is a useful mechanism for annotating instance data with additional information. The rules for determining when and how annotations are associated with an instance are outlined in section 3.3.

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.

10.1. "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.

10.2. "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.

10.3. "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 value MUST be true if any occurrence specifies a true value, and MUST be false 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.

10.4. "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.

11. 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.

12. References

12.1. 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-01 (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.

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[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>.

[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>.

[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>.

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <https://www.rfc-editor.org/info/rfc7159>.

12.2. 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. 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 B. ChangeLog

[[CREF3: This section to be removed before leaving Internet-Draft status.]]

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)
Cloudflare, Inc.
San Francisco, CA
USA

EMail: henry@cloudflare.com

Geraint Luff
Cambridge
UK

EMail: luffgd@gmail.com