Network Working Group
Internet-Draft
Intended status: Informational
Expires: 13 November 2026

L. Dusseault
Data Transfer Initiative
A. Wright

H. Andrews
12 May 2026

JSON Schema
draft-ietf-jsonschema-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.

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 13 November 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components

extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction

...

JSON Schemas are JSON documents that describe and constrain other JSON documents. JSON Schema defines 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.

The status of this document is literally a draft, and it is not ready for implementors to adopt in place of widely implemented versions of JSON Schema. It is a proposed starting point for a WG and a wide consensus process.

1.1. Notational Conventions

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 [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 [RFC8259].

1.2. Functionality

A JSON Schema implementation takes input in the form of a JSON-compatible data structure and performs validation and, if the input is accepted as a valid instance, returns a standard annotation output.

1.2.1. Validation

A JSON Schema document (a _schema_) notates a grammar that describes a language of JSON documents: That is, it describes a set of JSON documents by listing rules to classify an _input_, some arbitrary JSON document, as within or not within the set. The largest possible set is that of all JSON documents (all documents that are valid application/json). The smallest set is the empty set.

A _validator_ (also known as an _acceptor_) is a process that tests if a particular input is described by the schema, by evaluating it against the requirements. An accepted input is called an _instance_ of the schema, and a rejected input is called a _violation_ of the schema.

In all validator implementations, equivalent JSON values MUST return the same validation result and annotations. For example, selection between equivalent character escapes, or use of whitespace, does not affect the result. This also means that a validation keyword MUST NOT accept or reject based on third factors (those factors besides the schema and the input itself). A separate "outside verification" scheme that queries external data sources is possible, but outside the scope of this document.

1.2.2. Annotation

A JSON Schema may describe additional output to accompany an "accept" result, called _annotations_. An annotation is a tuple that, at the minimum, relates a particular annotation keyword in the schema to a particular value within the instance. Annotations typically document the meaning of properties, declare relationships between data, or denote hyperlinks. Several forms of output are defined, and the annotation process can be disabled as a performance or resource consumption optimization.

Output annotations might only be "as true as" the input, and useful only for select inputs. For example, annotations may only meaningfully describe inputs with a particular "profile" link relation, or in some particular context. In any event, annotations never describe violations (rejected inputs).

The interface to access annotations may be highly configurable depending on the implementation, in such ways as limiting output to certain annotation keywords, aggregating values together, or other features to enhance performance. Annotation output may be bypassed entirely.

Annotations may be presented as a set, or as a stream of events, however if the input is rejected during processing, this voids all annotations previously emitted from that input.

1.2.3. Internet media types

The specification registers the "application/schema+json" media type to identify a JSON Schema resource, and the "application/schema-instance+json" media type to identify an instance of a particular JSON Schema.

1.2.4. Keywords

JSON Schema uses _keywords_ to assert constraints on JSON documents or provide annotations with additional information. Additional keywords combine other keywords or provide references to sub-schemas, features which allow more complex JSON data structures.

In formal language theory, JSON Schema resembles a context-free language, as most keywords are context-free. Thus, JSON instances can be validated quickly and simply, without I/O or querying data elsewhere in the same JSON instance.

This document defines a core vocabulary of keywords that MUST be supported by any implementation. 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.

1.2.5. Vocabularies

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. Vocabularies, dialects and meta-schemas are not required features for most schema authors to understand, but must be understood by authors of meta-schemas and handled by implementors of validation or other JSON Schema processing software.

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 treated as annotations, while the behavior with respect to an unrecognized vocabulary can be controlled when declaring which vocabularies are in use.

2. Terminology

These terms are all defined in the context of JSON Schema. These quick definitions may not be enough to completely comprehend roles and uses, but may still provide a useful quick reference.

JSON Document

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

Input

A JSON document supplied to a validator or other implementation, in order to compare it to a schema, is an input until it is known to be in the valid set for that schema.

Instance

A JSON document that is in the valid set for a given schema is an _instance_ of that schema.

Object

Object is defined for JSON in [RFC8259] and has the same meaning here.

Schema / Schema Document

A JSON Schema document, or simply a _schema_, is a JSON document used to describe and constrain JSON Documents. Used in validation, the schema defines the valid set, or all possible instances that validate successfully. As a JSON document, a schema may also be an instance of some meta-schema.

Schema Resource

A JSON _Schema resource_ is a schema uniquely identified by a URI, in contrast to an anonymous schema which has no URI. A schema is canonically identified (in the sense of [RFC6596]) by an absolute URI ([RFC3986], Section 4.3).

Keyword

JSON Schema works by defining keywords with specific behavior. A _keyword_ appears as a JSON _name_ [RFC8259]. Not all JSON names in a schema are keywords, but all keywords have required behavior.

Vocabulary

A _vocabulary_ is a set of keywords that are defined to enable some functionality, particularly when JSON Schema is extended with external vocabularies.

Meta-Schema

A _meta-schema_ describes and constrains schemas which may be instances of the meta-schema.

Root schema

The root schema is the top-level schema object that serves as the starting point and container for all of a schema's rules and annotations. It typically defines the Base URI for the entire document.

Subschema

A subschema is a schema that happens to be located inside another schema. Structural keywords like 'items' and 'properties' use subschemas to describe structure inside arrays and objects. Aggregation keywords like 'anyOf' and 'not' direct how to apply the subschemas they hold.

A subschema can be directly inside its parent schema, can be elsewhere in the Schema Document (using a fragment reference), or in a separate document (using an absolute reference).

Implementation

An implementation is software that implements this specification. Some implementations validate inputs, but some implementations generate documentation, data or code based on a schema.

3. Overall Definitions and Requirements

This section is the start of normative requirements especially for implementations. However, a practitioner may wish to check the basic meaning of terms defined above and skip toSection 12 where specific keywords and purposes begin to be defined.

3.1. JSON Schema Documents

A JSON Schema, with its specific rules or constraints to describe and validate JSON data, is represented in digital form as a JSON schema document.

