JSON Schema Extensions

Austin Wright 2022-10-24

This paper describes an update to JSON Schema semantics to permit generic extensions to the media type, in a way that's interoperable between multiple implementations.

Part one will describe the problems being solved, and the functional requirements that solve them, including forward compatibility and reverse compatibility.

Part two will describe interoperability requirements that provide these guarantees.

Part three suggests how this affects releases and changes that come after.

Part four describes *standard extensions*: common features implemented in a standard way, but that aren't necessary to guarantee interoperability.

Part five describes extensions that implement obsolete or deprecated behaviors, not recommended for use in new schemas, but that maximize backwards compatibility with older schemas.

README

- This is a draft, while there is one core concept (the "indeterminate" state), this also describes many other ideas that might or might not be any good.
- Most of the concepts here are not significant changes in behavior, however there are stricter definitions. Some terms have been recycled to subtly different purposes (especially "assertion" and "argument"). "sibling arguments" "keyword arguments" and "argument keywords" are supposed to be different things but even I might be confusing them in one or two places. Ask if in doubt.
- This shouldn't be read as an endorsement of which features are "more important" or "less important" to JSON Schema. It merely describes requirements that must be in place before the interoperability of other features (e.g. annotations, $schema) is guaranteed. (e.g. HTTP defines cookies in a non-core specification, that doesn't mean they're unimportant, they're just not essential to understanding the different parts of an HTTP message.)
- Vocabulary note: "accept" and "reject" are outcomes, "validate" is a process that may accept or reject. "Author" is someone writing a schema.

1. Functional Requirements

The current JSON Schema specification is difficult to evolve, and while the $schema and $vocabulary keywords have found some use in mitigating these problems, forward and reverse compatibility is still a significant burden.

Implementations are burdened by backwards compatibility

JSON Schema does not discuss backwards compatibility: behavior specified in obsolete documents that no longer appears, but is still used in the wild. Support for these behaviors is left up to individual implementations, because different implementations are built to satisfy different needs.

One popular strategy is to implement each publication separately, and use some heuristic to select one of them—often reading the $schema keyword, analyzing the meta-schema (if available), or a "version" provided out-of-band. This is also not guaranteed because "$schema" is optional, or if provided, it doesn't have to be a known meta-schema, and even if the unknown meta-schema is machine readable, the version that the meta-schema was written against may be different than the schema!

This approach is quite burdensome for implementors, and concerns arise around the question of "which meta-schemas should the validator support"—and what that even means. How should validators handle schemas that are "too old?"

Multi-consumer schemas need strong interoperability guarantees

The very assumption that validators can implement their own backwards compatibility scheme fails when one schema is intended for use by a wide variety of validators (*multi-consumer schemas*). For example, a description of an API written using JSON Schema. Schema authors may encounter two classes of problems:

First, implementations may disagree on the validity of an instance due to specification changes over time. If Alice's validator ignores the "comprehensibility" keyword (because it is too new or too old), but Bob's supports it, schemas with this keyword will show contradictory validation results (Alice and Bob will think the same document is valid and invalid, respectively).

Second, compatibility between revisions of the specification must overlap in a uniform and predictable way. For example, if Alice's validator only supports 2021 meta-schemas and later (because it's impractical to support older meta-schemas), and Bob's validator only supports 2020 and earlier (because it is now unmaintained), then there is no way to publish a schema that works for both of them.

Behavior was removed when it was considered bad practice. However, the purpose of a specification is to promote an interoperable ecosystem, not necessarily to describe the ideal. A standard must describe backwards compatibility, too.

Robustness Principle: Be Liberal In What You Accept

Suppose the year is 2040, JSON Schema adds a "comprehensibility" keyword. But you're still using a validator from 2036, so it ignores this keyword. Your validator says your jetcar registration form is valid, but the flying vehicle office says it isn't. What's going on here?

One of the most prevalent problems in authoring the specification is that it isn't possible to change the features of the specification without changing some validation results. This is due to an early design decision that "unknown" keywords are "ignored" much the same way as HTTP and mail ignores unknown headers.

HTTP and mail can do this because the base message carries useful semantics, and clients can decide if they will act on the additional headers provided, if any. When the header *is* understood, you can see from the response that it was processed.

In HTTP, this is usually signaled in the status code. Depending on the method, media type, and requested features the server understands, it will send back different status codes. If the server doesn't understand the method, it will send 501 Not Implemented; and if it honors a Range request, it will send 206 (Partial Content). The "206" status lets the client know that the "Range" extension is in use!

In contrast, when JSON Schema adds a new keyword, validators do not signal if the new keyword is being honored. Honored keywords in valid instances are indistinguishable from ignored keywords.

