Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: June 11, 2021

A. Wright, Ed.

H. Andrews, Ed.

B. Hutton, Ed.

G. Dennis

December 8, 2020

JSON Schema: A Media Type for Describing JSON Documents
draft-bhutton-json-schema-00

Abstract

JSON Schema defines the media type "application/schema+json", a JSON-based format for describing the structure of JSON data. JSON Schema asserts what a JSON document must look like, ways to extract information from it, and how to interact with it. The "application/schema-instance+json" media type provides additional feature-rich integration with "application/schema+json" beyond what can be offered for "application/json" documents.

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

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 June 11, 2021.

Copyright Notice

Table of Contents

1. Introduction

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

This specification defines JSON Schema core terminology and mechanisms, including pointing to another JSON Schema by reference, dereferencing a JSON Schema reference, specifying the dialect being used, specifying a dialect's vocabulary requirements, and defining the expected output.

Other specifications define the vocabularies that perform assertions about validation, linking, annotation, navigation, and interaction.

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].

The terms "JSON", "JSON text", "JSON value", "member", "element", "object", "array", "number", "string", "boolean", "true", "false", and "null" in this document are to be interpreted as defined in RFC 8259 [RFC8259].

3. Overview

This document proposes a new media type "application/schema+json" to identify a JSON Schema for describing JSON data. It also proposes a further optional media type, "application/schema-instance+json", to provide additional integration features. JSON Schemas are themselves JSON documents. This, and related specifications, define keywords allowing authors to describe JSON data in several ways.

JSON Schema uses keywords to assert constraints on JSON instances or annotate those instances with additional information. Additional keywords are used to apply assertions and annotations to more complex JSON data structures, or based on some sort of condition.

To facilitate re-use, keywords can be organized into vocabularies. A vocabulary consists of a list of keywords, together with their syntax and semantics. A dialect is defined as a set of vocabularies and their required support identified in a meta-schema.

JSON Schema can be extended either by defining additional vocabularies, or less formally by defining additional keywords outside of any vocabulary. Unrecognized individual keywords simply have their values collected as annotations, while the behavior with respect to an unrecognized vocabulary can be controlled when declaring which vocabularies are in use.

This document defines a core vocabulary that MUST be supported by any implementation, and cannot be disabled. Its keywords are each prefixed with a "$" character to emphasize their required nature. This vocabulary is essential to the functioning of the "application/schema+json" media type, and is used to bootstrap the loading of other vocabularies.

Additionally, this document defines a RECOMMENDED vocabulary of keywords for applying subschemas conditionally, and for applying subschemas to the contents of objects and arrays. Either this vocabulary or one very much like it is required to write schemas for

non-trivial JSON instances, whether those schemas are intended for assertion validation, annotation, or both. While not part of the required core vocabulary, for maximum interoperability this additional vocabulary is included in this document and its use is strongly encouraged.

Further vocabularies for purposes such as structural validation or hypermedia annotation are defined in other documents. These other documents each define a dialect collecting the standard sets of vocabularies needed to write schemas for that document's purpose.

4. Definitions

4.1. JSON Document

A JSON document is an information resource (series of octets) described by the application/json media type.

In JSON Schema, the terms "JSON document", "JSON text", and "JSON value" are interchangeable because of the data model it defines.

JSON Schema is only defined over JSON documents. However, any document or memory structure that can be parsed into or processed according to the JSON Schema data model can be interpreted against a JSON Schema, including media types like CBOR [RFC7049].

4.2. Instance

A JSON document to which a schema is applied is known as an "instance".

JSON Schema is defined over "application/json" or compatible documents, including media types with the "+json" structured syntax suffix.

Among these, this specification defines the "application/schema-instance+json" media type which defines handling for fragments in the URI.

4.2.1. Instance Data Model

JSON Schema interprets documents according to a data model. A JSON value interpreted according to this data model is called an "instance".

An instance has one of six primitive types, and a range of possible values depending on the type:

null: A JSON "null" value

boolean: A "true" or "false" value, from the JSON "true" or "false" value

object: An unordered set of properties mapping a string to an instance, from the JSON "object" value

array: An ordered list of instances, from the JSON "array" value

number: An arbitrary-precision, base-10 decimal number value, from the JSON "number" value

string: A string of Unicode code points, from the JSON "string" value

Whitespace and formatting concerns, including different lexical representations of numbers that are equal within the data model, are thus outside the scope of JSON Schema. JSON Schema vocabularies (Section 8.1) that wish to work with such differences in lexical representations SHOULD define keywords to precisely interpret formatted strings within the data model rather than relying on having the original JSON representation Unicode characters available.

Since an object cannot have two properties with the same key, behavior for a JSON document that tries to define two properties with the same key in a single object is undefined.

Note that JSON Schema vocabularies are free to define their own extended type system. This should not be confused with the core data model types defined here. As an example, "integer" is a reasonable type for a vocabulary to define as a value for a keyword, but the data model makes no distinction between integers and other numbers.

4.2.2. Instance Equality

Two JSON instances are said to be equal if and only if they are of the same type and have the same value according to the data model. Specifically, this means:

both are null; or

both are true; or

both are false; or

both are strings, and are the same codepoint-for-codepoint; or

both are numbers, and have the same mathematical value; or

both are arrays, and have an equal value item-for-item; or

both are objects, and each property in one has exactly one property with a key equal to the other's, and that other property has an equal value.

Implied in this definition is that arrays must be the same length, objects must have the same number of members, properties in objects are unordered, there is no way to define multiple properties with the same key, and mere formatting differences (indentation, placement of commas, trailing zeros) are insignificant.

4.2.3. Non-JSON Instances

It is possible to use JSON Schema with a superset of the JSON Schema data model, where an instance may be outside any of the six JSON data types.

In this case, annotations still apply; but most validation keywords will not be useful, as they will always pass or always fail.

A custom vocabulary may define support for a superset of the core data model. The schema itself may only be expressible in this superset; for example, to make use of the "const" keyword.

4.3. JSON Schema Documents

A JSON Schema document, or simply a schema, is a JSON document used to describe an instance. A schema can itself be interpreted as an instance, but SHOULD always be given the media type "application/schema+json" rather than "application/schema-instance+json". The "application/schema+json" media type is defined to offer a superset of the fragment identifier syntax and semantics provided by "application/schema-instance+json".

A JSON Schema MUST be an object or a boolean.

4.3.1. JSON Schema Objects and Keywords

Object properties that are applied to the instance are called keywords, or schema keywords. Broadly speaking, keywords fall into one of five categories:

identifiers: control schema identification through setting the schema's canonical URI and/or changing how the base URI is determined

assertions: produce a boolean result when applied to an instance

annotations: attach information to an instance for application use

applicators: apply one or more subschemas to a particular location in the instance, and combine or modify their results

reserved locations: do not directly affect results, but reserve a place for a specific purpose to ensure interoperability

Keywords may fall into multiple categories, although applicators SHOULD only produce assertion results based on their subschemas' results. They should not define additional constraints independent of their subschemas.

Keywords which are properties within the same schema object are referred to as adjacent keywords.

Extension keywords, meaning those defined outside of this document and its companions, are free to define other behaviors as well.

A JSON Schema MAY contain properties which are not schema keywords. Unknown keywords SHOULD be treated as annotations, where the value of the keyword is the value of the annotation.

An empty schema is a JSON Schema with no properties, or only unknown properties.

4.3.2. Boolean JSON Schemas

The boolean schema values "true" and "false" are trivial schemas that always produce themselves as assertion results, regardless of the instance value. They never produce annotation results.

These boolean schemas exist to clarify schema author intent and facilitate schema processing optimizations. They behave identically to the following schema objects (where "not" is part of the subschema application vocabulary defined in this document).

true: Always passes validation, as if the empty schema {}

false: Always fails validation, as if the schema { "not": {} }

While the empty schema object is unambiguous, there are many possible equivalents to the "false" schema. Using the boolean values ensures that the intent is clear to both human readers and implementations.

4.3.3. Schema Vocabularies

A schema vocabulary, or simply a vocabulary, is a set of keywords, their syntax, and their semantics. A vocabulary is generally organized around a particular purpose. Different uses of JSON Schema, such as validation, hypermedia, or user interface generation, will involve different sets of vocabularies.

Vocabularies are the primary unit of re-use in JSON Schema, as schema authors can indicate what vocabularies are required or optional in order to process the schema. Since vocabularies are identified by URIs in the meta-schema, generic implementations can load extensions to support previously unknown vocabularies. While keywords can be supported outside of any vocabulary, there is no analogous mechanism to indicate individual keyword usage.

4.3.4. Meta-Schemas

A schema that itself describes a schema is called a meta-schema. Meta-schemas are used to validate JSON Schemas and specify which vocabularies they are using.

Typically, a meta-schema will specify a set of vocabularies, and validate schemas that conform to the syntax of those vocabularies. However, meta-schemas and vocabularies are separate in order to allow meta-schemas to validate schema conformance more strictly or more loosely than the vocabularies' specifications call for. Meta-schemas may also describe and validate additional keywords that are not part of a formal vocabulary.

4.3.5. Root Schema and Subschemas and Resources

A JSON Schema resource is a schema which is canonically [RFC6596] identified by an absolute URI [RFC3986].

The root schema is the schema that comprises the entire JSON document in question. The root schema is always a schema resource, where the URI is determined as described in section 9.1.1.

Some keywords take schemas themselves, allowing JSON Schemas to be nested:

```
{
    "title": "root",
    "items": {
        "title": "array item"
    }
}
```

In this example document, the schema titled "array item" is a subschema, and the schema titled "root" is the root schema.

As with the root schema, a subschema is either an object or a boolean.

As discussed in section 8.2.1, a JSON Schema document can contain multiple JSON Schema resources. When used without qualification, the term "root schema" refers to the document's root schema. In some cases, resource root schemas are discussed. A resource's root schema is its top-level schema object, which would also be a document root schema if the resource were to be extracted to a standalone JSON Schema document.

Whether multiple schema resources are embedded or linked with a reference, they are processed in the same way, with the same available behaviors.

5. Fragment Identifiers

In accordance with section 3.1 of RFC 6839 [RFC6839], the syntax and semantics of fragment identifiers specified for any +json media type SHOULD be as specified for "application/json". (At publication of this document, there is no fragment identification syntax defined for "application/json".)

Additionally, the "application/schema+json" media type supports two fragment identifier structures: plain names and JSON Pointers. The "application/schema-instance+json" media type supports one fragment identifier structure: JSON Pointers.

The use of JSON Pointers as URI fragment identifiers is described in RFC 6901 [RFC6901]. For "application/schema+json", which supports two fragment identifier syntaxes, fragment identifiers matching the JSON Pointer syntax, including the empty string, MUST be interpreted as JSON Pointer fragment identifiers.

Per the W3C's best practices for fragment identifiers [W3C.WD-fragid-best-practices-20121025], plain name fragment identifiers in "application/schema+json" are reserved for referencing

locally named schemas. All fragment identifiers that do not match the JSON Pointer syntax MUST be interpreted as plain name fragment identifiers.

Defining and referencing a plain name fragment identifier within an "application/schema+json" document are specified in the "$anchor" keyword (Section 8.2.2) section.

6. General Considerations

6.1. Range of JSON Values

An instance may be any valid JSON value as defined by JSON [RFC8259]. JSON Schema imposes no restrictions on type: JSON Schema can describe any JSON value, including, for example, null.

6.2. Programming Language Independence

JSON Schema is programming language agnostic, and supports the full range of values described in the data model. Be aware, however, that some languages and JSON parsers may not be able to represent in memory the full range of values describable by JSON.

6.3. Mathematical Integers

Some programming languages and parsers use different internal representations for floating point numbers than they do for integers.

