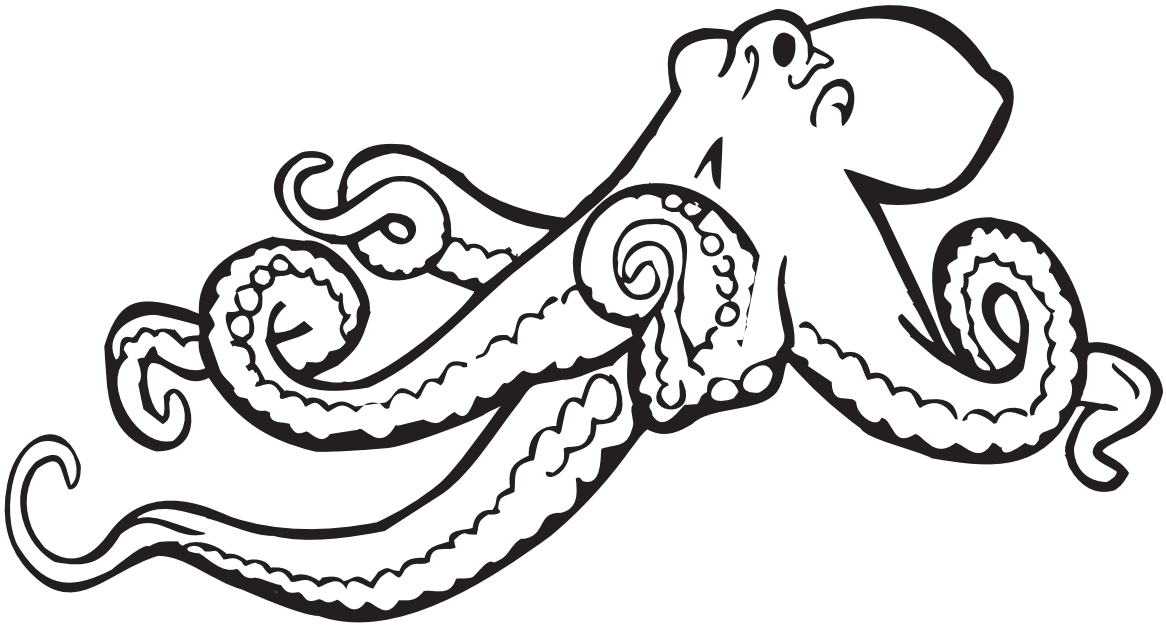


Understanding JSON Schema

Release 2020-12

Michael Droettboom, et al
Space Telescope Science Institute

Jan 13, 2023

Contents

JSON Schema is a powerful tool for validating the structure of JSON data. However, learning to use it by reading its specification is like learning to drive a car by looking at its blueprints. You don't need to know how an electric motor fits together if all you want to do is pick up the groceries. This book, therefore, aims to be the friendly driving instructor for JSON Schema. It's for those that want to write it and understand it, but maybe aren't interested in building their own car—er, writing their own JSON Schema validator—just yet.

Note: This book describes JSON Schema draft 2020-12. Earlier versions of JSON Schema are not completely compatible with the format described here, but for the most part, those differences are noted in the text.

Where to begin?

- This book uses some novel *conventions* (page 3) for showing schema examples and relating JSON Schema to your programming language of choice.
- If you're not sure what a schema is, check out *What is a schema?* (page 7).
- *The basics* (page 11) chapter should be enough to get you started with understanding the core *JSON Schema Reference* (page 15).
- When you start developing large schemas with many nested and repeated sections, check out *Structuring a complex schema* (page 73).
- json-schema.org has a number of resources, including the official specification and tools for working with JSON Schema from various programming languages.
- There are a number of online JSON Schema tools that allow you to run your own JSON schemas against example documents. These can be very handy if you want to try things out without installing any software.