The "robustness principle" doesn't imply that unknown fields be ignored; it endorses *graceful degradation*, which JSON schema does not do as well as HTTP or mail.

Validators may have incomplete understandings, and different validators different understandings

When authoring a schema, assume that it has a single, definite meaning; even if not every part is known to a given validator. *In the context of a specific validator*, a schema whose behavior is not completely known, may be called *incomplete*.

Forward Compatibility Principle

For lack of a better term, this means that the widest possible range of values should be reserved for future use, and there should be a disincentive to using reserved values until they're defined. Because once some behavior is defined and starts being used, it cannot be un-defined.

Thus, if in doubt, unused values should be reserved and handled as a special case, rather than being ignored.

For example, there's no reason an argument keyword should be permitted by itself without a validation keyword that reads it. This likely indicates a typo. If a good reason is found, it can be permitted later. But if it starts out permitted, it is more difficult to prohibit after the fact.

If a behavior is being reserved for future use, this is a type of *undefined* behavior called *reserved*. Nobody should write schemas using reserved behavior until it's been defined at a

later time, and if encountered, validators must treat the schema as if it's incomplete (the behavior may have since been defined).

Infrequently, there may be multiple ways an implementation may implement a behavior. This is a different type of *undefined* (undefined by the specification) called *implementation-defined*. If a specification calls for implementation-defined behavior, this is often a sign that the scope of the specification is too broad.

Strict handling of keywords and arguments alike

Currently, keywords with an unknown name are ignored, but known keywords with an invalid value usually error. This is an odd combination not found in most protocols: it is more typical to ignore all unknown values, or error on all unknown values.

In JSON Schema, validation keywords are closer to method calls in a programming language: ignoring a keyword as if it doesn't exist is almost never intended by the author.

This also applies to the property value of the keyword (the keyword argument); it would make little sense to ignore a keyword if its argument goes outside the defined range.

If this is the rule that applies to keyword arguments, then it should also apply to sibling arguments. If a schema adds a newly defined sibling argument, it is likely there for an important reason, it should not be ignored just because it's too new.

Typos

Unlike in HTTP, mail, or even many calls to a JSON API, JSON Schema documents are often authored by hand. A typo in the keyword name can be dangerous, effectively disabling it without warning.

Most (respectable) programming languages have some amount of redundancy builtin to their syntax, which will cause an error if there is a typo. For example, referencing a variable that has not been initialized.

The "$defs" and "$comment" keywords have removed most of the incentives for schema authors to invent their own properties, so the only remaining reasons that an unknown property might be encountered is (1) it is a typo, (2) it is an new keyword the schema depends on, or (3) it's an older schema and $defs/$comment wasn't standardized yet.

If there is some reason authors need to invent a keyword, there should be some way to declare this in the schema.

The "$schema" keyword

Originally, the "$schema" keyword allows a schema to specify a "meta-schema" that it should be valid against. At minimum it was a curiosity that shows JSON Schema is self-descriptive. Combined with annotations, it could be used to give semantics to custom keywords in a schema, although no mechanism for this was defined initially.

Later, the $schema keyword had come to be used by implementations to selectively enable or disable features only found in obsolete drafts, or to switch between entire implementations.

Eventually the $vocabulary keyword could be embedded in meta-schemas, making them machine-readable.

This keyword has several weaknesses:

- JSON Schema is not “$schema-aware” so it’s not possible to write a schema that changes meta-schemas, that is also valid against those meta-schemas.
- It is not possible to write a schema that combines the characteristics of multiple meta-schemas, the same way that a JSON document can be described my multiple profiles.
- Meta-schemas seem to be useful to determine if the schema that you’re authoring will be compatible with a validator. But if a meta-schema is supposed to accept schemas that the validator supports, this means that different validators will have different meta-schemas, each unique to the features that they support, and can be no “standard" meta-schema.
- If a meta-schema describes a schema *to* a validator, this limits the usability of the schema to the validators that understand the semantics of, or that can read, that meta-schema.

The “$vocabulary” keyword

While the $schema keyword allows a schema to express some information about how the schema itself is written, the keyword was not defined thoroughly enough to be useful in validation: The normative interpretation of a schema is the specification, not the meta-schema.

To answer this, the $vocabulary keyword allowed meta-schemas to be machine readable. That is, if a validator doesn’t have knowledge of how to handle a meta-schema, it can read the meta-schema for instructions. This provides the useful feature of being able to write extensions for JSON Schema that are not standardized, and some ability for validators to differentiate among different versions of non-core keywords.

This keyword has several weaknesses:

One, it cannot not cross $ref or sub-schema boundaries (as far as I can tell). You cannot import vocabularies by referencing another meta-schema in your meta-schema (e.g. `{"$ref": "draft-07/schema"}` is not meaningful), nor can you use it inside allOf. However, this limitation is easily relaxed.