A JSON Schema MUST be an _object_ or a boolean. When a schema is an object, it can be seen to be a schema from the ’$schema’ keyword in the object. See below for special treatment of the boolean schemas.

Schemas are used to validate inputs that may be instances (Section 3.2) of the schema, but each schema can itself be interpreted as an instance (see Section 14.3 for when and how this happens).

Schema documents SHOULD always be given the media type "application/schema+json", even when playing the role of an instance.

3.1.1. Trivial schema documents

The schema values "true" and "false" are trivial schemas. The valid set for the ’true’ schema is all possible JSON documents, and the valid set for the ’false’ schema is empty. The trivial 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.

3.1.2. Root Schema and Subschemas and Resources

A JSON Schema resource is a schema which is canonically identified (in the sense of [RFC6596]) by an absolute URI ([RFC3986], Section 4.3). Schema resources MAY also be identified by URIs, including URIs with fragments, if the resulting secondary resource (as defined by [RFC3986], Section 3.5) is identical to the primary resource. This can occur with the empty fragment, or when one schema resource is embedded in another. Any such URIs with fragments are considered to be non-canonical.

The root schema is always a schema resource, where the URI is determined as described in Section 11.1.1.
// Note that documents that embed schemas in another format will not
// have a root schema resource in this sense. Exactly how such
// usages fit with the JSON Schema document and resource concepts
// will be clarified in a future draft.

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 4.1.3, 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.

3.2. Instance

An instance of a schema is a JSON value or document that is in the valid set of a schema. 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 JSON "true" or "false" value

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

array An ordered list of instances, with 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 outside the scope of JSON Schema. JSON Schema Section 1.2.5 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 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.

3.2.1. Input Equality

Two JSON inputs are said to be equal if and only if they are of the same type and have the same value according to the JSON 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. Two equal inputs are guaranteed to yield identical validation results for a given schema, regardless of their original formatting.

3.3. 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 a URI for the schema 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 or are not recognized as schema keywords. The behavior of such keywords is governed by Section 12.4.

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.

3.4. Fragment Identifiers

In accordance with section 3.1 of [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 [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 4.1.4) section.

3.5. Other General Considerations

3.5.1. 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.

3.5.2. Requirements for handling extensions

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.

3.5.3. Validation

JSON Schema validation applies the rules of a JSON Schema to determine if an input is in the valid set for that schema. An instance location that satisfies all asserted constraints is then annotated with any keywords that contain non-assertion information, such as descriptive metadata and usage hints.

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

This specification defines a set of assertion keywords, as well as a small vocabulary of metadata keywords that can be used to annotate the JSON instance with useful information. The Section 8 keyword is intended primarily as an annotation, but can optionally be used as an assertion. The Section 9 keywords are annotations for working with documents embedded as JSON strings.

4. Core Keywords

Core keywords MUST be implemented by any processor indicating support for the "application/jsonschema+json" media type.

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.

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 (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.

4.1. Environment

Environment keywords MUST be read by implementations before other keywords may be evaluated, as they are capable of impacting the behavior of other keywords.

4.1.1. "$schema"

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 (containing a scheme, per [RFC3986], Section 3) 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.

4.1.2. "$vocabulary"

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.
// 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 Section 3.5.2, 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 resource intended for use as a meta-schema. It MUST NOT appear in subschemas.

The "$vocabulary" keyword MUST be ignored in schema resources 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 implementation to understand the vocabularies declared by M.

4.1.2.1. 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.

4.1.2.2. 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).
// 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.

4.1.2.3. 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.

4.1.3. "$id"

The $id keyword identifies a schema resource with its _canonical URI_ (in the sense of [RFC6596]). Explicit identification makes it easier for a schema to be referenced, especially from other schemas, allowing re-use, modularity and extensibility of schemas. An explicit identifier is a more stable way to reference a schema than its location (either its URL on the Web or as a location within the structure of a parent schema).

Requirements on $id value:

* MUST be a string

* MUST be a valid URI reference ([RFC3986], Section 4.1)

* SHOULD be normalized

* MUST NOT have a fragment ([RFC3986], Section 3.5)

The value of an $id may be a full URI or a _relative reference_ ([RFC3986], Section 4.2). When a relative URI is used, knowing the base URI becomes important. Read Section 11.2.1 and Appendix A to understand how that must be done.

Note that an URI does not have to be a URL. Even if it is a URL, the URL may not resolve and return a schema, and implementations are warned against automatically resolving network references to fetch schemas (see Section 11.1.2). Nevertheless, the URI still identifies the schema.

The presence of "$id" in a subschema indicates that the subschema constitutes a distinct schema resource within a single schema document.

See also the $anchor keyword (Section 4.1.4) for naming subschemas, and the $ref keyword (Section 4.2.1) in which $id and $anchor values are frequently used.

4.1.3.1. Identifying the root schema

The root schema of a JSON Schema document SHOULD contain an "$id" keyword with an absolute-URI ([RFC3986], Section 4.3; containing a scheme, but no fragment).

4.1.4. "$anchor" and "$dynamicAnchor"

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.

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 4.2.1) 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, per [XMLNS].
// 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.

4.2. Definitions and References

4.2.1. "$ref" and "$dynamicRef"

The "$ref" and "$dynamicRef" keywords are applicator keywords used to reference a schema. Their results are the results of the referenced schema.
// Note that this definition of how the results are determined means
// that other keywords can appear alongside of "$ref" in the same
// schema object.

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. Even if it is a network locator, implementations should refer to Section 11.1.2 about loading such schemas.

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 input cannot change how the reference resolves.

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 input.

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 "$dynamicAnchor" keyword, the initial URI MUST be replaced by the URI (including the fragment) for the outermost schema resource in the dynamic scope (Section 12.1) that defines an identically named fragment with "$dynamicAnchor".