For consistency, integer JSON numbers SHOULD NOT be encoded with a fractional part.

6.4. Regular Expressions

Keywords MAY use regular expressions to express constraints, or constrain the instance value to be a regular expression. These regular expressions SHOULD be valid according to the regular expression dialect described in ECMA-262, section 21.2.1 [ecma262].

Regular expressions SHOULD be built with the "u" flag (or equivalent) to provide Unicode support, or processed in such a way which provides Unicode support as defined by ECMA-262.

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 [RFC8259];

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".

6.5. Extending JSON Schema

Additional schema keywords and schema vocabularies MAY be defined by any entity. Save for explicit agreement, schema authors SHALL NOT expect these additional keywords and vocabularies to be supported by implementations that do not explicitly document such support. Implementations SHOULD treat keywords they do not support as annotations, where the value of the keyword is the value of the annotation.

Implementations MAY provide the ability to register or load handlers for vocabularies that they do not support directly. The exact mechanism for registering and implementing such handlers is implementation-dependent.

7. Keyword Behaviors

JSON Schema keywords fall into several general behavior categories. Assertions validate that an instance satisfies constraints, producing a boolean result. Annotations attach information that applications may use in any way they see fit. Applicators apply subschemas to parts of the instance and combine their results.

Extension keywords SHOULD stay within these categories, keeping in mind that annotations in particular are extremely flexible. Complex behavior is usually better delegated to applications on the basis of annotation data than implemented directly as schema keywords. However, extension keywords MAY define other behaviors for specialized purposes.

Evaluating an instance against a schema involves processing all of the keywords in the schema against the appropriate locations within the instance. Typically, applicator keywords are processed until a schema object with no applicators (and therefore no subschemas) is reached. The appropriate location in the instance is evaluated against the assertion and annotation keywords in the schema object, and their results are gathered into the parent schema according to the rules of the applicator.

Evaluation of a parent schema object can complete once all of its subschemas have been evaluated, although in some circumstances evaluation may be short-circuited due to assertion results. When annotations are being collected, some assertion result short-circuiting is not possible due to the need to examine all subschemas for annotation collection, including those that cannot further change the assertion result.

7.1. Lexical Scope and Dynamic Scope

While most JSON Schema keywords can be evaluated on their own, or at most need to take into account the values or results of adjacent keywords in the same schema object, a few have more complex behavior.

The lexical scope of a keyword is determined by the nested JSON data structure of objects and arrays. The largest such scope is an entire schema document. The smallest scope is a single schema object with no subschemas.

Keywords MAY be defined with a partial value, such as a URI-reference, which must be resolved against another value, such as another URI-reference or a full URI, which is found through the lexical structure of the JSON document. The "$id", "$ref", and "$dynamicRef" core keywords, and the "base" JSON Hyper-Schema keyword, are examples of this sort of behavior.

Note that some keywords, such as "$schema", apply to the lexical scope of the entire schema resource, and therefore MUST only appear in a schema resource's root schema.

Other keywords may take into account the dynamic scope that exists during the evaluation of a schema, typically together with an instance document. The outermost dynamic scope is the schema object at which processing begins, even if it is not a schema resource root. The path from this root schema to any particular keyword (that includes any "$ref" and "$dynamicRef" keywords that may have been resolved) is considered the keyword's "validation path."

Lexical and dynamic scopes align until a reference keyword is encountered. While following the reference keyword moves processing from one lexical scope into a different one, from the perspective of dynamic scope, following a reference is no different from descending into a subschema present as a value. A keyword on the far side of that reference that resolves information through the dynamic scope will consider the originating side of the reference to be their dynamic parent, rather than examining the local lexically enclosing parent.

The concept of dynamic scope is primarily used with "$dynamicRef" and "$dynamicAnchor", and should be considered an advanced feature and used with caution when defining additional keywords. It also appears when reporting errors and collected annotations, as it may be possible to revisit the same lexical scope repeatedly with different dynamic scopes. In such cases, it is important to inform the user of the dynamic path that produced the error or annotation.

7.2. Keyword Interactions

Keyword behavior MAY be defined in terms of the annotation results of subschemas (Section 4.3.5) and/or adjacent keywords (keywords within the same schema object) and their subschemas. Such keywords MUST NOT result in a circular dependency. Keywords MAY modify their behavior based on the presence or absence of another keyword in the same schema object (Section 4.3).

7.3. Default Behaviors

A missing keyword MUST NOT produce a false assertion result, MUST NOT produce annotation results, and MUST NOT cause any other schema to be evaluated as part of its own behavioral definition. However, given that missing keywords do not contribute annotations, the lack of annotation results may indirectly change the behavior of other keywords.

In some cases, the missing keyword assertion behavior of a keyword is identical to that produced by a certain value, and keyword definitions SHOULD note such values where known. However, even if the value which produces the default behavior would produce annotation results if present, the default behavior still MUST NOT result in annotations.

Because annotation collection can add significant cost in terms of both computation and memory, implementations MAY opt out of this feature. Keywords that are specified in terms of collected annotations SHOULD describe reasonable alternate approaches when

appropriate. This approach is demonstrated by the "items" and "additionalProperties" keywords in this document.

Note that when no such alternate approach is possible for a keyword, implementations that do not support annotation collections will not be able to support those keywords or vocabularies that contain them.

7.4. Identifiers

Identifiers set the canonical URI of a schema, or affect how such URIs are resolved in references (Section 8.2.3), or both. The Core vocabulary defined in this document defines several identifying keywords, most notably "$id".

Canonical schema URIs MUST NOT change while processing an instance, but keywords that affect URI-reference resolution MAY have behavior that is only fully determined at runtime.

While custom identifier keywords are possible, vocabulary designers should take care not to disrupt the functioning of core keywords. For example, the "$dynamicAnchor" keyword in this specification limits its URI resolution effects to the matching "$dynamicRef" keyword, leaving the behavior of "$ref" undisturbed.

7.5. Applicators

Applicators allow for building more complex schemas than can be accomplished with a single schema object. Evaluation of an instance against a schema document (Section 4.3) begins by applying the root schema (Section 4.3.5) to the complete instance document. From there, keywords known as applicators are used to determine which additional schemas are applied. Such schemas may be applied in-place to the current location, or to a child location.

The schemas to be applied may be present as subschemas comprising all or part of the keyword's value. Alternatively, an applicator may refer to a schema elsewhere in the same schema document, or in a different one. The mechanism for identifying such referenced schemas is defined by the keyword.

Applicator keywords also define how subschema or referenced schema boolean assertion (Section 7.6) results are modified and/or combined to produce the boolean result of the applicator. Applicators may apply any boolean logic operation to the assertion results of subschemas, but MUST NOT introduce new assertion conditions of their own.

Annotation (Section 7.7) results are preserved along with the instance location and the location of the schema keyword, so that applications can decide how to interpret multiple values.

7.5.1. Referenced and Referencing Schemas

As noted in Section 7.5, an applicator keyword may refer to a schema to be applied, rather than including it as a subschema in the applicator's value. In such situations, the schema being applied is known as the referenced schema, while the schema containing the applicator keyword is the referencing schema.

While root schemas and subschemas are static concepts based on a schema's position within a schema document, referenced and referencing schemas are dynamic. Different pairs of schemas may find themselves in various referenced and referencing arrangements during the evaluation of an instance against a schema.

For some by-reference applicators, such as "$ref" (Section 8.2.3.1), the referenced schema can be determined by static analysis of the schema document's lexical scope. Others, such as "$dynamicRef" (with "$dynamicAnchor"), may make use of dynamic scoping, and therefore only be resolvable in the process of evaluating the schema with an instance.

7.6. Assertions

JSON Schema can be used to assert constraints on a JSON document, which either passes or fails the assertions. This approach can be used to validate conformance with the constraints, or document what is needed to satisfy them.

JSON Schema implementations produce a single boolean result when evaluating an instance against schema assertions.

An instance can only fail an assertion that is present in the schema.

7.6.1. Assertions and Instance Primitive Types

Most 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 from the companion validation vocabulary [json-schema-validation]: 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.

This behavior allows keywords to be used more easily with instances that can be of multiple primitive types. The companion validation vocabulary also includes a "type" keyword which can independently restrict the instance to one or more primitive types. This allows for a concise expression of use cases such as a function that might return either a string of a certain length or a null value:

```
{
    "type": ["string", "null"],
    "maxLength": 255
}
```

If "maxLength" also restricted the instance type to be a string, then this would be substantially more cumbersome to express because the example as written would not actually allow null values. Each keyword is evaluated separately unless explicitly specified otherwise, so if "maxLength" restricted the instance to strings, then including "null" in "type" would not have any useful effect.

7.7. Annotations

JSON Schema can annotate an instance with information, whenever the instance validates against the schema object containing the annotation, and all of its parent schema objects. The information can be a simple value, or can be calculated based on the instance contents.

Annotations are attached to specific locations in an instance. Since many subschemas can be applied to any single location, applications may need to decide how to handle differing annotation values being attached to the same instance location by the same schema keyword in different schema objects.

Unlike assertion results, annotation data can take a wide variety of forms, which are provided to applications to use as they see fit. JSON Schema implementations are not expected to make use of the collected information on behalf of applications.

Unless otherwise specified, the value of an annotation keyword is the keyword's value. However, other behaviors are possible. For example, JSON Hyper-Schema's [json-hyper-schema] "links" keyword is a complex annotation that produces a value based in part on the instance data.

While "short-circuit" evaluation is possible for assertions, collecting annotations requires examining all schemas that apply to

an instance location, even if they cannot change the overall assertion result. The only exception is that subschemas of a schema object that has failed validation MAY be skipped, as annotations are not retained for failing schemas.

7.7.1. Collecting Annotations

Annotations are collected by keywords that explicitly define annotation-collecting behavior. Note that boolean schemas cannot produce annotations as they do not make use of keywords.

A collected annotation MUST include the following information:

The name of the keyword that produces the annotation

The instance location to which it is attached, as a JSON Pointer

The schema location path, indicating how reference keywords such as "$ref" were followed to reach the absolute schema location.

The absolute schema location of the attaching keyword, as a URI. This MAY be omitted if it is the same as the schema location path from above.

The attached value(s)

7.7.1.1. Distinguishing Among Multiple Values

Applications MAY make decisions on which of multiple annotation values to use based on the schema location that contributed the value. This is intended to allow flexible usage. Collecting the schema location facilitates such usage.

For example, consider this schema, which uses annotations and assertions from the Validation specification [json-schema-validation]:

Note that some lines are wrapped for clarity.

```
{
    "title": "Feature list",
    "type": "array",
    "prefixItems": [
        {
            "title": "Feature A",
            "properties": {
                "enabled": {
                    "$ref": "#/$defs/enabledToggle",
                    "default": true
                }
            }
        },
        {
            "title": "Feature B",
            "properties": {
                "enabled": {
                    "description": "If set to null, Feature B
                                    inherits the enabled
                                    value from Feature A",
                    "$ref": "#/$defs/enabledToggle"
                }
            }
        }
    ],
    "$defs": {
        "enabledToggle": {
            "title": "Enabled",
            "description": "Whether the feature is enabled (true),
                            disabled (false), or under
                            automatic control (null)",
            "type": ["boolean", "null"],
            "default": null
        }
    }
}
```

In this example, both Feature A and Feature B make use of the re-usable "enabledToggle" schema. That schema uses the "title", "description", and "default" annotations. Therefore the application has to decide how to handle the additional "default" value for Feature A, and the additional "description" value for Feature B.