Two, it is not clear that $vocabulary applies to core keywords. A single “core” vocabulary is assumed at runtime to “bootstrap” the core keywords including $schema and $vocabulary. It’s possible different or better defined handling could be defined, for vocabularies to override the behavior of core keywords, or define “super-core keywords” that configure the behavior of other core keywords. But it seems this would require much additional complexity that should be avoided if possible.

Three, it requires authors to write a meta-schema for schemas whenever an extension is needed. And JSON Schema semantics preclude embedding meta-schemas inside schemas. So using custom extensions require loading at least two separate schema documents into your validator (the schema and its meta-schema).

Reusability across dialects and versions

Some software applications implement different subsets of JSON Schema, or define their own custom extensions. For example, not implementing $ref; or defining a new data type that's separate from the six JSON types.

Despite these differences, these environments still have much in common, and schema like `{"type": "number"}` should be re-usable and behave the same, regardless of if the schema is "for" MongoDB or OpenAPI. It should still be possible to write a schema that works in both environments, if you're only using features common to both.

Self-describing vs. Meta-describing

To invent a couple terms, a data format is *self-describing* when the format encodes information about how to read it *into* the format. It is *meta-describing* when you must know information out-of-band in order to use it.

For example in JSON, it is possible to distinguish numbers from strings: the JSON types are self-describing. But without knowing the application, you don't know what number is for what purpose: the rest of the semantics are described by the application, separate from JSON.

In contrast, in ASN.1, you need a schema to read the message and distinguish numbers from strings. The type system is meta-described.

With JSON Schema, you should be able to distinguish validation keywords and arguments from other keywords, without needing a (meta-)schema to do so.

Conclusion

1. It should be possible to identify keywords that affect validation, and ignore keywords unrelated to validation, by following the media type.

2. For validation purposes, an unknown keyword and unknown/invalid arguments are the same class of exception: the schema author is asking for validation behavior not known to the validator.

3. Features should deployed on a granular level. If two schemas need the same behavior ("test the instance is a string"), the "same" keyword should express this.

2. Required Implementation Behavior

A schema should be just self-descriptive enough that a validator "knows what it doesn't know." It should be possible to distinguish keywords that perform validation from those that don't, and to distinguish validation keywords it knows from those it doesn't. Keywords that may impact validation, but whose exact behavior is not known to the validator, must not return valid or invalid; for graceful degradation, these must cause an *indeterminate* result.

Keywords may be defined by a variety of sources—in the core functionality, the semantics of the $schema keyword, a registry, or out-of-band. In the case of the $schema keyword, as a "control keyword", it can potentially "redefine" other keywords with newer or different meanings.

However to start, let's define the base case behavior, starting with the semantics of minimal schemas.

Reading the Schema

> *This section restates how schemas already work, but in slightly more formal terms; and specifies that unknown properties are different than validation keywords.*

Fundamentally, a schema describes a *valid set* and an *invalid set* of JSON instances. A schema may describe other data, too (providing annotations for an instance, or schema metadata).

These sets are enumerated by assertions: functions whose input is a JSON document, and whose output is *accepting/rejecting* (or *valid/invalid*). The valid set and invalid set are defined as the inverse of the assertion function: the "valid set" consists of all instances that the assertion accepts; and the "invalid set" consists of all instances that the assertion rejects.

Assertions have the same function signature as state machines, and the same operators (union, intersection, difference, symmetric difference, concatenation, and brzozowski derivative all apply). However, they may be arbitrarily computationally expensive[1].

Decoding the assertion from the schema begins by parsing the JSON.

- If the schema is the boolean "`true`", this represents a single assertion that accepts; i.e. the set of all JSON documents.
- If the schema is the boolean "`false`", this represents a single assertion that rejects; i.e. the empty set.
- If the schema is an object, this represents the intersection of any number of assertions, each encoded as a keyword. (The intersection of zero assertions is permitted, which is the complete set of JSON documents.)
- The behavior of any other type (null, string, number, array) is reserved.

---

[1] Assertions may even be turing-complete, though the handling of an assertion that never returns is outside the scope of this document.

Keywords are notated using properties in an object. A keyword consists of one property whose key is the *keyword name*, and whose value is the *keyword argument*, a value within a domain of values expected for the keyword; and any number of other properties in the same object, the *sibling arguments*, whose names and domains are associated with the keyword. These arguments completely define the validation behavior of the assertion; they are bound to the keyword (i.e. curried) to produce the assertion.