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

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

4.2.2. "$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
        }
    }
}
```

4.3. "$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.

5. Subschema keywords

This section defines 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 (https://json-schema.org/draft/2020-12/meta/applicator).

5.1. 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 affected by the presence and value of "minContains", in the Validation vocabulary

5.2. Keywords for Applying Subschemas in Place

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

Subschemas of these keywords evaluate the input 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.

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.

5.2.1. "allOf"

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

An input validates successfully against this keyword if it validates successfully against all schemas defined by this keyword’s value.

5.2.2. "anyOf"

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

An input 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.

5.2.3. "oneOf"

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

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

5.2.4. "not"

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

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

5.2.5. "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.

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

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

If Section 12.8 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".

5.2.6. "then"

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

When "if" is present, and the input successfully validates against its subschema, then validation succeeds against this keyword if the input also successfully validates against this keyword’s subschema.

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

5.2.7. "else"

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

When "if" is present, and the input fails to validate against its subschema, then validation succeeds against this keyword if the input successfully validates against this keyword’s subschema.

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

5.2.8. "dependentSchemas"

This keyword specifies subschemas that are evaluated if the input 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.

5.3. Keywords for Applying Subschemas to Arrays

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

5.3.1. "prefixItems"

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

Validation succeeds if each element of the input 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.

5.3.2. "items"

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

This keyword applies its subschema to all input 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 input array elements.

```
// 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 input array, it produces an annotation result of boolean true, indicating that all remaining array elements have been evaluated against this keyword's subschema. This annotation affects the behavior of "unevaluatedItems" in the Unevaluated vocabulary.

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.

5.3.3. "contains"

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

An array input is valid against "contains" if the number of elements that are valid against its subschema is within the inclusive range of the minimum and (if any) maximum number of occurrences.

The minimum and maximum numbers of occurrences are provided by the "minContains" and "maxContains" keywords, respectively, within the same schema object as "contains". If "minContains" is absent, the minimum MUST be 1. If "maxContains" is absent, the maximum MUST be unbounded.

Implementations MAY implement the dependency on "minContains" and "maxContains" by inspecting their values rather than by reading annotations provided by those keywords.

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 input array to which this keyword's schema applies is empty.

This annotation affects the behavior of "unevaluatedItems" in the Unevaluated vocabulary.

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.

5.4. Keywords for Applying Subschemas to Objects

Each of these keywords defines a rule for applying its subschema(s) to object properties and combining their results.

5.4.1. "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 input and as a name within this keyword’s value, the contents successfully validate against the corresponding schema.

The annotation result of this keyword is the set of instance property names matched by this keyword. This annotation affects the behavior of "additionalProperties" (in this vocabulary) and "unevaluatedProperties" in the Unevaluated vocabulary.

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

5.4.2. "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 input name that matches any regular expressions that appear as a property name in this keyword’s value, the contents successfully validate against each schema that corresponds to a matching regular expression. Recall: Regular expressions are not explicitly anchored.

The annotation result of this keyword is the set of instance property names matched by this keyword. This annotation affects the behavior of "additionalProperties" (in this vocabulary) and "unevaluatedProperties" (in the Unevaluated vocabulary).

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

5.4.3. "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 input names that do not appear in the annotation results of either "properties" or "patternProperties".

For all such properties, validation succeeds if the contents validate against the "additionalProperties" schema.

The annotation result of this keyword is the set of input property names validated by this keyword's subschema. This annotation affects the behavior of "unevaluatedProperties" in the Unevaluated vocabulary.

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 input property set. Implementations that do not support annotation collection MUST do so.
// In defining this option, it seems there is the potential for
// ambiguity in the output format. The ambiguity does not affect
// validation results, but it does affect the resulting output
// format. The ambiguity allows for multiple valid output results
// depending on whether annotations are used or a solution that
// "produces the same effect" as draft-07. It is understood that
// annotations from failing schemas are dropped. See our Decision
// Record (https://github.com/json-schema-org/json-schema-
// spec/tree/HEAD/adr/2022-04-08-cref-for-ambiguity-and-fix-later-gh-
// spec-issue-1172.md) for further details.

5.4.4. "propertyNames"

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

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

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

6. Keywords 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 input 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 input 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 input 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 (https://json-schema.org/draft/2020-12/meta/unevaluated).

6.1. 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

6.2. "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 input 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 5.2) 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 input array, it produces an annotation result of boolean true, analogous to the behavior of "items". This annotation affects the behavior of "unevaluatedItems" in parent schemas.

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

6.3. "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 input 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 5.2) 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 input 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 contents validate 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. This annotation affects the behavior of "unevaluatedProperties" in parent schemas.

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

7. Keywords for Structural Validation

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

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

The current URI for this vocabulary, known as the Validation vocabulary, is:

<https://json-schema.org/draft/2020-12/vocab/validation>.

The current URI for the corresponding meta-schema is:

https://json-schema.org/draft/2020-12/meta/validation (https://json-schema.org/draft/2020-12/meta/validation).

7.1. Validation Keywords for Any Instance Type

7.1.1. "type"

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

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

If the value of "type" is a string, then an input validates successfully if its type matches the type represented by the value of the string.

If the value of "type" is an array, then an input validates successfully if its type matches any of the types indicated by the strings in the array.

7.1.2. "enum"

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

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

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

7.1.3. "const"

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

Use of this keyword is functionally equivalent to an Section 7.1.2 with a single value.

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

7.2. Validation Keywords for Numeric Inputs (number and integer)

7.2.1. "multipleOf"

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

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

7.2.2. "maximum"

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

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

7.2.3. "exclusiveMaximum"

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

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

7.2.4. "minimum"

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

If the input value is a number, then it is valid only if it has a value that is greater than or exactly equal to "minimum".

7.2.5. "exclusiveMinimum"

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

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

7.3. Validation Keywords for Strings

7.3.1. "maxLength"

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

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

The length of a string input value is defined as the number of its characters as defined by [RFC8259].

7.3.2. "minLength"

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

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

The length of a string input value is defined as the number of its characters as defined by [RFC8259].