The application programmer and the schema author need to agree on the usage. For this example, let's assume that they agree that the most specific "default" value will be used, and any additional, more generic "default" values will be silently ignored. Let's also assume that they agree that all "description" text is to be used, starting with the most generic, and ending with the most specific. This requires the schema author to write descriptions that work when combined in this way.

The application can use the schema location path to determine which values are which. The values in the feature's immediate "enabled" property schema are more specific, while the values under the re-usable schema that is referenced to with "$ref" are more generic. The schema location path will show whether each value was found by crossing a "$ref" or not.

Feature A will therefore use a default value of true, while Feature B will use the generic default value of null. Feature A will only have the generic description from the "enabledToggle" schema, while Feature B will use that description, and also append its locally defined description that explains how to interpret a null value.

Note that there are other reasonable approaches that a different application might take. For example, an application may consider the presence of two different values for "default" to be an error, regardless of their schema locations.

7.7.1.2. Annotations and Assertions

Schema objects that produce a false assertion result MUST NOT produce any annotation results, whether from their own keywords or from keywords in subschemas.

Note that the overall schema results may still include annotations collected from other schema locations. Given this schema:

```
{
    "oneOf": [
        {
            "title": "Integer Value",
            "type": "integer"
        },
        {
            "title": "String Value",
            "type": "string"
        }
    ]
}
```

Against the instance ""This is a string"", the title annotation "Integer Value" is discarded because the type assertion in that schema object fails. The title annotation "String Value" is kept, as the instance passes the string type assertions.

7.7.1.3. Annotations and Applicators

In addition to possibly defining annotation results of their own, applicator keywords aggregate the annotations collected in their subschema(s) or referenced schema(s).

7.8. Reserved Locations

A fourth category of keywords simply reserve a location to hold re-usable components or data of interest to schema authors that is not suitable for re-use. These keywords do not affect validation or annotation results. Their purpose in the core vocabulary is to ensure that locations are available for certain purposes and will not be redefined by extension keywords.

While these keywords do not directly affect results, as explained in section 9.4.2 unrecognized extension keywords that reserve locations for re-usable schemas may have undesirable interactions with references in certain circumstances.

7.9. Loading Instance Data

While none of the vocabularies defined as part of this or the associated documents define a keyword which may target and/or load instance data, it is possible that other vocabularies may wish to do so.

Keywords MAY be defined to use JSON Pointers or Relative JSON Pointers to examine parts of an instance outside the current evaluation location.

Keywords that allow adjusting the location using a Relative JSON Pointer SHOULD default to using the current location if a default is desireable.

8. The JSON Schema Core Vocabulary

Keywords declared in this section, which all begin with "$", make up the JSON Schema Core vocabulary. These keywords are either required in order to process any schema or meta-schema, including those split across multiple documents, or exist to reserve keywords for purposes that require guaranteed interoperability.

The Core vocabulary MUST be considered mandatory at all times, in order to bootstrap the processing of further vocabularies. Meta-schemas that use the "$vocabulary" (Section 8.1) keyword to declare the vocabularies in use MUST explicitly list the Core vocabulary, which MUST have a value of true indicating that it is required.

The behavior of a false value for this vocabulary (and only this vocabulary) is undefined, as is the behavior when "$vocabulary" is present but the Core vocabulary is not included. However, it is RECOMMENDED that implementations detect these cases and raise an error when they occur. It is not meaningful to declare that a meta-schema optionally uses Core.

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

The current URI for the Core vocabulary is: <https://json-schema.org/draft/2020-12/vocab/core>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2020-12/meta/core>.

While the "$" prefix is not formally reserved for the Core vocabulary, it is RECOMMENDED that extension keywords (in vocabularies or otherwise) begin with a character other than "$" to avoid possible future collisions.

8.1. Meta-Schemas and Vocabularies

Two concepts, meta-schemas and vocabularies, are used to inform an implementation how to interpret a schema. Every schema has a meta-schema, which can be declared using the "$schema" keyword.

The meta-schema serves two purposes:

Declaring the vocabularies in use The "$vocabulary" keyword, when it appears in a meta-schema, declares which vocabularies are available to be used in schemas that refer to that meta-schema. Vocabularies define keyword semantics, as well as their general syntax.

Describing valid schema syntax A schema MUST successfully validate against its meta-schema, which constrains the syntax of the available keywords. The syntax described is expected to be compatible with the vocabularies declared; while it is possible to describe an incompatible syntax, such a meta-schema would be unlikely to be useful.

Meta-schemas are separate from vocabularies to allow for vocabularies to be combined in different ways, and for meta-schema authors to impose additional constraints such as forbidding certain keywords, or performing unusually strict syntactical validation, as might be done during a development and testing cycle. Each vocabulary typically identifies a meta-schema consisting only of the vocabulary's keywords.

Meta-schema authoring is an advanced usage of JSON Schema, so the design of meta-schema features emphasizes flexibility over simplicity.

8.1.1. The "$schema" Keyword

The "$schema" keyword is both used as a JSON Schema dialect identifier and as the identifier of a resource which is itself a JSON Schema, which describes the set of valid schemas written for this particular dialect.

The value of this keyword MUST be a URI [RFC3986] (containing a scheme) and this URI MUST be normalized. The current schema MUST be valid against the meta-schema identified by this URI.

If this URI identifies a retrievable resource, that resource SHOULD be of media type "application/schema+json".

The "$schema" keyword SHOULD be used in the document root schema object, and MAY be used in the root schema objects of embedded schema resources. It MUST NOT appear in non-resource root schema objects. If absent from the document root schema, the resulting behavior is implementation-defined.

Values for this property are defined elsewhere in this and other documents, and by other parties.

8.1.2. The "$vocabulary" Keyword

The "$vocabulary" keyword is used in meta-schemas to identify the vocabularies available for use in schemas described by that meta-schema. It is also used to indicate whether each vocabulary is required or optional, in the sense that an implementation MUST understand the required vocabularies in order to successfully process the schema. Together, this information forms a dialect. Any vocabulary that is understood by the implementation MUST be processed in a manner consistent with the semantic definitions contained within the vocabulary.

The value of this keyword MUST be an object. The property names in the object MUST be URIs (containing a scheme) and this URI MUST be normalized. Each URI that appears as a property name identifies a specific set of keywords and their semantics.

The URI MAY be a URL, but the nature of the retrievable resource is currently undefined, and reserved for future use. Vocabulary authors MAY use the URL of the vocabulary specification, in a human-readable media type such as text/html or text/plain, as the vocabulary URI. [[CREF1: Vocabulary documents may be added in forthcoming drafts. For now, identifying the keyword set is deemed sufficient as that, along with meta-schema validation, is how the current "vocabularies" work today. Any future vocabulary document format will be specified as a JSON document, so using text/html or other non-JSON formats in the meantime will not produce any future ambiguity.]]

The values of the object properties MUST be booleans. If the value is true, then implementations that do not recognize the vocabulary MUST refuse to process any schemas that declare this meta-schema with "$schema". If the value is false, implementations that do not recognize the vocabulary SHOULD proceed with processing such schemas. The value has no impact if the implementation understands the vocabulary.

Per 6.5, unrecognized keywords SHOULD be treated as annotations. This remains the case for keywords defined by unrecognized vocabularies. It is not currently possible to distinguish between

unrecognized keywords that are defined in vocabularies from those that are not part of any vocabulary.

The "$vocabulary" keyword SHOULD be used in the root schema of any schema document intended for use as a meta-schema. It MUST NOT appear in subschemas.

The "$vocabulary" keyword MUST be ignored in schema documents that are not being processed as a meta-schema. This allows validating a meta-schema M against its own meta-schema M' without requiring the validator to understand the vocabularies declared by M.

8.1.2.1. Default vocabularies

If "$vocabulary" is absent, an implementation MAY determine behavior based on the meta-schema if it is recognized from the URI value of the referring schema's "$schema" keyword. This is how behavior (such as Hyper-Schema usage) has been recognized prior to the existence of vocabularies.

If the meta-schema, as referenced by the schema, is not recognized, or is missing, then the behavior is implementation-defined. If the implementation proceeds with processing the schema, it MUST assume the use of the core vocabulary. If the implementation is built for a specific purpose, then it SHOULD assume the use of all of the most relevant vocabularies for that purpose.

For example, an implementation that is a validator SHOULD assume the use of all vocabularies in this specification and the companion Validation specification.

8.1.2.2. Non-inheritability of vocabularies

Note that the processing restrictions on "$vocabulary" mean that meta-schemas that reference other meta-schemas using "$ref" or similar keywords do not automatically inherit the vocabulary declarations of those other meta-schemas. All such declarations must be repeated in the root of each schema document intended for use as a meta-schema. This is demonstrated in the example meta-schema (Appendix D.2). [[CREF2: This requirement allows implementations to find all vocabulary requirement information in a single place for each meta-schema. As schema extensibility means that there are endless potential ways to combine more fine-grained meta-schemas by reference, requiring implementations to anticipate all possibilities and search for vocabularies in referenced meta-schemas would be overly burdensome.]]

8.1.3. Updates to Meta-Schema and Vocabulary URIs

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.

8.2. Base URI, Anchors, and Dereferencing

To differentiate between schemas in a vast ecosystem, schemas are identified by URI [RFC3986], and can embed references to other schemas by specifying their URI.

Several keywords can accept a relative URI-reference [RFC3986], or a value used to construct a relative URI-reference. For these keywords, it is necessary to establish a base URI in order to resolve the reference.

8.2.1. The "$id" Keyword

The "$id" keyword identifies a schema resource with its canonical [RFC6596] URI.

Note that this URI is an identifier and not necessarily a network locator. In the case of a network-addressable URL, a schema need not be downloadable from its canonical URI.

If present, the value for this keyword MUST be a string, and MUST represent a valid URI-reference [RFC3986]. This URI-reference SHOULD be normalized, and MUST resolve to an absolute-URI [RFC3986] (without a fragment). Therefore, "$id" MUST NOT contain a non-empty fragment, and SHOULD NOT contain an empty fragment.

Since an empty fragment in the context of the application/schema+json media type refers to the same resource as the base URI without a fragment, an implementation MAY normalize a URI ending with an empty fragment by removing the fragment. However, schema authors SHOULD NOT rely on this behavior across implementations. [[CREF3: This is primarily allowed because older meta-schemas have an empty fragment in their $id (or previously, id). A future draft may outright forbid even empty fragments in "$id".]]

This URI also serves as the base URI for relative URI-references in keywords within the schema resource, in accordance with RFC 3986 section 5.1.1 [RFC3986] regarding base URIs embedded in content.

The presence of "$id" in a subschema indicates that the subschema constitutes a distinct schema resource within a single schema document. Furthermore, in accordance with RFC 3986 section 5.1.2 [RFC3986] regarding encapsulating entities, if an "$id" in a subschema is a relative URI-reference, the base URI for resolving that reference is the URI of the parent schema resource.

If no parent schema object explicitly identifies itself as a resource with "$id", the base URI is that of the entire document, as established by the steps given in the previous section. (Section 9.1.1)

8.2.1.1. Identifying the root schema

The root schema of a JSON Schema document SHOULD contain an "$id" keyword with an absolute-URI [RFC3986] (containing a scheme, but no fragment).

8.2.2. Defining location-independent identifiers

Using JSON Pointer fragments requires knowledge of the structure of the schema. When writing schema documents with the intention to provide re-usable schemas, it may be preferable to use a plain name fragment that is not tied to any particular structural location. This allows a subschema to be relocated without requiring JSON Pointer references to be updated.

The "$anchor" and "$dynamicAnchor" keywords are used to specify such fragments. They are identifier keywords that can only be used to create plain name fragments, rather than absolute URIs as seen with "$id".