CHAPTER 1

Conventions used in this book

1.1 Language-specific notes

The names of the basic types in JavaScript and JSON can be confusing when coming from another dynamic language. I'm a Python programmer by day, so I've notated here when the names for things are different from what they are in Python, and any other Python-specific advice for using JSON and JSON Schema. I'm by no means trying to create a Python bias to this book, but it is what I know, so I've started there. In the long run, I hope this book will be useful to programmers of all stripes, so if you're interested in translating the Python references into Algol-68 or any other language you may know, pull requests are welcome!

The language-specific sections are shown with tabs for each language. Once you choose a language, that choice will be remembered as you read on from page to page.

For example, here's a language-specific section with advice on using JSON in a few different languages:

Python

In Python, JSON can be read using the json module in the standard library.

Ruby

In Ruby, JSON can be read using the json gem.

C

For C, you may want to consider using Jansson to read and write JSON.

1.2 Draft-specific notes

The JSON Schema standard has been through a number of revisions or "drafts". The current version is Draft 2020-12, but some older drafts are still widely used as well.

The text is written to encourage the use of Draft 2020-12 and gives priority to the latest conventions and features, but where it differs from earlier drafts, those differences are highlighted in special call-outs. If you only wish to target Draft 2020-12, you can safely ignore those sections.

New in draft 2020-12

Draft 2019-09

This is where anything pertaining to an old draft would be mentioned.

1.3 Examples

There are many examples throughout this book, and they all follow the same format. At the beginning of each example is a short JSON schema, illustrating a particular principle, followed by short JSON snippets that are either valid or invalid against that schema. Valid examples are in green, with a checkmark. Invalid examples are in red, with a cross. Often there are comments in between to explain why something is or isn't valid.

Note: These examples are tested automatically whenever the book is built, so hopefully they are not just helpful, but also correct!

For example, here's a snippet illustrating how to use the `number` type:

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
-1
```

Simple floating point number:

```
5.0
```

Exponential notation also works:

```
2.99792458e8
```

Numbers as strings are rejected:

```
"42"
```

CHAPTER 2

What is a schema?

If you've ever used XML Schema, RelaxNG or ASN.1 you probably already know what a schema is and you can happily skip along to the next section. If all that sounds like gobbledygook to you, you've come to the right place. To define what JSON Schema is, we should probably first define what JSON is.

JSON stands for "JavaScript Object Notation", a simple data interchange format. It began as a notation for the world wide web. Since JavaScript exists in most web browsers, and JSON is based on JavaScript, it's very easy to support there. However, it has proven useful enough and simple enough that it is now used in many other contexts that don't involve web surfing.

At its heart, JSON is built on the following data structures:

- object:

  ```
  { "key1": "value1", "key2": "value2" }
  ```

- array:

  ```
  [ "first", "second", "third" ]
  ```

- number:

  ```
  42
  3.1415926
  ```

- string:

  ```
  "This is a string"
  ```

- boolean:

  ```
  true
  false
  ```

- null:

```
null
```

These types have analogs in most programming languages, though they may go by different names.

Python

The following table maps from the names of JSON types to their analogous types in Python:

JSON	Python
string	string
number	int/float
object	dict
array	list
boolean	bool
null	None

45

[4] Since JSON strings always support unicode, they are analogous to `unicode` on Python 2.x and `str` on Python 3.x.
[5] JSON does not have separate types for integer and floating-point.

Ruby

The following table maps from the names of JSON types to their analogous types in Ruby:

JSON	Ruby
string	String
number	Integer/Float
object	Hash
array	Array
boolean	TrueClass/FalseClass
null	NilClass

6

[6] JSON does not have separate types for integer and floating-point.

With these simple data types, all kinds of structured data can be represented. With that great flexibility comes great responsibility, however, as the same concept could be represented in myriad ways. For example, you could imagine representing information about a person in JSON in different ways:

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

```
{
  "first_name": "George",
```

(continues on next page)

(continued from previous page)

```
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

Both representations are equally valid, though one is clearly more formal than the other. The design of a record will largely depend on its intended use within the application, so there's no right or wrong answer here. However, when an application says "give me a JSON record for a person", it's important to know exactly how that record should be organized. For example, we need to know what fields are expected, and how the values are represented. That's where JSON Schema comes in. The following JSON Schema fragment describes how the second example above is structured. Don't worry too much about the details for now. They are explained in subsequent chapters.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "birthday": { "type": "string", "format": "date" },
    "address": {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type" : "string" }
      }
    }
  }
}
```

By "validating" the first example against this schema, you can see that it fails:

✗