Arguments may have a domain. For example, the “minimum” keyword is only defined over numbers. Therefore, `{"minimum": "foo"}` will not be recognized as the “minimum” keyword. (It may be technically possible to define two keywords with the same name, if the domain of their arguments do not overlap. But this is practically the same as one keyword that accepts both domains.)

In order to separate “assertions-producing” properties from other properties, the validator iterates over each of the properties in the object and categorizes them into one of four categories:

1. **Control keywords** are properties that potentially effect how the rest of the schema is read. They may indicate behavior that's defined in a new version of JSON Schema, change features available, or how other properties are categorized. “$” keywords often fall into this category.

2. **Validation keywords** are properties that encode an assertion; they exclude JSON documents from the valid set, depending on their arguments (the keyword argument or sibling arguments). Sibling arguments may be validation keywords, or argument keywords.

3. **Argument keywords** are properties that are read and used by a sibling validation keyword, but do not validate instances by themselves. They are defined only in the presence of a validation keyword that reads them, which means it is invalid to list an argument keyword by itself. It is also invalid for two validation keywords to read an argument keyword for different purposes. They are read from the same object (a sibling property) and do not cross schema/object boundaries.

4. **Non-validation keywords** are properties that do not directly play a part in validating instances, although they may hold schemas used indirectly for validation, or produce annotation results that affect the end application. These includes the annotation-only keywords.

Note how some validation keywords might also be arguments to another keyword; these properties are “arguments” and “sibling arguments”—but with respect to these categories, they are not “argument keywords.” For example, “patternProperties” is a validation keyword, and also an argument to “additionalProperties”.

Schemas with unknown properties (properties not known to fall into any category) are called *incomplete*. Since incomplete schemas have some amount of unknown behavior, they cannot be used to compute closed sets of JSON documents. Validators attempting to use an incomplete schema may emit an error.

The open set of valid instances

> *This adds the concept of “open sets” and “closed sets” of JSON documents, which are differently useful in different situations.*

Normally, assertions have a domain of all JSON instances. But in a validator, assertions may only be defined over a *subset* of JSON instances, including none at all. Assertions that are not defined over all JSON instances are called *incomplete*. This usually happens because a subschema in an argument is itself incomplete.

When a schema is incomplete (it contains an unknown keyword), then “closed” (exhaustive) sets of instances cannot be computed. In category theory terms: an assertion is incomplete when the image of its inverse is a strict subset of the set of JSON documents.

However, sophisticated validators may still describe an *open set* of instances: a set of known valid instances, but not necessarily all of the instances that the schema intended. This allows a sophisticated validator to return valid, invalid, or *indeterminate*, depending on the instance.

For example, given the schema:

```
{ type: “object”, properties: { “foo”: {comprehensibility: 10} } }
```

So long as an object does not contain the “foo” property, a sophisticated validator could determine it is valid. However objects that *do* contain “foo” would be outside the domain of the assertion, so attempts to validate these instances would be indeterminate: they might be valid or invalid, pending the definition of the “comprehensibility” keyword.

Validators are not required to support open sets; if the schema uses keywords outside of the required keywords (see below), they may simply error.

Strict/non-strict modes

The handling of valid instances differently from incomplete schemas may be referred to as "strict handling." Most authors will immediately benefit from strict handling: there’s not a need to write schemas with unknown properties, these are usually typos.

However strict handling may break a small number of 3rd party schemas that invented property names for various reasons (comments, or making a place for named subschemas). The introduction of a feature flag can keep reverse compatibility these cases:

- In non-strict mode, any properties that are unknown are reclassified as argument keywords. (Or non-validation keywords.)

The following rollout is recommended for validator packages:

1. A new minor (feature) release of the package should add a “test strict” function that tests if the schema is “complete” and how properties in the schemas were categorized; and an option to enable strict handling. Applications can then opt-into strict mode and handle invalid schemas appropriately.

2. The next major (breaking) release of the package enables strict mode by default.

"Non-strict" schemas are easily fixed by renaming invented keywords to use the standard keyword name for their purpose. This change keeps compatibility with the original validator it was written for, too.

Schemas that require strict validators may use the errorAssert keyword (see below).

Required Keywords

Certain keywords MUST be implemented to claim "JSON Schema" compatibility, particularly enough to fully support "structural validation."

Authors can rely on these keywords to do all the same things they could do with ABNF, and be confident that it will accept/reject without error. This is an important guarantee for standards specifications wishing to describe JSON using JSON Schema.

(For software packages that use JSON Schema features but don't need these keywords, because of cost or lack of need, we can invent a generic name, e.g. JSON meta-framework.)

Within the category of validation keywords, there's a few subclasses: context-free "structural" validation keywords, and error handling "greasing" keywords.