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

7.3.3. "pattern"

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

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

7.4. Validation Keywords for Arrays

7.4.1. "maxItems"

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

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

7.4.2. "minItems"

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

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

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

7.4.3. "uniqueItems"

The value of this keyword MUST be a boolean.

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

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

7.4.4. "maxContains"

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

Validation MUST always succeed against this keyword; its validation effect is to modify the behavior of "contains" within the same schema object, as described in that keyword’s section.

This keyword behaves as an annotation, which MAY be used by "Section 5.3.3".

7.4.5. "minContains"

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

Validation MUST always succeed against this keyword; its validation effect is to modify the behavior of "contains" within the same schema object, as described in that keyword’s section.

This keyword behaves as an annotation, which MAY be used by "Section 5.3.3".

Omitting this keyword has the same behavior as a value of 1. Per Section 12.3, omitted keywords MUST NOT produce annotation results. However, as described in the section for "contains", the absence of this keyword’s annotation causes "contains" to assume a value of 1.

7.5. Validation Keywords for Objects

7.5.1. "maxProperties"

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

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

7.5.2. "minProperties"

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

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

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

7.5.3. "required"

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

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

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

7.5.4. "dependentRequired"

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

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

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

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

8. Vocabularies for Semantic Content With "format"

8.1. Foreword

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

The value of this keyword is called a format attribute. It MUST be a string. A format attribute can generally only validate a given set of input types. If the type of the input is not in this set, validation for this format attribute and input SHOULD succeed. All format attributes defined in this section apply to strings, but a format attribute can be specified to apply to any input type in the instance data model (Section 3.2).
// Note that the "type" keyword in this specification defines an
// "integer" type which is not part of the data model. Therefore a
// format attribute can be limited to numbers, but not specifically
// to integers. However, a numeric format can be used alongside the
// "type" keyword with a value of "integer", or could be explicitly
// defined to always pass if the number is not an integer, which
// produces essentially the same behavior as only applying to
// integers.

The current URI for this vocabulary, known as the Format-Annotation vocabulary, is:

<https://json-schema.org/draft/2020-12/vocab/format-annotation>.

The current URI for the corresponding meta-schema is:

https://json-schema.org/draft/2020-12/meta/format-annotation (https://json-schema.org/draft/2020-12/meta/format-annotation).

Implementing support for this vocabulary is REQUIRED.

In addition to the Format-Annotation vocabulary, a secondary vocabulary is available for custom meta-schemas that defines "format" as an assertion. The URI for the Format-Assertion vocabulary, is:

<https://json-schema.org/draft/2020-12/vocab/format-assertion>.

The current URI for the corresponding meta-schema is:

https://json-schema.org/draft/2020-12/meta/format-assertion (https://json-schema.org/draft/2020-12/meta/format-assertion).

Implementing support for the Format-Assertion vocabulary is OPTIONAL.

Specifying both the Format-Annotation and the Format-Assertion vocabularies is functionally equivalent to specifying only the Format-Assertion vocabulary since its requirements are a superset of the Format-Annotation vocabulary.

8.2. Implementation Requirements

The "format" keyword functions as defined by the vocabulary which is referenced.

8.2.1. Format-Annotation Vocabulary

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

Implementations MAY still treat "format" as an assertion in addition to an annotation and attempt to validate the value's conformance to the specified semantics. The implementation MUST provide options to enable and disable such evaluation and MUST be disabled by default. Implementations SHOULD document their level of support for such validation.
// Specifying the Format-Annotation vocabulary and enabling
// validation in an implementation should not be viewed as being
// equivalent to specifying the Format-Assertion vocabulary since
// implementations are not required to provide full validation
// support when the Format-Assertion vocabulary is not specified.

When the implementation is configured for assertion behavior, it:

* SHOULD provide an implementation-specific best effort validation for each format attribute defined below;

* MAY choose to implement validation of any or all format attributes as a no-op by always producing a validation result of true;

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

8.2.2. Format-Assertion Vocabulary

When the Format-Assertion vocabulary is declared with a value of true, implementations MUST provide full validation support for all of the formats defined by this specificaion. Implementations that cannot provide full validation support MUST refuse to process the schema.

An implementation that supports the Format-Assertion vocabulary:

* MUST still collect "format" as an annotation if the implementation supports annotation collection;

* MUST evaluate "format" as an assertion;

* MUST implement syntactic validation for all format attributes defined in this specification, and for any additional format attributes that it recognizes, such that there exist possible input values of the correct type that will fail validation.

The requirement for minimal validation of format attributes is intentionally vague and permissive, due to the complexity involved in many of the attributes. Note in particular that the requirement is limited to syntactic checking; it is not to be expected that an implementation would send an email, attempt to connect to a URL, or otherwise check the existence of an entity identified by a format instance.
// The expectation is that for simple formats such as date-time,
// syntactic validation will be thorough. For a complex format such
// as email addresses, which are the amalgamation of various
// standards and numerous adjustments over time, with obscure and/or
// obsolete rules that may or may not be restricted by other
// applications making use of the value, a minimal validation is
// sufficient. For example, an input string that does not contain an
// "@" is clearly not a valid email address, and an "email" or
// "hostname" containing characters outside of 7-bit ASCII is
// likewise clearly invalid.

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

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

8.2.3. Custom format attributes

Implementations MAY support custom format attributes. Save for agreement between parties, schema authors SHALL NOT expect a peer implementation to support such custom format attributes. An implementation MUST NOT fail to collect unknown formats as annotations. When the Format-Assertion vocabulary is specified, implementations MUST fail upon encountering unknown formats.

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

8.3. Defined Formats

8.3.1. Dates, Times, and Duration

These attributes apply to string inputs.

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

Implementations supporting formats SHOULD implement support for the following attributes:

8.3.1.1. "date-time"

A string input is valid against this attribute if it is a valid representation according to the "date-time’ ABNF rule (referenced above).

8.3.1.2. "date"

A string input is valid against this attribute if it is a valid representation according to the "full-date" ABNF rule (referenced above).