The base URI to which the resulting fragment is appended is the canonical URI of the schema resource containing the "$anchor" or "$dynamicAnchor" in question. As discussed in the previous section, this is either the nearest "$id" in the same or parent schema object, or the base URI for the document as determined according to RFC 3986.

Separately from the usual usage of URIs, "$dynamicAnchor" indicates that the fragment is an extension point when used with the "$dynamicRef" keyword. This low-level, advanced feature makes it easier to extend recursive schemas such as the meta-schemas, without imposing any particular semantics on that extension. See the section on "$dynamicRef" (Section 8.2.3.2) for details.

In most cases, the normal fragment behavior both suffices and is more intuitive. Therefore it is RECOMMENDED that "$anchor" be used to

create plain name fragments unless there is a clear need for "$dynamicAnchor".

If present, the value of this keyword MUST be a string and MUST start with a letter ([A-Za-z]) or underscore ("_"), followed by any number of letters, digits ([0-9]), hyphens ("-"), underscores ("_"), and periods ("."). This matches the US-ASCII part of XML's NCName production [xml-names]. [[CREF4: Note that the anchor string does not include the "#" character, as it is not a URI-reference. An "$anchor": "foo" becomes the fragment "#foo" when used in a URI. See below for full examples.]]

The effect of specifying the same fragment name multiple times within the same resource, using any combination of "$anchor" and/or "$dynamicAnchor", is undefined. Implementations MAY raise an error if such usage is detected.

8.2.3. Schema References

Several keywords can be used to reference a schema which is to be applied to the current instance location. "$ref" and "$dynamicRef" are applicator keywords, applying the referenced schema to the instance.

As the values of "$ref" and "$dynamicRef" are URI References, this allows the possibility to externalise or divide a schema across multiple files, and provides the ability to validate recursive structures through self-reference.

The resolved URI produced by these keywords is not necessarily a network locator, only an identifier. A schema need not be downloadable from the address if it is a network-addressable URL, and implementations SHOULD NOT assume they should perform a network operation when they encounter a network-addressable URI.

8.2.3.1. Direct References with "$ref"

The "$ref" keyword is an applicator that is used to reference a statically identified schema. Its results are the results of the referenced schema. [[CREF5: Note that this definition of how the results are determined means that other keywords can appear alongside of "$ref" in the same schema object.]]

The value of the "$ref" keyword MUST be a string which is a URI-Reference. Resolved against the current URI base, it produces the URI of the schema to apply. This resolution is safe to perform on schema load, as the process of evaluating an instance cannot change how the reference resolves.

8.2.3.2. Dynamic References with "$dynamicRef"

The "$dynamicRef" keyword is an applicator that allows for deferring the full resolution until runtime, at which point it is resolved each time it is encountered while evaluating an instance.

Together with "$dynamicAnchor", "$dynamicRef" implements a cooperative extension mechanism that is primarily useful with recursive schemas (schemas that reference themselves). Both the extension point and the runtime-determined extension target are defined with "$dynamicAnchor", and only exhibit runtime dynamic behavior when referenced with "$dynamicRef".

The value of the "$dynamicRef" property MUST be a string which is a URI-Reference. Resolved against the current URI base, it produces the URI used as the starting point for runtime resolution. This initial resolution is safe to perform on schema load.

If the initially resolved starting point URI includes a fragment that was created by the "$dynamicAnchor" keyword, the initial URI MUST be replaced by the URI (including the fragment) for the outermost schema resource in the dynamic scope (Section 7.1) that defines an identically named fragment with "$dynamicAnchor".

Otherwise, its behavior is identical to "$ref", and no runtime resolution is needed.

For a full example using these keyword, see appendix C. [[CREF6: The difference between the hyper-schema meta-schema in pre-2019 drafts and an this draft dramatically demonstrates the utility of these keywords.]]

8.2.4. Schema Re-Use With "$defs"

The "$defs" keyword reserves a 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 "$defs":

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

8.3. Comments With "$comment"

This keyword reserves a location for comments from schema authors to readers or maintainers of the schema.

The value of this keyword MUST be a string. Implementations MUST NOT present this string to end users. Tools for editing schemas SHOULD support displaying and editing this keyword. The value of this keyword MAY be used in debug or error output which is intended for developers making use of schemas.

Schema vocabularies SHOULD allow "$comment" within any object containing vocabulary keywords. Implementations MAY assume "$comment" is allowed unless the vocabulary specifically forbids it. Vocabularies MUST NOT specify any effect of "$comment" beyond what is described in this specification.

Tools that translate other media types or programming languages to and from application/schema+json MAY choose to convert that media type or programming language's native comments to or from "$comment" values. The behavior of such translation when both native comments and "$comment" properties are present is implementation-dependent.

Implementations MAY strip "$comment" values at any point during processing. In particular, this allows for shortening schemas when the size of deployed schemas is a concern.

Implementations MUST NOT take any other action based on the presence, absence, or contents of "$comment" properties. In particular, the value of "$comment" MUST NOT be collected as an annotation result.

9. Loading and Processing Schemas

9.1. Loading a Schema

9.1.1. Initial Base URI

RFC3986 Section 5.1 [RFC3986] defines how to determine the default base URI of a document.

Informatively, the initial base URI of a schema is the URI at which it was found, whether that was a network location, a local filesystem, or any other situation identifiable by a URI of any known scheme.

If a schema document defines no explicit base URI with "$id" (embedded in content), the base URI is that determined per RFC 3986 section 5 [RFC3986].

If no source is known, or no URI scheme is known for the source, a suitable implementation-specific default URI MAY be used as described in RFC 3986 Section 5.1.4 [RFC3986]. It is RECOMMENDED that implementations document any default base URI that they assume.

If a schema object is embedded in a document of another media type, then the initial base URI is determined according to the rules of that media type.

Unless the "$id" keyword described in the next section is present in the root schema, this base URI SHOULD be considered the canonical URI of the schema document's root schema resource.

9.1.2. Loading a referenced schema

The use of URIs to identify remote schemas does not necessarily mean anything is downloaded, but instead JSON Schema implementations SHOULD understand ahead of time which schemas they will be using, and the URIs that identify them.

When schemas are downloaded, for example by a generic user-agent that does not know until runtime which schemas to download, see Usage for Hypermedia (Section 9.5.1).

Implementations SHOULD be able to associate arbitrary URIs with an arbitrary schema and/or automatically associate a schema's "$id"-given URI, depending on the trust that the validator has in the schema. Such URIs and schemas can be supplied to an implementation prior to processing instances, or may be noted within a schema document as it is processed, producing associations as shown in appendix A.

A schema MAY (and likely will) have multiple URIs, but there is no way for a URI to identify more than one schema. When multiple schemas try to identify as the same URI, validators SHOULD raise an error condition.

9.1.3. Detecting a Meta-Schema

Implementations MUST recognize a schema as a meta-schema if it is being examined because it was identified as such by another schema's "$schema" keyword. This means that a single schema document might sometimes be considered a regular schema, and other times be considered a meta-schema.

In the case of examining a schema which is its own meta-schema, when an implementation begins processing it as a regular schema, it is processed under those rules. However, when loaded a second time as a result of checking its own "$schema" value, it is treated as a meta-schema. So the same document is processed both ways in the course of one session.

Implementations MAY allow a schema to be explicitly passed as a meta-schema, for implementation-specific purposes, such as pre-loading a commonly used meta-schema and checking its vocabulary support requirements up front. Meta-schema authors MUST NOT expect such features to be interoperable across implementations.

9.2. Dereferencing

Schemas can be identified by any URI that has been given to them, including a JSON Pointer or their URI given directly by "$id". In all cases, dereferencing a "$ref" reference involves first resolving its value as a URI reference against the current base URI per RFC 3986 [RFC3986].

If the resulting URI identifies a schema within the current document, or within another schema document that has been made available to the implementation, then that schema SHOULD be used automatically.

For example, consider this schema:

```
{
    "$id": "https://example.net/root.json",
    "items": {
        "type": "array",
        "items": { "$ref": "#item" }
    },
    "$defs": {
        "single": {
            "$anchor": "item",
            "type": "object",
            "additionalProperties": { "$ref": "other.json" }
        }
    }
}
```

When an implementation encounters the <#/$defs/single> schema, it resolves the "$anchor" value as a fragment name against the current base URI to form <https://example.net/root.json#item>.

When an implementation then looks inside the <#/items> schema, it encounters the <#item> reference, and resolves this to <https://example.net/root.json#item>, which it has seen defined in this same document and can therefore use automatically.

When an implementation encounters the reference to "other.json", it resolves this to <https://example.net/other.json>, which is not defined in this document. If a schema with that identifier has otherwise been supplied to the implementation, it can also be used automatically. [[CREF7: What should implementations do when the referenced schema is not known? Are there circumstances in which automatic network dereferencing is allowed? A same origin policy? A user-configurable option? In the case of an evolving API described by Hyper-Schema, it is expected that new schemas will be added to the system dynamically, so placing an absolute requirement of pre-loading schema documents is not feasible.]]

9.2.1. JSON Pointer fragments and embedded schema resources

Since JSON Pointer URI fragments are constructed based on the structure of the schema document, an embedded schema resource and its subschemas can be identified by JSON Pointer fragments relative to either its own canonical URI, or relative to the containing resource's URI.

Conceptually, a set of linked schema resources should behave identically whether each resource is a separate document connected

with schema references (Section 8.2.3), or is structured as a single document with one or more schema resources embedded as subschemas.

Since URIs involving JSON Pointer fragments relative to the parent schema resource's URI cease to be valid when the embedded schema is moved to a separate document and referenced, applications and schemas SHOULD NOT use such URIs to identify embedded schema resources or locations within them.

Consider the following schema document that contains another schema resource embedded within it:

```
{
    "$id": "https://example.com/foo",
    "items": {
        "$id": "https://example.com/bar",
        "additionalProperties": { }
    }
}
```

The URI "https://example.com/foo#/items/additionalProperties" points to the schema of the "additionalProperties" keyword in the embedded resource. The canonical URI of that schema, however, is "https://example.com/bar#/additionalProperties".

Now consider the following two schema resources linked by reference using a URI value for "$ref":

```
{
    "$id": "https://example.com/foo",
    "items": {
        "$ref": "bar"
    }
}

{
    "$id": "https://example.com/bar",
    "additionalProperties": { }
}
```

Here we see that the canonical URI for that "additionalProperties" subschema is still valid, while the non-canonical URI with the fragment beginning with "#/items/$ref" now resolves to nothing.

Note also that "https://example.com/foo#/items" is valid in both arrangements, but resolves to a different value. This URI ends up functioning similarly to a retrieval URI for a resource. While valid, examining the resolved value and either using the "$id" (if the value is a subschema), or resolving the reference and using the "$id" of the reference target, is preferable.

An implementation MAY choose not to support addressing schemas by non-canonical URIs. As such, it is RECOMMENDED that schema authors only use canonical URIs, as using non-canonical URIs may reduce schema interoperability. [[CREF8: This is to avoid requiring implementations to keep track of a whole stack of possible base URIs and JSON Pointer fragments for each, given that all but one will be fragile if the schema resources are reorganized. Some have argued that this is easy so there is no point in forbidding it, while others have argued that it complicates schema identification and should be forbidden. Feedback on this topic is encouraged.]]

Further examples of such non-canonical URIs, as well as the appropriate canonical URIs to use instead, are provided in appendix A.

9.3. Compound Documents

A Compound Schema Document is defined as a JSON document (sometimes called a "bundled" schema) which has multiple embedded JSON Schema Resources bundled into the same document to ease transportation.

Each embedded Schema Resource MUST be treated as an individual Schema Resource, following standard schema loading and processing requirements, including determining vocabulary support.