Required Control Keywords

Control keywords are used to configure certain aspects of schema authoring (especially URI references), or declare/import extensions to the required JSON Schema behavior.

Control keywords can potentially affect each other.

$id

This keyword changes the base URI within the rest of the schema, that URI References are resolved against; and it provides a "self" link relation that other schemas (even subschemas in the same document) may use.

If it is itself a URI Reference, this is resolved against the schema/document base URI (as specified in RFC3986: the base URI of the parent schema, if any, or the URI that the document was retrieved from, if any; or else an application-specific URI).

And it is only defined when the resolved URI is "absolute" i.e. "fragmentless" (as is required in an HTTP request).

$anchor

This keyword names a schema so that it may be referenced in a fragment identifier. It is similar to the `id=` attribute in HTML.

Required Structural Validation Keywords

The required keywords include all the structural validation keywords: all the keywords necessary to describe any context-free grammar that's valid JSON and where two semantically identical documents are not distinguished (e.g. whitespace differences, or escaped characters in strings).

> *By defining "structural validation" this way, we can objectively verify that the builtin set of core functionality is comprehensive and is unlikely to need changes in order to accommodate some unforeseen use case.*

By limiting structural validation to context-free grammars, validators can evaluate instances in O(m) time, by using a non-deterministic pushdown automaton, or even a regular expression.

- type, allOf, anyOf, oneOf, someOf, const, enum, assert
- minimum, exclusiveMinimum, maximum, exclusiveMaximum, multipleOf
- pattern, minLength, maxLength
- items, prefixItems, concat, repeatMin, repeatMax
- properties, patternProperties, additionalProperties, required
- $ref

Notes for some of these keywords:

$ref

Only defined for values that resolve to URIs known to the validator.

assert

Argument is a subschema. Accepts iff the subschema accepts the instance. This is a terse alternative to `{ allOf: [ subschema ] }`.

concat

Argument is an array of subschemas. Accepts arrays iff there is any way to split the instance into segments (one per subschema), such that subschema accepts the corresponding segment. It is required to describe some types of regular array patterns.

someOf

Argument is an array of subschemas. Accepts when *one or more* of its subschemas accepts the instance. It is required to describe some types of regular array patterns.

repeatMin, repeatMax

Required for describing repeating patterns of items within a single array.

Required Greasing Validation Keywords

> *This introduces new keywords not used for any validation purpose, but that verify implementations are compliant with the spec. They must exist in all compliant implementations, otherwise they would lose their usefulness.*

Three “validation” keywords are defined that validate the sub-schema against the instance, but may discard the validation result, and instead act on the error result.

Note that these keywords could cause a validator to legitimately accept or reject an instance, depending on the functionality it supports. This is acceptable in this case because that's the author's *intention*.

errorTry

The "errorTry" keyword applies the subschema to the instance, except if there is an indeterminate result, it instead uses the result of the subschema in "errorCatch" (or a valid result, if this is not defined).

This keyword can force all of the unknown properties within the subschema to be ignored:

```
{errorTry: subschema}
```

> *There may need to be two variations of this keyword; one that handles an indeterminate result, and another that runs the assertion in "non-strict" mode. These may be slightly different if a validator doesn't attempt to validate incomplete schemas. (See "open set" above.)*

errorCatch

This indicates a schema that will be validated only if the subschema in the "errorTry" keyword cannot be evaluated. Its default value is empty: if no errorCatch is provided, then errorTry catches all errors.

This keyword may be used of force errors to instead return invalid:

```
{errorTry: subschema, errorCatch: false}
```

errorAssert

This keyword "asserts" that a schema is incomplete. It accepts any value, then attempts to read the argument as a sub-schema, and rejects if the sub-schema is complete (a valid schema).

In conjunction with the "not" keyword, it can be used to test that valid results only come from validators that implement strict keyword handling:

```
{ not: { errorAssert: { errorThrow: true } } }
```

Placing this in a schema is an effective way of preventing obsolete JSON Schema validators from saying that an invalid instance is actually valid. (Which is typically preferable to a validator accepting what's actually an invalid instance.)

errorThrow

This keyword is undefined but reserved, for the purpose of testing what implementations do when encountering an unknown keyword.

Identifying Annotation Keywords

To maximize forward compatibility, there has to be a way to indicate that a keyword has no effect on validation, so that new annotation keywords that the validator doesn't understand doesn't interfere in its validation result.

The simplest way to do this is to introduce a naming scheme, much like the `$` prefix does.

So, annotation keywords are any keywords known to be annotation keywords, or any keyword prefixed with an "@". You can think of it as meaning "annotation," or like the @ error suppression operator in PHP.

When any keyword is prefixed with an "@", this suppresses the assertion (and the validation result). Annotation-only keywords should use an "@" so that validators that don't implement the keyword will not error.