8.3.1.3. "time"

A string input is valid against this attribute if it is a valid representation according to the "full-time" ABNF rule (referenced above).

8.3.1.4. "duration"

A string input is valid against this attribute if it is a valid representation according to the "duration" ABNF rule (referenced above).

8.3.1.5. Additional RFC3339 Formats

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

8.3.2. Email Addresses

These attributes apply to string inputs.

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

8.3.2.1. "email"

As defined by the "Mailbox" ABNF rule in [RFC5321], Section 4.1.2.

8.3.2.2. "idn-email"

As defined by the extended "Mailbox" ABNF rule in [RFC6531], Section 3.3.

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

8.3.3. Hostnames

These attributes apply to string inputs.

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

8.3.3.1. "hostname"

As defined by [RFC1123], Section 2.1, including host names produced using the Punycode algorithm specified in [RFC5891], Section 4.4.

8.3.3.2. "idn-hostname"

As defined by either RFC 1123 as for hostname, or an internationalized hostname as defined by [RFC5890], Section 2.3.2.3.

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

8.3.4. IP Addresses

These attributes apply to string inputs.

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

8.3.4.1. "ipv4"

An IPv4 address according to the "dotted-quad" ABNF syntax as defined in [RFC2673], Section 3.2.

8.3.4.2. "ipv6"

An IPv6 address as defined in [RFC4291], Section 2.2.

8.3.5. Resource Identifiers

These attributes apply to string inputs.

8.3.5.1. "uri"

A string input is valid against this attribute if it is a valid URI, according to [RFC3986], Section 3.

8.3.5.2. "uri-reference"

A string input is valid against this attribute if it is a valid URI Reference (either a URI or a relative-reference), according to [RFC3986], Section 4.

8.3.5.3. "iri"

A string input is valid against this attribute if it is a valid IRI, according to [RFC3987], Section 2.2.

8.3.5.4. "iri-reference"

A string input is valid against this attribute if it is a valid IRI Reference (either an IRI or a relative-reference), according to [RFC3987], Section 2.2.

8.3.5.5. "uuid"

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

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

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

8.3.6. Templates

8.3.6.1. "uri-template"

This attribute applies to string inputs.

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

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

8.3.7. JSON Pointers

These attributes apply to string inputs.

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

8.3.7.1. "json-pointer"

A string input is valid against this attribute if it is a valid JSON string representation of a JSON Pointer, according to [RFC6901], Section 5.

8.3.7.2. "relative-json-pointer"

A string input is valid against this attribute if it is a valid [I-D.hha-relative-json-pointer].

8.3.8. Expressions

8.3.8.1. "regex"

This attribute applies to string inputs.

A regular expression, which SHOULD be valid according to the [ECMA262] regular expression dialect.

Implementations that validate formats MUST accept at least the subset of ECMA-262 defined in Regular Expressions (Section 8.3.8.1) section of this specification, and SHOULD accept all valid ECMA-262 expressions.

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

9.1. Foreword

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

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

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

The current URI for this vocabulary, known as the Content vocabulary, is:

<https://json-schema.org/draft/2020-12/vocab/content>.

The current URI for the corresponding meta-schema is:

https://json-schema.org/draft/2020-12/meta/content (https://json-schema.org/draft/2020-12/meta/content).

9.2. Implementation Requirements

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

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

Implementations MAY offer the ability to decode, parse, and/or validate the string contents automatically. However, it MUST NOT perform these operations by default, and MUST provide the validation result of each string-encoded document separately from the enclosing document. This process SHOULD be equivalent to fully evaluating the input against the original schema, followed by using the annotations to decode, parse, and/or validate each string-encoded document.
// For now, the exact mechanism of performing and returning parsed
// data and/or validation results from such an automatic decoding,
// parsing, and validating feature is left unspecified. Should such
// a feature prove popular, it may be specified more thoroughly in a
// future draft.

See also the Security Considerations (Section 15) sections for possible vulnerabilities introduced by automatically processing inputs according to these keywords.

9.3. "contentEncoding"

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

Possible values indicating base 16, 32, and 64 encodings with several variations are listed in [RFC4648]. Additionally, sections 6.7 and 6.8 of [RFC2045] provide encodings used in MIME. This keyword is derived from MIME's Content-Transfer-Encoding header, which was designed to map binary data into ASCII characters. It is not related to HTTP's Content-Encoding header, which is used to encode (e.g. compress or encrypt) the content of HTTP request and responses.

As "base64" is defined in both RFCs, the definition from RFC 4648 SHOULD be assumed unless the string is specifically intended for use in a MIME context. Note that all of these encodings result in strings consisting only of 7-bit ASCII characters. Therefore, this keyword has no meaning for strings containing characters outside of that range.

If this keyword is absent, but "contentMediaType" is present, this indicates that the encoding is the identity encoding, meaning that no transformation was needed in order to represent the content in a UTF-8 string.

The value of this property MUST be a string.

9.4. "contentMediaType"

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

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

9.5. "contentSchema"

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

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

The value of this property MUST be a valid JSON schema. It SHOULD be ignored if "contentMediaType" is not present.

9.6. Example

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

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

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

Another example:

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

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

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

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

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

10. A Vocabulary for Basic Meta-Data Annotations

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

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

The current URI for this vocabulary, known as the Meta-Data vocabulary, is:

<https://json-schema.org/draft/2020-12/vocab/meta-data>.

The current URI for the corresponding meta-schema is:

https://json-schema.org/draft/2020-12/meta/meta-data (https://json-schema.org/draft/2020-12/meta/meta-data).

10.1. "title" and "description"

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

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

10.2. "default"

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

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

10.3. "deprecated"

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

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

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

The "deprecated" keyword applies to each instance location to which the schema object containing the keyword successfully applies. This can result in scenarios where every array item or object property is deprecated even though the containing array or object is not.