9.3.1. Bundling

The bundling process for creating a Compound Schema Document is defined as taking references (such as "$ref") to an external Schema Resource and embedding the referenced Schema Resources within the referring document. Bundling SHOULD be done in such a way that all URIs (used for referencing) in the base document and any referenced/embedded documents do not require altering.

Each embedded JSON Schema Resource MUST identify itself with a URI using the "$id" keyword, and SHOULD make use of the "$schema" keyword to identify the dialect it is using, in the root of the schema resource. It is RECOMMENDED that the URI identifier value of "$id" be an Absolute URI.

When the Schema Resource referenced by a by-reference applicator is bundled, it is RECOMMENDED that the Schema Resource be located as a value of a "$defs" object at the containing schema's root. The key of the "$defs" for the now embedded Schema Resource MAY be the "$id" of the bundled schema or some other form of application defined unique identifer (such as a UUID). This key is not intended to be referenced in JSON Schema, but may be used by an application to aid the bundling process.

A Schema Resource MAY be embedded in a location other than "$defs" where the location is defined as a schema value.

A Bundled Schema Resource MUST NOT be bundled by replacing the schema object from which it was referenced, or by wrapping the Schema Resource in other applicator keywords.

In order to produce identical output, references in the containing schema document to the previously external Schema Resources MUST NOT be changed, and now resolve to a schema using the "$id" of an embedded Schema Resource. Such identical output includes validation evaluation and URIs or paths used in resulting annotations or errors.

While the bundling process will often be the main method for creating a Compound Schema Document, it is also possible and expected that some will be created by hand, potentially without individual Schema Resources existing on their own previously.

9.3.2. Differing and Default Dialects

When multiple schema resources are present in a single document, schema resources which do not define with which dialect they should be processed MUST be processed with the same dialect as the enclosing resource.

Since any schema that can be referenced can also be embedded, embedded schema resources MAY specify different processing dialects using the "$schema" values from their enclosing resource.

9.3.3. Validating

Given that a Compound Schema Document may have embedded resources which identify as using different dialects, these documents SHOULD NOT be validated by applying a meta-schema to the Compound Schema Document as an instance. It is RECOMMENDED that an alternate validation process be provided in order to validate Schema Documents. Each Schema Resource SHOULD be separately validated against its associated meta-schema. [[CREF9: If you know a schema is what's being validated, you can identify if the schemas is a Compound Schema

Document or not, by way of use of "$id", which identifies an embedded resource when used not at the document's root.]]

A Compound Schema Document in which all embedded resources identify as using the same dialect, or in which "$schema" is omitted and therefore defaults to that of the enclosing resource, MAY be validated by applying the appropriate meta-schema.

9.4. Caveats

9.4.1. Guarding Against Infinite Recursion

A schema MUST NOT be run into an infinite loop against an instance. For example, if two schemas "#alice" and "#bob" both have an "allOf" property that refers to the other, a naive validator might get stuck in an infinite recursive loop trying to validate the instance. Schemas SHOULD NOT make use of infinite recursive nesting like this; the behavior is undefined.

9.4.2. References to Possible Non-Schemas

Subschema objects (or booleans) are recognized by their use with known applicator keywords or with location-reserving keywords such as "$defs" (Section 8.2.4) that take one or more subschemas as a value. These keywords may be "$defs" and the standard applicators from this document, or extension keywords from a known vocabulary, or implementation-specific custom keywords.

Multi-level structures of unknown keywords are capable of introducing nested subschemas, which would be subject to the processing rules for "$id". Therefore, having a reference target in such an unrecognized structure cannot be reliably implemented, and the resulting behavior is undefined. Similarly, a reference target under a known keyword, for which the value is known not to be a schema, results in undefined behavior in order to avoid burdening implementations with the need to detect such targets. [[CREF10: These scenarios are analogous to fetching a schema over HTTP but receiving a response with a Content-Type other than application/schema+json. An implementation can certainly try to interpret it as a schema, but the origin server offered no guarantee that it actually is any such thing. Therefore, interpreting it as such has security implications and may produce unpredictable results.]]

Note that single-level custom keywords with identical syntax and semantics to "$defs" do not allow for any intervening "$id" keywords, and therefore will behave correctly under implementations that attempt to use any reference target as a schema. However, this

behavior is implementation-specific and MUST NOT be relied upon for interoperability.

9.5. Associating Instances and Schemas

9.5.1. Usage for Hypermedia

JSON has been adopted widely by HTTP servers for automated APIs and robots. This section describes how to enhance processing of JSON documents in a more RESTful manner when used with protocols that support media types and Web linking [RFC8288].

9.5.1.1. Linking to a Schema

It is RECOMMENDED that instances described by a schema provide a link to a downloadable JSON Schema using the link relation "describedby", as defined by Linked Data Protocol 1.0, section 8.1 [W3C.REC-ldp-20150226].

In HTTP, such links can be attached to any response using the Link header [RFC8288]. An example of such a header would be:

```
Link: <https://example.com/my-hyper-schema>; rel="describedby"
```

9.5.1.2. Usage Over HTTP

When used for hypermedia systems over a network, HTTP [RFC7231] is frequently the protocol of choice for distributing schemas. Misbehaving clients can pose problems for server maintainers if they pull a schema over the network more frequently than necessary, when it's instead possible to cache a schema for a long period of time.

HTTP servers SHOULD set long-lived caching headers on JSON Schemas. HTTP clients SHOULD observe caching headers and not re-request documents within their freshness period. Distributed systems SHOULD make use of a shared cache and/or caching proxy.

Clients SHOULD set or prepend a User-Agent header specific to the JSON Schema implementation or software product. Since symbols are listed in decreasing order of significance, the JSON Schema library name/version should precede the more generic HTTP library name (if any). For example:

```
User-Agent: product-name/5.4.1 so-cool-json-schema/1.0.2 curl/7.43.0
```

Clients SHOULD be able to make requests with a "From" header so that server operators can contact the owner of a potentially misbehaving script.

10. A Vocabulary for Applying Subschemas

This section defines a vocabulary of applicator keywords that are RECOMMENDED for use as the basis of other vocabularies.

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 Applicator vocabulary, is: <https://json-schema.org/draft/2020-12/vocab/applicator>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2020-12/meta/applicator>.

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.

10.1. Keyword Independence

Schema keywords typically operate independently, without affecting each other's outcomes.

For schema author convenience, there are some exceptions among the keywords in this vocabulary:

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

"items", whose behavior is defined in terms of "prefixItems"

"contains", whose behavior is defined in terms of "minContains"

10.2. Keywords for Applying Subschemas in Place

These keywords apply subschemas to the same location in the instance as the parent schema is being applied. They allow combining or modifying the subschema results in various ways.

Subschemas of these keywords evaluate the instance completely independently such that the results of one such subschema MUST NOT impact the results of sibling subschemas. Therefore subschemas may be applied in any order.

10.2.1. Keywords for Applying Subschemas With Logic

These keywords correspond to logical operators for combining or modifying the boolean assertion results of the subschemas. They have no direct impact on annotation collection, although they enable the same annotation keyword to be applied to an instance location with different values. Annotation keywords define their own rules for combining such values.

10.2.1.1. 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.

10.2.1.2. 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. Note that when annotations are being collected, all subschemas MUST be examined so that annotations are collected from each subschema that validates successfully.

10.2.1.3. 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.

10.2.1.4. 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.

10.2.2. Keywords for Applying Subschemas Conditionally

Three of these keywords work together to implement conditional application of a subschema based on the outcome of another subschema. The fourth is a shortcut for a specific conditional case.

"if", "then", and "else" 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 "if", "then", or "else" 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.

10.2.2.1. 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 7.7) 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".

10.2.2.2. then

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

When "if" is present, and the instance successfully validates against its subschema, then validation 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.

10.2.2.3. 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 validation 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.

10.2.2.4. dependentSchemas

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

This keyword's value MUST be an object. Each value in the object MUST be a valid JSON Schema.

If the object key is a property in the instance, the entire instance must validate against the subschema. Its use is dependent on the presence of the property.

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

10.3. Keywords for Applying Subschemas to Child Instances

Each of these keywords defines a rule for applying its subschema(s) to child instances, specifically object properties and array items, and combining their results.

10.3.1. Keywords for Applying Subschemas to Arrays

10.3.1.1. prefixItems

The value of "prefixItems" MUST be a non-empty array of valid JSON Schemas.

Validation succeeds if each element of the instance validates against the schema at the same position, if any. This keyword does not constrain the length of the array. If the array is longer than this

keyword's value, this keyword validates only the prefix of matching length.

This keyword produces an annotation value which is the largest index to which this keyword applied a subschema. The value MAY be a boolean true if a subschema was applied to every index of the instance, such as is produced by the "items" keyword. This annotation affects the behavior of "items" and "unevaluatedItems".

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

10.3.1.2. items

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

This keyword applies its subschema to all instance elements at indexes greater than the length of the "prefixItems" array in the same schema object, as reported by the annotation result of that "prefixItems" keyword. If no such annotation result exists, "items" applies its subschema to all instance array elements. [[CREF11: Note that the behavior of "items" without "prefixItems" is identical to that of the schema form of "items" in prior drafts. When "prefixItems" is present, the behavior of "items" is identical to the former "additionalItems" keyword.]]

If the "items" subschema is applied to any positions within the instance array, it produces an annotation result of boolean true, indicating that all remaining array elements have been evaluated against this keyword's subschema.

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

Implementations MAY choose to implement or optimize this keyword in another way that produces the same effect, such as by directly checking for the presence and size of a "prefixItems" array. Implementations that do not support annotation collection MUST do so.

10.3.1.3. 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. The subschema MUST be applied to every array element even after the first match has been found, in order to collect annotations for use by other keywords. This is to ensure that all possible annotations are collected.

Logically, the validation result of applying the value subschema to each item in the array MUST be ORed with "false", resulting in an overall validation result.

This keyword produces an annotation value which is an array of the indexes to which this keyword validates successfully when applying its subschema, in ascending order. The value MAY be a boolean "true" if the subschema validates successfully when applied to every index of the instance. The annotation MUST be present if the instance array to which this keyword's schema applies is empty.

10.3.2. Keywords for Applying Subschemas to Objects

10.3.2.1. properties

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

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.

The annotation result of this keyword is the set of instance property names matched by this keyword.

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

10.3.2.2. 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.

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.

The annotation result of this keyword is the set of instance property names matched by this keyword.

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

10.3.2.3. additionalProperties

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

The behavior of this keyword depends on the presence and annotation results of "properties" and "patternProperties" within the same schema object. Validation with "additionalProperties" applies only to the child values of instance names that do not appear in the annotation results of either "properties" or "patternProperties".

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

The annotation result of this keyword is the set of instance property names validated by this keyword's subschema.

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

Implementations MAY choose to implement or optimize this keyword in another way that produces the same effect, such as by directly checking the names in "properties" and the patterns in "patternProperties" against the instance property set. Implementations that do not support annotation collection MUST do so.

10.3.2.4. 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.

11. A Vocabulary for Unevaluated Locations

The purpose of these keywords is to enable schema authors to apply subschemas to array items or object properties that have not been successfully evaluated against any dynamic-scope subschema of any adjacent keywords.

These instance items or properties may have been unsuccessfully evaluated against one or more adjacent keyword subschemas, such as when an assertion in a branch of an "anyOf" fails. Such failed evaluations are not considered to contribute to whether or not the

item or property has been evaluated. Only successful evaluations are considered.

If an item in an array or an object property is "successfully evaluated", it is logically considered to be valid in terms of the representation of the object or array that's expected. For example if a subschema represents a car, which requires between 2-4 wheels, and the value of "wheels" is 6, the instance object is not "evaluated" to be a car, and the "wheels" property is considered "unevaluated (successfully as a known thing)", and does not retain any annotations.