Usage with validation keywords

It's occasionally handy with validation keywords too. For example, "@oneOf":

```
{ "@oneOf": [
      { "required": ["label"], "@description": "Human-readable
title" },
      { "required": ["tag"], "@description": "Machine-readable
title" },
] }
```

This schema has the effect of producing one description or another, or potentially no description at all, in which case the oneOf will still accept.

"format"

Likewise, "@format" would unambiguously be an annotation keyword; the "format" keyword would make sense as a validation keyword that errors if the argument is not understood.

```
{ "format": "http://example.com/postal-tracking-no" }
// indeterminate: error: unknown format

{ "@format": "http://example.com/postal-tracking-no" }
// valid
```

(If making "format" a validation keyword again is too much to bear, consider introducing "stringFormat" and "numberFormat" keywords that only apply to strings and numbers, respectively.)

Note "format" makes sense as an extension, not a core/required keyword (see "Formats" below for rationale).

Other Non-Validation Keywords

This subcategory includes keywords that don't perform validation and don't produce annotations. They usually provide a place to store some value or schema when there's no other good place to.

$defs, definitions

A key-value map of schemas. They are not applied to the instance by this keyword, however since their values are understood as a schema, they may be referenced elsewhere in the document from $ref.

$comment

An arbitrary value, for discussing how the schema is written (rather than the meaning of instances)

Meta-schemas

Validators may describe the range of behavior that they implement in the form of a meta-schema. The valid set of a meta-schema represents the schemas that a specific validator is able to produce closed sets for.

This set of schemas is not necessarily closed; if the validator supports processing open sets, it may be possible to write schemas that use unknown keywords that can still be processed by the validator.

For example, this schema accepts all instances, but are not likely found in any validator's set of valid schemas:

```
{ "$defs":
   "foo": {$anchor: "foo", comprehensibility: 10}
}
```

It is possible for an application to statically analyze this schema and determine that it is complete, but no combination of currently known keywords can do this. The validator would have to walk the keywords from the root, and look for $ref keywords, then dereference and recursively walk them.

3. Evolving the JSON Schema Standards

This considers how updates to JSON Schema might be published, that don't impose requirements on validators.

Core Semantics

The core semantics would be published in a single media type specification. It contains the bare minimum rules that must be implemented for validators to not produce inaccurate results, regardless of future revisions or extensions.

- The core mechanic: A schema describes a closed set of JSON documents
- The ability to use a JSON Schema to produce an open set of JSON documents
- The definition of validation, assertions, and keywords
- Detecting extensions and experiments
- (Maybe) The definition of annotations, and their relationship to resources & profiles
- Minimum required validation functionality
- The media type definition, including fragment identifiers
- A standard keyword/extension registry

Revisions to Core Semantics

Revisions to the core semantics document should be infrequent, usually just to update best practices. Small changes that don't impact validators or interoperability may be held as errata.

Behavior/semantics changes are possible, but reverse compatibility must be supported. Updates would completely replace the old document, and would have to be backwards compatible with it.

Standard Extensions

Standard extensions are features that are used in multiple different environments, and so should be written in a standard way. The extensions in section 4. Standard Extensions in particular will be so ubiquitous that they should be considered essential, only excepting one or two niche cases.

There would be a registry of keywords, which lists the standard keywords and points to the document that defines them.

Revisions to standard extensions

Published extensions may be revised to add keywords, or expand the range of legal arguments to a keyword. These types of changes will not break schemas in the wild.

Changing the behavior of already-published keywords may be done in two situations, with great care: either the behavior not used in the wild, or the official behavior is so undesirable or inconsistent, that it's just easier to change the standard definition to match what's seen in the wild.

A control keyword could technically work as a way to change how an existing keyword behaves. However, it is usually simpler and more straightforward to copy & rename the keyword, rather than introduce a control keyword.

User Extensions

User extensions are extensions to a schema that are not standardized. It may be a one-off experiment by a single party, or a use so niche that there's no benefit to standardization.

These keywords are written as a URI:

```
{
   "http://example.com/kw/comprehensibility": 10
}
```

Despite the naming scheme, they are handled the same way by validators. Validators do not make a distinction between standard extensions and user extensions.

Todo: there should be a way to declare two keywords have the same behavior, in case a user keyword is standardized with a different name.

Test Suites

Tests should be developed and published per specification document, even historical ones. This allows tracking of how features evolve over publications, and testing compatibility with historical publications together at the same time.

4. Standard Extensions

This section lists a number of RECOMMENDED extensions that provide useful features in a standard fashion, but that are not always applicable to all environments. (Usually this means environments with constrained memory usage, applications requiring performance guarantees, or handling schemas written by untrusted parties.)