```
{
  "name": "George Washington",
  "birthday": "February 22, 1732",
  "address": "Mount Vernon, Virginia, United States"
}
```

However, the second example passes:

✓

```
{
  "first_name": "George",
  "last_name": "Washington",
  "birthday": "1732-02-22",
  "address": {
    "street_address": "3200 Mount Vernon Memorial Highway",
    "city": "Mount Vernon",
    "state": "Virginia",
    "country": "United States"
  }
}
```

You may have noticed that the JSON Schema itself is written in JSON. It is data itself, not a computer program. It's just a declarative format for "describing the structure of other data". This is both its strength and its weakness (which it shares with other similar schema languages). It is easy to concisely describe the surface structure of data, and automate validating data against it. However, since a JSON Schema can't contain arbitrary code, there are certain constraints on the relationships between data elements that can't be expressed. Any "validation tool" for a sufficiently complex data format, therefore, will likely have two phases of validation: one at the schema (or structural) level, and one at the semantic level. The latter check will likely need to be implemented using a more general-purpose programming language.

CHAPTER 3

The basics

In *What is a schema?* (page 7), we described what a schema is, and hopefully justified the need for schema languages. Here, we proceed to write a simple JSON Schema.

3.1 Hello, World!

When learning any new language, it's often helpful to start with the simplest thing possible. In JSON Schema, an empty object is a completely valid schema that will accept any valid JSON.

{ json schema }

```
{ }
```

This accepts anything, as long as it's valid JSON

✓

```
42
```

✓

```
"I'm a string"
```

✓

```
{ "an": [ "arbitrarily", "nested" ], "data": "structure" }
```

New in draft 6

You can also use `true` in place of the empty object to represent a schema that matches anything, or `false` for a schema that matches nothing.

{ json schema }

```
true
```

This accepts anything, as long as it's valid JSON

✓

```
42
```

✓

```
"I'm a string"
```

✓

```
{ "an": [ "arbitrarily", "nested" ], "data": "structure" }
```

{ json schema }

```
false
```

✗

```
"Resistance is futile...  This will always fail!!!"
```

3.2 The type keyword

Of course, we wouldn't be using JSON Schema if we wanted to just accept any JSON document. The most common thing to do in a JSON Schema is to restrict to a specific type. The `type` keyword is used for that.

Note: When this book refers to JSON Schema "keywords", it means the "key" part of the key/value pair in an object. Most of the work of writing a JSON Schema involves mapping a special "keyword" to a value within an object.

For example, in the following, only strings are accepted:

{ json schema }

```
{ "type": "string" }
```

✓

```
"I'm a string"
```

✗

```
42
```

The `type` keyword is described in more detail in *Type-specific keywords* (page 15).

3.3 Declaring a JSON Schema

It's not always easy to tell which draft a JSON Schema is using. You can use the `$schema` keyword to declare which version of the JSON Schema specification the schema is written to. See *$schema* (page 69) for more information. It's generally good practice to include it, though it is not required.

Note: For brevity, the `$schema` keyword isn't included in most of the examples in this book, but it should always be used in the real world.

{ json schema }

```
{ "$schema": "https://json-schema.org/draft/2020-12/schema" }
```

Draft 4

In Draft 4, a `$schema` value of `http://json-schema.org/schema#` referred to the latest version of JSON Schema. This usage has since been deprecated and the use of specific version URIs is required.

3.4 Declaring a unique identifier

It is also best practice to include an `$id` property as a unique identifier for each schema. For now, just set it to a URL at a domain you control, for example:

```
{ "$id": "http://yourdomain.com/schemas/myschema.json" }
```

The details of *$id* (page 75) become more apparent when you start *Structuring a complex schema* (page 73).

New in draft 6

Draft 4

In Draft 4, `$id` is just `id` (without the dollar-sign).

CHAPTER 4

JSON Schema Reference

4.1 Type-specific keywords

The `type` keyword is fundamental to JSON Schema. It specifies the data type for a schema.

At its core, JSON Schema defines the following basic types:

- *string* (page 17)
- *number* (page 25)
- *integer* (page 24)
- *object* (page 29)
- *array* (page 41)
- *boolean* (page 49)
- *null* (page 50)

These types have analogs in most programming languages, though they may go by different names.

Python

The following table maps from the names of JSON types to their analogous types in Python:

JSON	Python
string	string
number	int/float
object	dict
array	list
boolean	bool
null	None

```
45
```

[4] Since JSON strings always support unicode, they are analogous to `unicode` on Python 2.x and `str` on Python 3.x.
[5] JSON does not have separate types for integer and floating-point.

Ruby

The following table maps from the names of JSON types to their analogous types in Ruby:

JSON	Ruby
string	String
number	Integer/Float
object	Hash
array	Array
boolean	TrueClass/FalseClass
null	NilClass

```
6
```

[6] JSON does not have separate types for integer and floating-point.

The `type` keyword may either be a string or an array:

- If it's a string, it is the name of one of the basic types above.
- If it is an array, it must be an array of strings, where each string is the name of one of the basic types, and each element is unique. In this case, the JSON snippet is valid if it matches *any* of the given types.

Here is a simple example of using the `type` keyword:

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
42.0
```

This is not a number, it is a string containing a number.

✗

```
"42"
```

In the following example, we accept strings and numbers, but not structured data types:

{ json schema }

```
{ "type": ["number", "string"] }
```

✓

```
42
```

✓

```
"Life, the universe, and everything"
```

✗

```
["Life", "the universe", "and everything"]
```

For each of these types, there are keywords that only apply to those types. For example, numeric types have a way of specifying a numeric range, that would not be applicable to other types. In this reference, these validation keywords are described along with each of their corresponding types in the following chapters.

4.2 string

The `string` type is used for strings of text. It may contain Unicode characters.

Python

In Python, "string" is analogous to the `unicode` type on Python 2.x, and the `str` type on Python 3.x.

Ruby

In Ruby, "string" is analogous to the `String` type.

{ json schema }

```
{ "type": "string" }
```

✓

```
"This is a string"
```

Unicode characters:

✓

```
"Déjà vu"
```

✓

```
""
```

✓

```
"42"
```

✗

```
42
```

4.2.1 Length

The length of a string can be constrained using the `minLength` and `maxLength` keywords. For both keywords, the value must be a non-negative number.

{ json schema }

```
{
  "type": "string",
  "minLength": 2,
  "maxLength": 3
}
```

✗

```
"A"
```

✓

```
"AB"
```

✓

```
"ABC"
```

✗

```
"ABCD"
```

4.2.2 Regular Expressions

The `pattern` keyword is used to restrict a string to a particular regular expression. The regular expression syntax is the one defined in JavaScript (ECMA 262 specifically) with Unicode support. See *Regular Expressions* (page 22) for more information.

Note: When defining the regular expressions, it's important to note that the string is considered valid if the expression matches anywhere within the string. For example, the regular expression `"p"` will match any string with a p in it, such as `"apple"` not just a string that is simply `"p"`. Therefore, it is usually less confusing, as a matter of course, to surround the regular expression in `^...$`, for example, `"^p$"`, unless there is a good reason not to do so.

The following example matches a simple North American telephone number with an optional area code:

{ json schema }

```
{
   "type": "string",
   "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

✓