Recall that adjacent keywords are keywords within the same schema object, and that the dynamic-scope subschemas include reference targets as well as lexical subschemas.

The behavior of these keywords depend on the annotation results of adjacent keywords that apply to the instance location being validated.

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 Unevaluated Applicator vocabulary, is: <https://json-schema.org/draft/2020-12/vocab/unevaluated>.

The current URI for the corresponding meta-schema is: <https://json-schema.org/draft/2020-12/meta/unevaluated>.

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.

11.1. Keyword Independence

Schema keywords typically operate independently, without affecting each other's outcomes. However, the keywords in this vocabulary are notable exceptions:

- "unevaluatedItems", whose behavior is defined in terms of annotations from "prefixItems", "items", "contains", and itself

"unevaluatedProperties", whose behavior is defined in terms of annotations from "properties", "patternProperties", "additionalProperties" and itself

11.2. unevaluatedItems

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

The behavior of this keyword depends on the annotation results of adjacent keywords that apply to the instance location being validated. Specifically, the annotations from "prefixItems", "items", and "contains", which can come from those keywords when they are adjacent to the "unevaluatedItems" keyword. Those three annotations, as well as "unevaluatedItems", can also result from any and all adjacent in-place applicator (Section 10.2) keywords. This includes but is not limited to the in-place applicators defined in this document.

If no relevant annotations are present, the "unevaluatedItems" subschema MUST be applied to all locations in the array. If a boolean true value is present from any of the relevant annotations, "unevaluatedItems" MUST be ignored. Otherwise, the subschema MUST be applied to any index greater than the largest annotation value for "prefixItems", which does not appear in any annotation value for "contains".

This means that "prefixItems", "items", "contains", and all in-place applicators MUST be evaluated before this keyword can be evaluated. Authors of extension keywords MUST NOT define an in-place applicator that would need to be evaluated after this keyword.

If the "unevaluatedItems" subschema is applied to any positions within the instance array, it produces an annotation result of boolean true, analogous to the behavior of "items".

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

11.3. unevaluatedProperties

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

The behavior of this keyword depends on the annotation results of adjacent keywords that apply to the instance location being validated. Specifically, the annotations from "properties", "patternProperties", and "additionalProperties", which can come from those keywords when they are adjacent to the "unevaluatedProperties" keyword. Those three annotations, as well as

"unevaluatedProperties", can also result from any and all adjacent in-place applicator (Section 10.2) keywords. This includes but is not limited to the in-place applicators defined in this document.

Validation with "unevaluatedProperties" applies only to the child values of instance names that do not appear in the "properties", "patternProperties", "additionalProperties", or "unevaluatedProperties" annotation results that apply to the instance location being validated.

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

This means that "properties", "patternProperties", "additionalProperties", and all in-place applicators MUST be evaluated before this keyword can be evaluated. Authors of extension keywords MUST NOT define an in-place applicator that would need to be evaluated after this keyword.

The annotation result of this keyword is the set of instance property names validated by this keyword's subschema.

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

12. Output Formatting

JSON Schema is defined to be platform-independent. As such, to increase compatibility across platforms, implementations SHOULD conform to a standard validation output format. This section describes the minimum requirements that consumers will need to properly interpret validation results.

12.1. Format

JSON Schema output is defined using the JSON Schema data instance model as described in section 4.2.1. Implementations MAY deviate from this as supported by their specific languages and platforms, however it is RECOMMENDED that the output be convertible to the JSON format defined herein via serialization or other means.

12.2. Output Formats

This specification defines four output formats. See the "Output Structure" section for the requirements of each format.

Flag - A boolean which simply indicates the overall validation result with no further details.

Basic - Provides validation information in a flat list structure.

Detailed - Provides validation information in a condensed hierarchical structure based on the structure of the schema.

Verbose - Provides validation information in an uncondensed hierarchical structure that matches the exact structure of the schema.

An implementation SHOULD provide at least one of the "flag", "basic", or "detailed" format and MAY provide the "verbose" format. If it provides one or more of the "detailed" or "verbose" formats, it MUST also provide the "flag" format. Implementations SHOULD specify in their documentation which formats they support.

12.3. Minimum Information

Beyond the simplistic "flag" output, additional information is useful to aid in debugging a schema or instance. Each sub-result SHOULD contain the information contained within this section at a minimum.

A single object that contains all of these components is considered an output unit.

Implementations MAY elect to provide additional information.

12.3.1. Keyword Relative Location

The relative location of the validating keyword that follows the validation path. The value MUST be expressed as a JSON Pointer, and it MUST include any by-reference applicators such as "$ref" or "$dynamicRef".

```
#/properties/width/$ref/minimum
```

Note that this pointer may not be resolvable by the normal JSON Pointer process due to the inclusion of these by-reference applicator keywords.

The JSON key for this information is "keywordLocation".

12.3.2. Keyword Absolute Location

The absolute, dereferenced location of the validating keyword. The value MUST be expressed as a full URI using the canonical URI of the relevant schema object, and it MUST NOT include by-reference

applicators such as "$ref" or "$dynamicRef" as non-terminal path components. It MAY end in such keywords if the error or annotation is for that keyword, such as an unresolvable reference. [[CREF12: Note that "absolute" here is in the sense of "absolute filesystem path" (meaning the complete location) rather than the "absolute-URI" terminology from RFC 3986 (meaning with scheme but without fragment). Keyword absolute locations will have a fragment in order to identify the keyword.]]

https://example.com/schemas/common#/$defs/count/minimum

This information MAY be omitted only if either the dynamic scope did not pass over a reference or if the schema does not declare an absolute URI as its "$id".

The JSON key for this information is "absoluteKeywordLocation".

12.3.3. Instance Location

The location of the JSON value within the instance being validated. The value MUST be expressed as a JSON Pointer.

The JSON key for this information is "instanceLocation".

12.3.4. Error or Annotation

The error or annotation that is produced by the validation.

For errors, the specific wording for the message is not defined by this specification. Implementations will need to provide this.

For annotations, each keyword that produces an annotation specifies its format. By default, it is the keyword's value.

The JSON key for failed validations is "error"; for successful validations it is "annotation".

12.3.5. Nested Results

For the two hierarchical structures, this property will hold nested errors and annotations.

The JSON key for nested results in failed validations is "errors"; for successful validations it is "annotations". Note the plural forms, as a keyword with nested results can also have a local error or annotation.

12.4. Output Structure

The output MUST be an object containing a boolean property named "valid". When additional information about the result is required, the output MUST also contain "errors" or "annotations" as described below.

"valid" - a boolean value indicating the overall validation success or failure

"errors" - the collection of errors or annotations produced by a failed validation

"annotations" - the collection of errors or annotations produced by a successful validation

For these examples, the following schema and instance will be used.

```
{
  "$id": "https://example.com/polygon",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$defs": {
    "point": {
      "type": "object",
      "properties": {
        "x": { "type": "number" },
        "y": { "type": "number" }
      },
      "additionalProperties": false,
      "required": [ "x", "y" ]
    }
  },
  "type": "array",
  "items": { "$ref": "#/$defs/point" },
  "minItems": 3
}
```

```
[
  {
    "x": 2.5,
    "y": 1.3
  },
  {
    "x": 1,
    "z": 6.7
  }
]
```

This instance will fail validation and produce errors, but it's trivial to deduce examples for passing schemas that produce annotations.

Specifically, the errors it will produce are:

The second object is missing a "y" property.

The second object has a disallowed "z" property.

There are only two objects, but three are required.

Note that the error message wording as depicted in these examples is not a requirement of this specification. Implementations SHOULD craft error messages tailored for their audience or provide a templating mechanism that allows their users to craft their own messages.

12.4.1. Flag

In the simplest case, merely the boolean result for the "valid" valid property needs to be fulfilled.

```
{
  "valid": false
}
```

Because no errors or annotations are returned with this format, it is RECOMMENDED that implementations use short-circuiting logic to return failure or success as soon as the outcome can be determined. For example, if an "anyOf" keyword contains five sub-schemas, and the second one passes, there is no need to check the other three. The logic can simply return with success.

12.4.2. Basic

The "Basic" structure is a flat list of output units.

```
{
  "valid": false,
  "errors": [
    {
      "keywordLocation": "",
      "instanceLocation": "",
      "error": "A subschema had errors."
    },
    {
      "keywordLocation": "/items/$ref",
      "absoluteKeywordLocation":
        "https://example.com/polygon#/$defs/point",
      "instanceLocation": "/1",
      "error": "A subschema had errors."
    },
    {
      "keywordLocation": "/items/$ref/required",
      "absoluteKeywordLocation":
        "https://example.com/polygon#/$defs/point/required",
      "instanceLocation": "/1",
      "error": "Required property 'y' not found."
    },
    {
      "keywordLocation": "/items/$ref/additionalProperties",
      "absoluteKeywordLocation":
        "https://example.com/polygon#/$defs/point/additionalProperties",
      "instanceLocation": "/1/z",
      "error": "Additional property 'z' found but was invalid."
    },
    {
      "keywordLocation": "/minItems",
      "instanceLocation": "",
      "error": "Expected at least 3 items but found 2"
    }
  ]
}
```

12.4.3. Detailed

The "Detailed" structure is based on the schema and can be more readable for both humans and machines. Having the structure organized this way makes associations between the errors more apparent. For example, the fact that the missing "y" property and the extra "z" property both stem from the same location in the instance is not immediately obvious in the "Basic" structure. In a hierarchy, the correlation is more easily identified.

The following rules govern the construction of the results object:

All applicator keywords ("*Of", "$ref", "if"/"then"/"else", etc.) require a node.

Nodes that have no children are removed.

Nodes that have a single child are replaced by the child.

Branch nodes do not require an error message or an annotation.

```
{
  "valid": false,
  "keywordLocation": "",
  "instanceLocation": "",
  "errors": [
    {
      "valid": false,
      "keywordLocation": "/items/$ref",
      "absoluteKeywordLocation":
        "https://example.com/polygon#/$defs/point",
      "instanceLocation": "/1",
      "errors": [
        {
          "valid": false,
          "keywordLocation": "/items/$ref/required",
          "absoluteKeywordLocation":
            "https://example.com/polygon#/$defs/point/required",
          "instanceLocation": "/1",
          "error": "Required property 'y' not found."
        },
        {
          "valid": false,
          "keywordLocation": "/items/$ref/additionalProperties",
          "absoluteKeywordLocation":
            "https://example.com/polygon#/$defs/point/additionalProperties",
          "instanceLocation": "/1/z",
          "error": "Additional property 'z' found but was invalid."
        }
      ]
    },
    {
      "valid": false,
      "keywordLocation": "/minItems",
      "instanceLocation": "",
      "error": "Expected at least 3 items but found 2"
    }
  ]
}
```

12.4.4. Verbose

The "Verbose" structure is a fully realized hierarchy that exactly matches that of the schema. This structure has applications in form generation and validation where the error's location is important.

The primary difference between this and the "Detailed" structure is that all results are returned. This includes sub-schema validation

results that would otherwise be removed (e.g. annotations for failed validations, successful validations inside a `not` keyword, etc.). Because of this, it is RECOMMENDED that each node also carry a `valid` property to indicate the validation result for that node.