$schema

The $schema keyword provides a way to to import a common "dialect" of keywords. The argument is a URI specifying how to (re)define the semantics of all non "`$`" keywords, even for keywords that otherwise would have a different definition. (And in very special cases, even some "`$`" keywords.)

> *Whenever meta-schemas are used to validate schemas, they merely describe the behavior that a* specific *validator understands. Asserting that a schema conforms to a meta-schema will necessarily make that schema less interoperable!*

The rule that a validator must completely understand the validation behavior still applies by default, though this too may have exceptions.

> *Note the meta-schema is not the final authority in the validity of a schema, there may still be application-level errors. For example, unknown keywords in a schema may be best implemented as an application error, rather than in the meta-schema; however see the Academic Notes section.*

The referenced resource does not necessarily need to be a JSON Schema, it may be any resource that provides these same semantics.

> *This keyword could also be defined to be strictly descriptive (as originally defined) but it's unlikely that would be more useful. An "@schema" keyword could do that.*
>
> *The "`$`" keywords are also called "core keywords" but I would like to suggest calling them "system keywords". I've avoided both terms in this document.*

"$schema" is only defined over arguments whose URI is meaningful to the validator, so if an unknown value is passed in, this would stop validation. This means that if the $schema keyword is not strictly necessary for ensuring correct validation behavior, an annotation-only variation should be used instead.

$vocabulary

This is technically an extension to the $schema keyword.

This will read the meta-schema referenced by that $schema keyword, and selectively enable extensions depending on the contents of $vocabulary in that meta-schema.

unevaluatedProperties

This it is not included in the required set, because it's not immediately apparent if it actually enables new kinds of validation, or if it's just an authoring convenience for what is already possible.

uniqueItems

This is a way to verify that an array represents a set with only unique items; no two items are identical.

It is excluded from the required set because it cannot be evaluated in better than $O(m^2)$ time or or O(m) memory (m = size of instance).

Formats

"format" applies an arbitrary rule to the instance.

It accepts a format name in its keyword argument, and is only defined over format names known to the validator, meaning that unknown arguments will raise an error.

User formats could be specified by passing a URI; how to validate a user format could be specified as a regular expression or function, passed out-of-band to the validator.

If validation is not needed, use "@format" to suppress validation, but keep the other semantics and annotations.

Annotations

The behavior of annotation keywords as a whole, as well as the standard meaning of individual annotations, may be defined as an extension. This is because almost all support for annotations can be determined from the validation result. (If a validator doesn't support an annotation as used in a schema, it will usually be obvious to the consuming application.)

- title
- description
- default
- deprecated
- readOnly
- writeOnly
- examples

5. Obsolete Behavior Compatibility

This section lists a number of OPTIONAL extensions that validators may implement, to maximize backwards compatibility with schemas "in the wild": those written according to previous drafts or popular nonstandard extensions, that would otherwise be incomplete.

Typically, when there's a mechanism to do so, validators should generate a deprecation warning indicating that the schema is not interoperable.

These extensions are "optional" because there is no interoperability benefit to saying "recommended". The benefit, if any, is only to the implementation supporting it; there is little negative externality being avoided by using "recommended".

Implementing all of these should preserve backwards compatibility for schemas since draft-03.

Special cases of $schema

$schema may define special behaviors for previously published meta-schemas, when the extensions below do not suffice.

Boolean exclusiveMinimum/exclusiveMaximum

If the schema matches:

```
requiredProperties: {
      "minimum": { type: "number" },
      "exclusiveMinimum": { type: "boolean" }
}
```

Then "minimum" is an alias for "exclusiveMinimum".

This same pattern also applies to "maximum".

Boolean "required"

When used within a subschema of the "properties" keyword, this is the same as listing the name of the respective property in the "required" keyword that is sibling to "properties".

"divisibleBy"

An alias for "multipleOf".

Only defined if "multipleOf" is not used.

"disallow"

Reject if the instance is one of the types listed in the argument. Operates similar to "type".

“extends”

I don't know what this was ever intending to do. Apparently it works like $ref but isn’t the same thing.

“dependencies”

This may be an alias for “dependentSchemas" or “dependentRequired” depending on the argument.

Only defined if “dependentSchemas” and “dependentRequired” are both not used.

“id”

An alias for “$id”.

Only defined if “$id” is not used. (This means both “id” and “$id” in the same schema is undefined.)

Fragment in “$id”

The “$id” keyword is undefined for values that resolve to a fragment URI; this extension defines that behavior.

An alias for $id and $anchor, split at the first “#” character.

(Tests: write a test for `{“id”: “foo.json#bar#2”}` the extra “#” character may not match the anchor-name syntax, though it is a legal URI.)