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

10.4. "readOnly" and "writeOnly"

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

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

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

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

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

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

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

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

10.4.1. "readOnly" and "writeOnly" example

In the following example of a read/write API accepting and producing JSON representations of resources, "username" is a field meant for display and cannot be changed, while "password" cannot be retrieved for display but can be set to a new value.

```
json { "$id": "https://example.com/schema", "type": "object",
"properties": { "username": { "type": "string", "readOnly": true },
"password": { "type": "string", "writeOnly": true } } }
```

With the instance following instance processed:

```
json { "username": "xyz", "password": "123" }
```

two annotation output units would be produced:

```
json { "keywordLocation": "/properties/username/readOnly",
"absoluteKeywordLocation":
"https://example.com/schema#/properties/username/readOnly",
"instanceLocation": "/username", "annotation": true }
```

```
json { "keywordLocation": "/properties/password/writeOnly",
"absoluteKeywordLocation":
"https://example.com/schema#/properties/password/writeOnly",
"instanceLocation": "/password", "annotation": true }
```

These annotations are used for context-dependent validation, which is performed by the application that invoked schema evaluation.

The API’s behavior when sent a readonly field in a write request or a write-only field in a read request is out of scope of this document. This illustrates how the API server can use the schema to trigger whatever that behavior is, rather than hard-code read-only and write-only flags or find a custom data-oriented solution.

10.5. "examples"

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

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

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

11. Loading and Processing Schemas

11.1. Loading a Schema

11.1.1. Initial Base URI

[RFC3986], Section 5.1 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 [RFC3986], Section 5.

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 [RFC3986], Section 5.1.4. 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 an earlier section is present in the root schema, this base URI SHOULD be considered the canonical URI of the schema document’s root schema resource.

11.1.2. Loading a referenced schema

Although it's impossible to cover all use cases, we start by assuming that an implementation given a schema with references to other schemas not in the same document can be given instructions about those other documents, and the implementation therefore SHOULD NOT automatically dereference network locations or search the network for schemas not already loaded in.

What should implementations do when the referenced schema is not known? The implementation could have error messages, flags or UX to explicitly get instructions to fetch a schema or to signal that it should be configured differently. The examples from HTTP of same-origin policies would seem relevant here too, but such a feature has not yet been defined for JSON Schema.

Some use cases may involve schema definitions that regularly are extended or updated by reference. For example, a service hosting an evolving API might include documentation and requirements via JSON schemas, and the schemas might be intended for dynamic fetching and inclusion of sub-schemas, so placing an absolute requirement of pre-loading schema documents is not feasible.

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 11.5).

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 implementation 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, an implementation SHOULD raise an error condition.

11.1.3. 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.

11.2. 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 [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" }
        }
    }
}
```

In this example, 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.

11.2.1. Relative References

Many hypermedia contexts (like HTML) make use of full URIs, anchors/names, and _relative references_ ([RFC3986], Section 5.1). In JSON Schema, different contexts and use cases may make any of these three approaches the most convenient and least brittle; but relative references do require the most care in implementations.

A fully conformant implementation MUST handle relative references, with the following guidance hopefully keeping implementation logic and overhead to a reasonable level. A schema’s $id acts as a base URI (see [RFC3986], Section 5.1.1) for relative references within the schema.

In accordance with [RFC3986], Section 5.1.2 regarding encapsulating entities, if an "$id" in a subschema is a relative 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 Section 11.1.1.

11.2.2. JSON Pointer fragments and embedded schema resources

JSON Pointer URI fragments are constructed based on the structure of the schema document, allowing an embedded schema resource and its subschemas to be identified by JSON Pointer fragments relative to either its own canonical URI, or relative to any 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 4.2.1), 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" points to the "items" schema, which is an embedded resource. The canonical URI of that schema resource, however, is "https://example.com/bar".

For the "additionalProperties" schema within that embedded resource, the URI "https://example.com/foo#/items/additionalProperties" points to the correct object, but that object's URI relative to its resource's canonical URI 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 "https://example.com/bar#/additionalProperties", using a JSON Pointer fragment appended to the canonical URI of the "bar" schema resource, is still valid, while "https://example.com/foo#/items/additionalProperties", which relied on a JSON Pointer fragment appended to the canonical URI of the "foo" schema resource, no longer resolves to anything.

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 this URI is valid, it is more robust to use the "$id" of the embedded or referenced resource unless it is specifically desired to identify the object containing the "$ref" in the second (non-embedded) arrangement.

An implementation MAY choose not to support addressing schema resource contents by URIs using a base other than the resource's canonical URI, plus a JSON Pointer fragment relative to that base. Therefore, schema authors SHOULD NOT rely on such URIs, as using them may reduce interoperability.
// 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. After some discussion, we
// feel that we need to remove the use of "canonical" in favour of
// talking about JSON Pointers which reference across schema resource
// boundaries as undefined or even forbidden behavior
// (https://github.com/json-schema-org/json-schema-spec/issues/937,
// https://github.com/json-schema-org/json-schema-spec/issues/1183)

Further examples of such non-canonical URI construction, as well as the appropriate canonical URI-based fragments to use instead, are provided in Appendix A.

11.3. 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.

11.3.1. 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.

11.3.2. 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.

11.3.3. 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.
// 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.

11.4. Caveats

11.4.1. Guarding Against Infinite Recursion

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

11.4.2. 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 "$defs" (Section 4.2.2) that take one or more subschemas as a value. These keywords may be "$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.
// 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

11.5. RESTful / Hypermedia Schema References

JSON and JSON schemas are not always used for HTTP resources or other hypermedia resources, and the rest of this document assumes no one protocol, nor does it even assume network access. However since HTTP resources in JSON with JSON Schemas to describe them are pretty common in Web APIs, this section describes how to process JSON documents in a more RESTful manner when using protocols that support media types and Web linking ([RFC8288]).

11.5.1. 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, ([LDP] Section 8.1.

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"
```

11.5.2. Usage Over HTTP

When used for hypermedia systems over a network, HTTP ([RFC9110]) 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.