Because this output structure can be quite large, a smaller example is given here for brevity. The URI of the full output structure of the example above is: <https://json-schema.org/draft/2020-12/output/verbose-example>.

```
// schema
{
  "$id": "https://example.com/polygon",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "validProp": true,
  },
  "additionalProperties": false
}

// instance
{
  "validProp": 5,
  "disallowedProp": "value"
}

// result
{
  "valid": false,
  "keywordLocation": "",
  "instanceLocation": "",
  "errors": [
    {
      "valid": true,
      "keywordLocation": "/type",
      "instanceLocation": ""
    },
    {
      "valid": true,
      "keywordLocation": "/properties",
      "instanceLocation": ""
    },
    {
      "valid": false,
      "keywordLocation": "/additionalProperties",
      "instanceLocation": "",
```

```
      "errors": [
        {
          "valid": false,
          "keywordLocation": "/additionalProperties",
          "instanceLocation": "/disallowedProp",
          "error": "Additional property 'disallowedProp' found but was
invalid."
        }
      ]
    }
  ]
}
```

12.4.5. Output validation schemas

For convenience, JSON Schema has been provided to validate output generated by implementations. Its URI is: <https://json-schema.org/draft/2020-12/output/schema>.

13. Security Considerations

Both schemas and instances are JSON values. As such, all security considerations defined in RFC 8259 [RFC8259] apply.

Instances and schemas are both frequently written by untrusted third parties, to be deployed on public Internet servers. Validators should take care that the parsing and validating against schemas does not consume excessive system resources. Validators MUST NOT fall into an infinite loop.

A malicious party could cause an implementation to repeatedly collect a copy of a very large value as an annotation. Implementations SHOULD guard against excessive consumption of system resources in such a scenario.

Servers MUST ensure that malicious parties cannot change the functionality of existing schemas by uploading a schema with a pre-existing or very similar "$id".

Individual JSON Schema vocabularies are liable to also have their own security considerations. Consult the respective specifications for more information.

Schema authors should take care with "$comment" contents, as a malicious implementation can display them to end-users in violation of a spec, or fail to strip them if such behavior is expected.

A malicious schema author could place executable code or other dangerous material within a "$comment". Implementations MUST NOT parse or otherwise take action based on "$comment" contents.

14. IANA Considerations

14.1. application/schema+json

The proposed MIME media type for JSON Schema is defined as follows:

Type name: application

Subtype name: schema+json

Required parameters: N/A

Optional parameters:

schema: A non-empty list of space-separated URIs, each identifying a JSON Schema resource. The instance SHOULD successfully validate against at least one of these meta-schemas. Non-validating meta-schemas MAY be included for purposes such as allowing clients to make use of older versions of a meta-schema as long as the runtime instance validates against that older version.

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type. See JSON [RFC8259].

Security considerations: See Section 13 above.

Interoperability considerations: See Sections 6.2, 6.3, and 6.4 above.

Fragment identifier considerations: See Section 5

14.2. application/schema-instance+json

The proposed MIME media type for JSON Schema Instances that require a JSON Schema-specific media type is defined as follows:

Type name: application

Subtype name: schema-instance+json

Required parameters:

schema: A non-empty list of space-separated URIs, each identifying a JSON Schema resource. The instance SHOULD successfully validate against at least one of these schemas. Non-validating schemas MAY be included for purposes such as allowing clients to make use of older versions of a schema as long as the runtime instance validates against that older version.

Encoding considerations: Encoding considerations are identical to those specified for the "application/json" media type. See JSON [RFC8259].

Security considerations: See Section 13 above.

Interoperability considerations: See Sections 6.2, 6.3, and 6.4 above.

Fragment identifier considerations: See Section 5

15. References

15.1. Normative References

[ecma262] "ECMA-262, 11th edition specification", June 2020, <https://www.ecma-international.org/ecma-262/11.0/index.html>.

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

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

[RFC6839] Hansen, T. and A. Melnikov, "Additional Media Type Structured Syntax Suffixes", RFC 6839, DOI 10.17487/RFC6839, January 2013, <https://www.rfc-editor.org/info/rfc6839>.

[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>.

[W3C.REC-ldp-20150226] Speicher, S., Arwe, J., and A. Malhotra, "Linked Data Platform 1.0", World Wide Web Consortium Recommendation REC-ldp-20150226, February 2015, <https://www.w3.org/TR/2015/REC-ldp-20150226>.

15.2. Informative References

[json-hyper-schema] Andrews, H. and A. Wright, "JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON", draft-handrews-json-schema-hyperschema-02 (work in progress), November 2017.

[json-schema-validation] Wright, A., Andrews, H., and B. Hutton, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", draft-bhutton-json-schema-validation-00 (work in progress), December 2020.

[RFC6596] Ohye, M. and J. Kupke, "The Canonical Link Relation", RFC 6596, DOI 10.17487/RFC6596, April 2012, <https://www.rfc-editor.org/info/rfc6596>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

[W3C.WD-fragid-best-practices-20121025] Tennison, J., "Best Practices for Fragment Identifiers and Media Type Definitions", World Wide Web Consortium WD WD-fragid-best-practices-20121025, October 2012, <https://www.w3.org/TR/2012/WD-fragid-best-practices-20121025>.

[xml-names]
Bray, T., Ed., Hollander, D., Ed., Layman, A., Ed., and R. Tobin, Ed., "Namespaces in XML 1.1 (Second Edition)", August 2006, <http://www.w3.org/TR/2006/REC-xml-names11-20060816>.

Appendix A. Schema identification examples

Consider the following schema, which shows "$id" being used to identify both the root schema and various subschemas, and "$anchor" being used to define plain name fragment identifiers.

```
{
    "$id": "https://example.com/root.json",
    "$defs": {
        "A": { "$anchor": "foo" },
        "B": {
            "$id": "other.json",
            "$defs": {
                "X": { "$anchor": "bar" },
                "Y": {
                    "$id": "t/inner.json",
                    "$anchor": "bar"
                }
            }
        },
        "C": {
            "$id": "urn:uuid:ee564b8a-7a87-4125-8c96-e9f123d6766f"
        }
    }
}
```

The schemas at the following URI-encoded JSON Pointers [RFC6901] (relative to the root schema) have the following base URIs, and are identifiable by any listed URI in accordance with sections 5 and 9.2.1 above.

```
# (document root)

   canonical absolute-URI (and also base URI)  https://example.com/
      root.json

   canonical URI with pointer fragment  https://example.com/
      root.json#

#/$defs/A

   base URI  https://example.com/root.json

   canonical URI with plain fragment  https://example.com/
      root.json#foo
```

```
      canonical URI with pointer fragment
      https://example.com/root.json#/$defs/A

   #/$defs/B

      base URI  https://example.com/other.json

      canonical URI with pointer fragment  https://example.com/
         other.json#

      non-canonical URI with fragment relative to root.json
      https://example.com/root.json#/$defs/B

   #/$defs/B/$defs/X

      base URI  https://example.com/other.json

      canonical URI with plain fragment  https://example.com/
         other.json#bar

      canonical URI with pointer fragment
      https://example.com/other.json#/$defs/X

   non-canonical URI with fragment relative to root.json
      https://example.com/root.json#/$defs/B/$defs/X

   #/$defs/B/$defs/Y

      base URI  https://example.com/t/inner.json

      canonical URI with plain fragment  https://example.com/t/
         inner.json#bar

      canonical URI with pointer fragment  https://example.com/t/
         inner.json#

      non-canonical URI with fragment relative to other.json
      https://example.com/other.json#/$defs/Y

   non-canonical URI with fragment relative to root.json
      https://example.com/root.json#/$defs/B/$defs/Y

   #/$defs/C

      base URI  urn:uuid:ee564b8a-7a87-4125-8c96-e9f123d6766f

      canonical URI with pointer fragment  urn:uuid:ee564b8a-
         7a87-4125-8c96-e9f123d6766f#
```

non-canonical URI with fragment relative to root.json
https://example.com/root.json#/$defs/C

Appendix B. Manipulating schema documents and references

Various tools have been created to rearrange schema documents based on how and where references ("$ref") appear. This appendix discusses which use cases and actions are compliant with this specification.

B.1. Bundling schema resources into a single document

A set of schema resources intended for use together can be organized with each in its own schema document, all in the same schema document, or any granularity of document grouping in between.

Numerous tools exist to perform various sorts of reference removal. A common case of this is producing a single file where all references can be resolved within that file. This is typically done to simplify distribution, or to simplify coding so that various invocations of JSON Schema libraries do not have to keep track of and load a large number of resources.

This transformation can be safely and reversibly done as long as all static references (e.g. "$ref") use URI-references that resolve to canonical URIs, and all schema resources have an absolute-URI as the "$id" in their root schema.

With these conditions met, each external resource can be copied under "$defs", without breaking any references among the resources' schema objects, and without changing any aspect of validation or annotation results. The names of the schemas under "$defs" do not affect behavior, assuming they are each unique, as they do not appear in canonical URIs for the embedded resources.

B.2. Reference removal is not always safe

Attempting to remove all references and produce a single schema document does not, in all cases, produce a schema with identical behavior to the original form.

Since "$ref" is now treated like any other keyword, with other keywords allowed in the same schema objects, fully supporting non-recursive "$ref" removal in all cases can require relatively complex schema manipulations. It is beyond the scope of this specification to determine or provide a set of safe "$ref" removal transformations, as they depend not only on the schema structure but also on the intended usage.

Appendix C. Example of recursive schema extension

Consider the following two schemas describing a simple recursive tree structure, where each node in the tree can have a "data" field of any type. The first schema allows and ignores other instance properties. The second is more strict and only allows the "data" and "children" properties. An example instance with "data" misspelled as "daat" is also shown.

```
// tree schema, extensible
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/tree",
    "$dynamicAnchor": "node",

    "type": "object",
    "properties": {
        "data": true,
        "children": {
            "type": "array",
            "items": {
                "$dynamicRef": "#node"
            }
        }
    }
}

// strict-tree schema, guards against misspelled properties
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "$id": "https://example.com/strict-tree",
    "$dynamicAnchor": "node",

    "$ref": "tree",
    "unevaluatedProperties": false
}

// instance with misspelled field
{
    "children": [ { "daat": 1 } ]
}
```

When we load these two schemas, we will notice the "$dynamicAnchor" named "node" (note the lack of "#" as this is just the name) present in each, resulting in the following full schema URIs:

- "https://example.com/tree#node"

- "https://example.com/strict-tree#node"

In addition, JSON Schema implementations keep track of the fact that these fragments were created with "$dynamicAnchor".

If we apply the "strict-tree" schema to the instance, we will follow the "$ref" to the "tree" schema, examine its "children" subschema, and find the "$dynamicRef": to "#node" (note the "#" for URI fragment syntax) in its "items" subschema. That reference resolves to "https://example.com/tree#node", which is a URI with a fragment created by "$dynamicAnchor". Therefore we must examine the dynamic scope before following the reference.

At this point, the dynamic path is "#/$ref/properties/children/items/$dynamicRef", with a dynamic scope containing (from the outermost scope to the innermost):

1. "https://example.com/strict-tree#"

2. "https://example.com/tree#"

3. "https://example.com/tree#/properties/children"

4. "https://example.com/tree#/properties/children/items"

Since we are looking for a plain name fragment, which can be defined anywhere within a schema resource, the JSON Pointer fragments are irrelevant to this check. That means that we can remove those fragments and eliminate consecutive duplicates, producing:

1. "https://example.com/strict-tree"

2. "https://example.com/tree"

In this case, the outermost resource also has a "node" fragment defined by "$dynamicAnchor". Therefore instead of resolving the "$dynamicRef" to "https://example.com/tree#node", we resolve it to "https://example.com/strict-tree#node".

This way, the recursion in the "tree" schema recurses to the root of "strict-tree", instead of only applying "strict-tree" to the instance root, but applying "tree" to instance children.