6. Notes and Observations

A keyword for strict handling of unknown properties?

This extension model may seem to have much in common with `{"additionalProperties": false}`, or `{unevaluatedProperties: false}`, although frustratingly, it is not exactly the same thing.

This extension model is more akin to saying "There should be some set of (zero or more) JSON Schemas that together annotate every property in the schema, and if there's not, that's an error." No schema by itself errors on an unknown property, the lack of ability to parse a portion of the JSON document does.

The "unevaluatedProperties" keyword may actually be suitable (I need to double-check), but it may also may not be generic enough: We may know a keyword's role in annotations but not its role in validation, and unevaluatedProperties cannot distinguish between these cases.

There is also no easy way to describe recursion; the "required" keywords may be described as a meta-schema, but anything more complicated than this means that each validator has its own meta-schema that describes the behavior it implements.

> *Meta-schemas describe the behavior that a validator implements, which is not necessarily the behavior that the schema is using.*

There ought to be a way to list multiple JSON Schemas, instead of taking the union (alternation) of the objects as a whole ("an object with a 'foo' keyword, or a 'bar' keyword"), you take the union of *their properties*.

"There exists some set derivative objects (one per subschema) where every property in the instance is assigned to one of the derivative objects, such that every derivative object is valid against a different sub-schema."

Example:

```
A.json:

{ properties: {
      "foo": { type: "string" },
      "bar": { type: "string" },
}

B.json:

{ properties: {
      "foo": { type: "number" },
      "bar": { type: "number" },
}


allOf A.json, B.json: "foo" or "bar" always invalid
anyOf A.json, B.json: "foo" and "bar" must both the same type
union A.json, B.json: "foo" or "bar" may each be a number or string
```

This “union” keyword may more completely solve the problem that “unevaluatedProperties” was trying to solve.

Two remaining backwards compatibility cases

I’m aware of only two remaining cases that’s not addressed elsewhere in this document.

First is $ref. Properties in any schema containing the $ref keyword were at one time supposed to be ignored, and the contents of the target substituted in. This was changed because this rule was widely ignored, and it was easier to make $ref a regular keyword like all the others, and this makes more sense anyways. It is also unlikely that schemas put validation keywords next to $ref and expected these to be ignored.

Second is “integer”. At one point it did not permit a decimal point, now it refers to “mathematical” integers which may. If the idea was that JSON should have separate types for floats and integers, then they should not overlap, but they did (“number” includes “integer”). And you can’t forget about the scientific notation format, which has an “e”. Overall this was never implemented consistently and was never interoperable. Of the implementations that depend on integers not having a decimal point, very few of them probably read instances that provide a decimal number. The few implementations that need to prohibit or permit decimal points should use “numberFormat”.

Set operations on assertions

Up to now most validators assume that an “assertion that is always valid” is the same as “no assertion” or ignoring the assertion.

This assumption only holds for the intersection operation.

JSON Schema doesn’t use any operations besides intersection, but it’s plausible that union, difference, etc could become useful, in which case this assumption may no longer hold.

Annotating API keywords?

The $vocabulary keyword is solving a problem highly related to those seen in other APIs. It, or a similar keyword, ought to be able to specify the semantics of any JSON property, without specifying its format, validation, or anything else about it. This could be an annotation-only version of $ref (`"@ref"` ? `"@$ref"` ?) or format (or `"@format"`).

Post-parsing validation

Most applications don’t care about how the JSON document is actually *encoded*—they care about how the *application* reads the data.

For example, expressing that the number will be converted to a float64, and *that* number must be a whole number within Number.MAX_SAFE_INTEGER.

This is probably outside the scope of this interoperability scheme, but it could be, is still related to interoperability in a sense.

The multiple uses of $schema

This keyword has historically been used for many different uses:

1. A schema annotates the definitions of keywords for authoring convenience.
2. A validator declares the schemas that it is compatible with.
3. A schema declares the minimum functionality it requires to work.
4. A schema internally asserts it is using keywords correctly, to catch authoring errors.
5. Import a dialect that replaces the definitions of keywords.

Of these, uses 1-4 are accomplished by other mechanisms in this paper, so the use of $schema is reduced in scope to use 5 (importing a dialect).

Supersets of the JSON data model

This is technically outside the scope of JSON Schema, but maybe some thought could be given to handling supersets of JSON Schema. If you were to write a program that compiled a JSON Schema into an ABNF that consumes CBOR—how would that work? Could you have CBOR-specific keywords?

This is especially necessary because most implementations validate native memory structures and not raw JSON—meaning things like `NaN`, `undefined`, and recursive pointers and loops can appear and cause unexpected results.