12. 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 input 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 input against a schema involves processing all of the keywords in the schema against the appropriate locations within the input. Typically, applicator keywords are processed until a schema object with no applicators (and therefore no subschemas) is reached. The appropriate location in the input is evaluated against the assertion and annotation keywords in the schema object. The interactions of those keyword results to produce the schema object results are governed by Section 12.8.1.2, while the relationship of subschema results to the results of the applicator keyword that applied them is described by Section 12.6.

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.

12.1. 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 input 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.

12.2. Keyword Interactions

Keyword behavior MAY be defined in terms of the annotation results of subschemas (Section 3.1.2) 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 3.1).

12.3. 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.

12.4. Handling unrecognized or unsupported keywords

Implementations SHOULD treat keywords they do not recognize, or that they recognize but do not support, as annotations, where the value of the keyword is the value of the annotation. Whether an implementation collects these annotations or not, they MUST otherwise ignore the keywords.

12.5. Identifiers

Identifiers define URIs for a schema, or affect how such URIs are resolved in schema references (Section 4.2.1), 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 input, 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.

12.6. Applicators

Applicators allow for building more complex schemas than can be accomplished with a single schema object. Evaluation of an input against a schema document (Section 3.1) begins by applying the root schema (Section 3.1.2) to the complete input 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 12.7) 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 12.8) results from subschemas are preserved in accordance with Section 12.8.1 so that applications can decide how to interpret multiple values. Applicator keywords do not play a direct role in this preservation.

Annotation 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.

12.6.1. Referenced and Referencing Schemas

As noted in Section 12.6, 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 input against a schema.

For some by-reference applicators, such as "$ref" (Section 4.2.1), the referenced schema can be determined by static analysis of the schema document's lexical scope. Others, such as "$dynamicRef" (with "$dynamicAnchor"), may make use of dynamic scoping, and therefore only be resolvable in the process of evaluating an input with the schema.

12.7. 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 input against schema assertions.

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

12.7.1. Assertions and Input Primitive Types

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

For example, the "maxLength" keyword from the companion validation vocabulary ([I-D.bhutton-json-schema-validation]): will only restrict certain strings (that are too long) from being valid. If the input 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 inputs that can be of multiple primitive types. The companion validation vocabulary also includes a "type" keyword which can independently

restrict the input 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 input 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 input to strings, then including "null" in "type" would not have any useful effect.

12.8. 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 ([I-D.handrews-json-schema-hyperschema]) "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.

12.8.1. 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)

12.8.1.1. 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 ([I-D.bhutton-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 Fe
ature A",
             "$ref": "#/$defs/enabledToggle"
           }
         }
       }
     ],
     "$defs": {
       "enabledToggle": {
         "title": "Enabled",
         "description": "Whether the feature is enabled (true), disabled (false), or unde
r 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.

12.8.1.2. 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 input "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 input passes the string type assertions.

12.9. 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 11.4.2 unrecognized extension keywords that reserve locations for re-usable schemas may have undesirable interactions with references in certain circumstances.

12.10. Loading Input Data

While none of the vocabularies defined as part of this or the associated documents define a keyword which may target and/or load input 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 input 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.

13. 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.

13.1. Format

JSON Schema output is defined using the JSON Schema data model. 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.

13.2. 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.

13.3. Minimum Information

Beyond the simplistic "flag" output, additional information is useful to aid in debugging a schema or input. 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.

13.3.1. 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".

13.3.2. 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 resource with a JSON Pointer fragment, 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.
// 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".

13.3.3. Instance Location

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

The JSON key for this information is "instanceLocation".

13.3.4. 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".

13.3.5. 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.

13.4. 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 input 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 input 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.

13.4.1. 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.

13.4.2. 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"
    }
  ]
}
```

13.4.3. 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"
    }
  ]
}
```

13.4.4. 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 (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
}
```

input:

```
{
  "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."
        }
      ]
    }
  ]
}
```

13.4.5. 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 (https://json-schema.org/draft/2020-12/output/schema).

14. Extensibility

14.1. Non-JSON Inputs

It is possible to use JSON Schema with a superset of the JSON Schema data model, where an input 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.

14.2. 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.

A schema vocabulary can be defined by anything from an informal description to a standards proposal, depending on the audience and interoperability expectations. In particular, in order to facilitate vocabulary use within non-public organizations, a vocabulary specification need not be published outside of its scope of use.

14.3. 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.

Meta-schemas that use the "$vocabulary" keyword (Section 4.1.2) to declare the vocabularies in use MUST explicitly list the Core vocabulary, which MUST have a value of true indicating that it is required.

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.

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.

Meta-schemas and vocabularies together 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:

1. 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.

2. 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.

14.4. Default JSON Schema Dialect

The current URI for the default JSON Schema dialect meta-schema is https://json-schema.org/draft/2020-12/schema. For schema author convenience, this meta-schema describes a dialect consisting of all vocabularies defined in this specification, as well as two former keywords which are reserved for a transitional period. Individual vocabulary and vocabulary meta-schema URIs are given for each section below. Certain vocabularies are optional to support, which is explained in detail in the relevant sections.

Updated vocabulary and meta-schema URIs may be published between specification drafts in order to correct errors.

15. Security Considerations

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

Instances and schemas are both frequently written by untrusted third parties, to be deployed on public Internet servers. Implementations should take care that the parsing and validating against schemas does not consume excessive system resources. Implementations 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.

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

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

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

16. Interoperability Considerations

16.1. 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.

16.2. 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.

16.3. Regular Expressions

Keywords MAY use regular expressions to express constraints, or constrain the input value to be a regular expression. These regular expressions SHOULD be valid according to the regular expression dialect described in [ECMA262], Section 21.2.1.

Unless otherwise specified by a keyword, regular expressions MUST NOT be considered to be implicitly anchored at either end. All regular expression keywords in this specification and its companion documents are un-anchored.

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".

17. IANA Considerations

17.1. 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

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

* Security considerations: See Section 15 above.

* Interoperability considerations: See Section 16.1, Section 16.2, and Section 16.3 above.

* Fragment identifier considerations: See Section 3.4.

17.2. 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: N/A

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

* Security considerations: See Section 15 above.

* Interoperability considerations: See Section 16.1, Section 16.2, and Section 16.3 above.

* Fragment identifier considerations: See Section 3.4.

18. References

18.1. Normative References

[ECMA262] European Computer Manufacturers Association, "ECMAScript Language Specification 5.1 Edition", ECMA Standard ECMA-262, June 2011, <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>.

[I-D.hha-relative-json-pointer] Luff, G., Andrews, H., and B. Hutton, "Relative JSON Pointers", Work in Progress, Internet-Draft, draft-hha-relative-json-pointer-00, 19 June 2023, <https://datatracker.ietf.org/doc/html/draft-hha-relative-json-pointer-00>.

[LDP] Speicher, S., Arwe, J., and A. Malhotra, "Linked Data Platform 1.0", W3C Recommendation REC-ldp-20150226, 26 February 2015, <http://www.w3.org/TR/2015/REC-ldp-20150226/>.

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

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

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

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

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

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

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

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

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

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

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

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/rfc/rfc5321>.

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

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

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

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

[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/rfc/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/rfc/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/rfc/rfc8259>.

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

[XMLNS] Bray, T., Hollander, D., Layman, A., Tobin, R., and H. Thompson, "Namespaces in XML 1.0 (Third Edition)", W3C Recommendation REC-xml-names-20091208, 8 December 2009, <http://www.w3.org/TR/2009/REC-xml-names-20091208/>.

18.2. Informative References

[I-D.bhutton-json-schema-validation] Wright, A., Andrews, H., and B. Hutton, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", Work in Progress, Internet-Draft, draft-bhutton-json-schema-validation-01, 10 June 2022, <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-validation-01>.

[I-D.handrews-json-schema-hyperschema]
Andrews, H. and A. Wright, "JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON", Work in Progress, Internet-Draft, draft-handrews-json-schema-hyperschema-02, 17 September 2019, <https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-hyperschema-02>.

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

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

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[W3C.WD-fragid-best-practices-20121025]
Tennison, J., Ed., "Best Practices for Fragment Identifiers and Media Type Definitions", W3C WD WD-fragid-best-practices-20121025, W3C WD-fragid-best-practices-20121025, 25 October 2012, <https://www.w3.org/TR/2012/WD-fragid-best-practices-20121025/>.

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 Section 3.4 and Section 11.2.2 above.

* # (document root)
  - canonical (and base) URI: https://example.com/root.json
  - canonical resource URI plus pointer fragment: https://example.com/root.json#
* #/$defs/A
  - base URI: https://example.com/root.json
  - canonical resource URI plus plain fragment: https://example.com/root.json#foo
  - canonical resource URI plus pointer fragment: https://example.com/root.json#/$defs/A
* #/$defs/B
  - canonical (and base) URI: https://example.com/other.json
  - canonical resource URI plus pointer fragment: https://example.com/other.json#

    - base URI of enclosing (root.json) resource plus fragment:
      https://example.com/root.json#/$defs/B

* #/$defs/B/$defs/X

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

    - canonical resource URI plus plain fragment:
      https://example.com/other.json#bar

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

    - base URI of enclosing (root.json) resource plus fragment:
      https://example.com/root.json#/$defs/B/$defs/X

* #/$defs/B/$defs/Y

    - canonical (and base) URI: https://example.com/t/inner.json

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

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

    - base URI of enclosing (other.json) resource plus fragment:
      https://example.com/other.json#/$defs/Y

    - base URI of enclosing (root.json) resource plus fragment:
      https://example.com/root.json#/$defs/B/$defs/Y

* #/$defs/C

    - canonical (and base) URI: urn:uuid:ee564b8a-
      7a87-4125-8c96-e9f123d6766f

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

    - base URI of enclosing (root.json) resource plus fragment:
      https://example.com/root.json#/$defs/C

Note: The fragment part of the URI does not make it canonical or non-
canonical, rather, the base URI used (as part of the full URI with
any fragment) is what determines the canonical nature of the
resulting full URI.
// Multiple "canonical" URIs? We Acknowledge this is potentially

```
   // confusing, and direct you to read the CREF located in the JSON
   // Pointer fragments and embedded schema resources (Section 11.2.2)
   // section for futher comments.
```

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. "$ref") use URI-references that resolve to URIs using the canonical resource URI as the base, and all schema resources have an absolute-URI as the "$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 the 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 input 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
}
```

input 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 input, 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 input root, but applying "tree" to input 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 5.2.1) 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 ([I-D.handrews-json-schema-hyperschema]) 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 ([I-D.handrews-json-schema-hyperschema]), 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 input 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 the validation specification ([I-D.bhutton-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

This draft is based on great amounts of superb work by creators and contributors to JSON Schema. The authors of the 2020-12 spec include Ben Hutton and Greg Dennis. Thanks also to Jason Desrosiers.

Past contributors include, with thanks: Gary Court, Francis Galiegue, Kris Zyp, Geraint Luff Daniel Perrett, Erik Wilde, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave Finlay, Denis Laxalde, Phil Sturgeon, Shawn Silverman, and Karen Etheridge.

Appendix G. Change Log

// This section to be removed before leaving Internet-Draft status.

G.1. draft-dusseault-json-schema-00

Compared to the "2020-12" version of JSON Schema, this draft makes the following changes.

* Consolidate the 2020-12 core and validation documents
* Delevel headers for a more readable Table of Contents
* Shift terminology to a terminology section
* Reorder conceptually: intro, keywords, processing and output, extensibility.
* Define input and instance as different things.

Authors' Addresses

Lisa Dusseault
Data Transfer Initiative
Email: lisa@rtfm.com

Austin Wright
Email: aaa@bzfx.net

Henry H. Andrews
Email: andrews_henry@yahoo.com