This example shows both "$dynamicAnchor"s in the same place in each schema, specifically the resource root schema. Since plain-name fragments are independent of the JSON structure, this would work just

as well if one or both of the node schema objects were moved under "$defs". It is the matching "$dynamicAnchor" values which tell us how to resolve the dynamic reference, not any sort of correlation in JSON structure.

Appendix D. Working with vocabularies

D.1. Best practices for vocabulary and meta-schema authors

Vocabulary authors should take care to avoid keyword name collisions if the vocabulary is intended for broad use, and potentially combined with other vocabularies. JSON Schema does not provide any formal namespacing system, but also does not constrain keyword names, allowing for any number of namespacing approaches.

Vocabularies may build on each other, such as by defining the behavior of their keywords with respect to the behavior of keywords from another vocabulary, or by using a keyword from another vocabulary with a restricted or expanded set of acceptable values. Not all such vocabulary re-use will result in a new vocabulary that is compatible with the vocabulary on which it is built. Vocabulary authors should clearly document what level of compatibility, if any, is expected.

Meta-schema authors should not use "$vocabulary" to combine multiple vocabularies that define conflicting syntax or semantics for the same keyword. As semantic conflicts are not generally detectable through schema validation, implementations are not expected to detect such conflicts. If conflicting vocabularies are declared, the resulting behavior is undefined.

Vocabulary authors SHOULD provide a meta-schema that validates the expected usage of the vocabulary's keywords on their own. Such meta-schemas SHOULD not forbid additional keywords, and MUST not forbid any keywords from the Core vocabulary.

It is recommended that meta-schema authors reference each vocabulary's meta-schema using the "allOf" (Section 10.2.1.1) keyword, although other mechanisms for constructing the meta-schema may be appropriate for certain use cases.

The recursive nature of meta-schemas makes the "$dynamicAnchor" and "$dynamicRef" keywords particularly useful for extending existing meta-schemas, as can be seen in the JSON Hyper-Schema meta-schema which extends the Validation meta-schema.

Meta-schemas may impose additional constraints, including describing keywords not present in any vocabulary, beyond what the meta-schemas

associated with the declared vocabularies describe. This allows for restricting usage to a subset of a vocabulary, and for validating locally defined keywords not intended for re-use.

However, meta-schemas should not contradict any vocabularies that they declare, such as by requiring a different JSON type than the vocabulary expects. The resulting behavior is undefined.

Meta-schemas intended for local use, with no need to test for vocabulary support in arbitrary implementations, can safely omit "$vocabulary" entirely.

D.2. Example meta-schema with vocabulary declarations

This meta-schema explicitly declares both the Core and Applicator vocabularies, together with an extension vocabulary, and combines their meta-schemas with an "allOf". The extension vocabulary's meta-schema, which describes only the keywords in that vocabulary, is shown after the main example meta-schema.

The main example meta-schema also restricts the usage of the Unevaluated vocabulary by forbidding the keywords prefixed with "unevaluated", which are particularly complex to implement. This does not change the semantics or set of keywords defined by the other vocabularies. It just ensures that schemas using this meta-schema that attempt to use the keywords prefixed with "unevaluated" will fail validation against this meta-schema.

Finally, this meta-schema describes the syntax of a keyword, "localKeyword", that is not part of any vocabulary. Presumably, the implementors and users of this meta-schema will understand the semantics of "localKeyword". JSON Schema does not define any mechanism for expressing keyword semantics outside of vocabularies, making them unsuitable for use except in a specific environment in which they are understood.

This meta-schema combines several vocabularies for general use.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/meta/general-use-example",
  "$dynamicAnchor": "meta",
  "$vocabulary": {
    "https://json-schema.org/draft/2020-12/vocab/core": true,
    "https://json-schema.org/draft/2020-12/vocab/applicator": true,
    "https://json-schema.org/draft/2020-12/vocab/validation": true,
    "https://example.com/vocab/example-vocab": true
  },
  "allOf": [
    {"$ref": "https://json-schema.org/draft/2020-12/meta/core"},
    {"$ref": "https://json-schema.org/draft/2020-12/meta/applicator"},
    {"$ref": "https://json-schema.org/draft/2020-12/meta/validation"},
    {"$ref": "https://example.com/meta/example-vocab",
  ],
  "patternProperties": {
    "^unevaluated": false
  },
  "properties": {
    "localKeyword": {
      "$comment": "Not in vocabulary, but validated if used",
      "type": "string"
    }
  }
}
```

This meta-schema describes only a single extension vocabulary.

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/meta/example-vocab",
  "$dynamicAnchor": "meta",
  "$vocabulary": {
    "https://example.com/vocab/example-vocab": true,
  },
  "type": ["object", "boolean"],
  "properties": {
    "minDate": {
      "type": "string",
      "pattern": "\d\d\d\d-\d\d-\d\d",
      "format": "date",
    }
  }
}
```

As shown above, even though each of the single-vocabulary meta-schemas referenced in the general-use meta-schema's "allOf" declares its corresponding vocabulary, this new meta-schema must re-declare them.

The standard meta-schemas that combine all vocabularies defined by the Core and Validation specification, and that combine all vocabularies defined by those specifications as well as the Hyper-Schema specification, demonstrate additional complex combinations. These URIs for these meta-schemas may be found in the Validation and Hyper-Schema specifications, respectively.

While the general-use meta-schema can validate the syntax of "minDate", it is the vocabulary that defines the logic behind the semantic meaning of "minDate". Without an understanding of the semantics (in this example, that the instance value must be a date equal to or after the date provided as the keyword's value in the schema), an implementation can only validate the syntactic usage. In this case, that means validating that it is a date-formatted string (using "pattern" to ensure that it is validated even when "format" functions purely as an annotation, as explained in the Validation specification [json-schema-validation].

Appendix E. References and generative use cases

While the presence of references is expected to be transparent to validation results, generative use cases such as code generators and UI renderers often consider references to be semantically significant.

To make such use case-specific semantics explicit, the best practice is to create an annotation keyword for use in the same schema object alongside of a reference keyword such as "$ref".

For example, here is a hypothetical keyword for determining whether a code generator should consider the reference target to be a distinct class, and how those classes are related. Note that this example is solely for illustrative purposes, and is not intended to propose a functional code generation keyword.

```
{
    "allOf": [
        {
            "classRelation": "is-a",
            "$ref": "classes/base.json"
        },
        {
            "$ref": "fields/common.json"
        }
    ],
    "properties": {
        "foo": {
            "classRelation": "has-a",
            "$ref": "classes/foo.json"
        },
        "date": {
            "$ref": "types/dateStruct.json",
        }
    }
}
```

Here, this schema represents some sort of object-oriented class. The first reference in the "allOf" is noted as the base class. The second is not assigned a class relationship, meaning that the code generator should combine the target's definition with this one as if no reference were involved.

Looking at the properties, "foo" is flagged as object composition, while the "date" property is not. It is simply a field with sub-fields, rather than an instance of a distinct class.

This style of usage requires the annotation to be in the same object as the reference, which must be recognizable as a reference.

Appendix F. 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, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave Finlay, Denis Laxalde, Phil Sturgeon, Shawn Silverman, and Karen Etheridge for their submissions and patches to the document.

Appendix G. ChangeLog

[[CREF13: This section to be removed before leaving Internet-Draft status.]]

draft-bhutton-json-schema-00

* "$schema" MAY change for embedded resources
* Array-value "items" functionality is now "prefixItems"
* "items" subsumes the old function of "additionalItems"
* "contains" and "unevaluatedItems" interactions now specified
* Rename $recursive* to $dynamic*
* $dynamicAnchor defines a fragment like $anchor
* $dynamic* (previously $recursive) no longer use runtime base URI determination
* Define Compound Schema Documents (bundle) and processing
* Reference ECMA-262, 11th edition for regular expression support
* Regular expression should support unicode
* Remove media type parameters
* Specify Unknown keywords are collected as annotations

* Moved "unevaluatedItems" and "unevaluatedProperties" from core into their own vocabulary

draft-handrews-json-schema-02

* Update to RFC 8259 for JSON specification

* Moved "definitions" from the Validation specification here as "$defs"

* Moved applicator keywords from the Validation specification as their own vocabulary

* Moved the schema form of "dependencies" from the Validation specification as "dependentSchemas"

* Formalized annotation collection

* Specified recommended output formats

* Defined keyword interactions in terms of annotation and assertion results

* Added "unevaluatedProperties" and "unevaluatedItems"

* Define "$ref" behavior in terms of the assertion, applicator, and annotation model

* Allow keywords adjacent to "$ref"

* Note undefined behavior for "$ref" targets involving unknown keywords

* Add recursive referencing, primarily for meta-schema extension

* Add the concept of formal vocabularies, and how they can be recognized through meta-schemas

* Additional guidance on initial base URIs beyond network retrieval

* Allow "schema" media type parameter for "application/schema+json"

* Better explanation of media type parameters and the HTTP Accept header

* Use "$id" to establish canonical and base absolute-URIs only, no fragments

* Replace plain-name-fragment-only form of "$id" with "$anchor"

* Clarified that the behavior of JSON Pointers across "$id" boundary is unreliable

draft-handrews-json-schema-01

* This draft is purely a clarification with no functional changes

* Emphasized annotations as a primary usage of JSON Schema

* Clarified $id by use cases

* Exhaustive schema identification examples

* Replaced "external referencing" with how and when an implementation might know of a schema from another document

* Replaced "internal referencing" with how an implementation should recognized schema identifiers during parsing

* Dereferencing the former "internal" or "external" references is always the same process

* Minor formatting improvements

draft-handrews-json-schema-00

* Make the concept of a schema keyword vocabulary more clear

* Note that the concept of "integer" is from a vocabulary, not the data model

* Classify keywords as assertions or annotations and describe their general behavior

* Explain the boolean schemas in terms of generalized assertions

* Reserve "$comment" for non-user-visible notes about the schema

* Wording improvements around "$id" and fragments

* Note the challenges of extending meta-schemas with recursive references

* Add "application/schema-instance+json" media type

* Recommend a "schema" link relation / parameter instead of "profile"

draft-wright-json-schema-01

* Updated intro

* Allowed for any schema to be a boolean

* "$schema" SHOULD NOT appear in subschemas, although that may change

* Changed "id" to "$id"; all core keywords prefixed with "$"

* Clarify and formalize fragments for application/schema+json

* Note applicability to formats such as CBOR that can be represented in the JSON data model

draft-wright-json-schema-00

* Updated references to JSON

* Updated references to HTTP

* Updated references to JSON Pointer

* Behavior for "id" is now specified in terms of RFC3986

* Aligned vocabulary usage for URIs with RFC3986

* Removed reference to draft-pbryan-zyp-json-ref-03

* Limited use of "$ref" to wherever a schema is expected

* Added definition of the "JSON Schema data model"

* Added additional security considerations

* Defined use of subschema identifiers for "id"

* Rewrote section on usage with HTTP

* Rewrote section on usage with rel="describedBy" and rel="profile"

* Fixed numerous invalid examples

draft-zyp-json-schema-04

* Salvaged from draft v3.

* Split validation keywords into separate document.

* Split hypermedia keywords into separate document.

* Initial post-split draft.

* Mandate the use of JSON Reference, JSON Pointer.

* Define the role of "id". Define URI resolution scope.

* Add interoperability considerations.

draft-zyp-json-schema-00

* Initial draft.

Authors' Addresses

Austin Wright (editor)

EMail: aaa@bzfx.net

Henry Andrews (editor)

EMail: andrews_henry@yahoo.com

Ben Hutton (editor)

EMail: ben@jsonschema.dev
URI: https://jsonschema.dev

Greg Dennis

EMail: gregsdennis@yahoo.com
URI: https://github.com/gregsdennis