```
"555-1212"
```

✓

```
"(888)555-1212"
```

✗

```
"(888)555-1212 ext. 532"
```

✗

```
"(800)FLOWERS"
```

4.2.3 Format

The `format` keyword allows for basic semantic identification of certain kinds of string values that are commonly used. For example, because JSON doesn't have a "DateTime" type, dates need to be encoded as strings. `format` allows the schema author to indicate that the string value should be interpreted as a date. By default, `format` is just an annotation and does not effect validation.

Optionally, validator implementations can provide a configuration option to enable `format` to function as an assertion rather than just an annotation. That means that validation will fail if, for example, a value with a `date` format isn't in a form that can be parsed as a date. This can allow values to be constrained beyond what the other tools in JSON Schema, including *Regular Expressions* (page 22) can do.

Note: Implementations may provide validation for only a subset of the built-in formats or do partial validation for a given format. For example, some implementations may consider a string an email if it contains a `@`, while others might do additional checks for other aspects of a well formed email address.

Draft 4-7

In Draft 4-7, there is no guarantee that you get annotation-only behavior by default.

There is a bias toward networking-related formats in the JSON Schema specification, most likely due to its heritage in web technologies. However, custom formats may also be used, as long as the parties exchanging the JSON documents also exchange information about the custom format types. A JSON Schema validator will ignore any format type that it does not understand.

Built-in formats

The following is the list of formats specified in the JSON Schema specification.

Dates and times

Dates and times are represented in RFC 3339, section 5.6. This is a subset of the date format also commonly known as ISO8601 format.

- `"date-time"`: Date and time together, for example, `2018-11-13T20:20:39+00:00`.
- `"time"`: New in draft 7 Time, for example, `20:20:39+00:00`
- `"date"`: New in draft 7 Date, for example, `2018-11-13`.
- `"duration"`: New in draft 2019-09 A duration as defined by the ISO 8601 ABNF for "duration". For example, `P3D` expresses a duration of 3 days.

Email addresses

- `"email"`: Internet email address, see RFC 5321, section 4.1.2.
- `"idn-email"`: New in draft 7 The internationalized form of an Internet email address, see RFC 6531.

Hostnames

- `"hostname"`: Internet host name, see RFC 1123, section 2.1.
- `"idn-hostname"`: New in draft 7 An internationalized Internet host name, see RFC5890, section 2.3.2.3.

IP Addresses

- `"ipv4"`: IPv4 address, according to dotted-quad ABNF syntax as defined in RFC 2673, section 3.2.
- `"ipv6"`: IPv6 address, as defined in RFC 2373, section 2.2.

Resource identifiers

- `"uuid"`: New in draft 2019-09 A Universally Unique Identifier as defined by RFC 4122. Example: `3e4666bf-d5e5-4aa7-b8ce-cefe41c7568a`
- `"uri"`: A universal resource identifier (URI), according to RFC3986.
- `"uri-reference"`: New in draft 6 A URI Reference (either a URI or a relative-reference), according to RFC3986, section 4.1.
- `"iri"`: New in draft 7 The internationalized equivalent of a "uri", according to RFC3987.
- `"iri-reference"`: New in draft 7 The internationalized equivalent of a "uri-reference", according to RFC3987

If the values in the schema have the ability to be relative to a particular source path (such as a link from a webpage), it is generally better practice to use `"uri-reference"` (or `"iri-reference"`) rather than `"uri"` (or `"iri"`). `"uri"` should only be used when the path must be absolute.

Draft 4

Draft 4 only includes `"uri"`, not `"uri-reference"`. Therefore, there is some ambiguity around whether `"uri"` should accept relative paths.

URI template

- `"uri-template"`: New in draft 6 A URI Template (of any level) according to RFC6570. If you don't already know what a URI Template is, you probably don't need this value.

JSON Pointer

- `"json-pointer"`: New in draft 6 A JSON Pointer, according to RFC6901. There is more discussion on the use of JSON Pointer within JSON Schema in *Structuring a complex schema* (page 73). Note that this should be used only when the entire string contains only JSON Pointer content, e.g. `/foo/bar`. JSON Pointer URI fragments, e.g. `#/foo/bar/` should use `"uri-reference"`.
- `"relative-json-pointer"`: New in draft 7 A relative JSON pointer.

Regular Expressions

- `"regex"`: New in draft 7 A regular expression, which should be valid according to the ECMA 262 dialect.

Be careful, in practice, JSON schema validators are only required to accept the safe subset of *Regular Expressions* (page 22) described elsewhere in this document.

4.3 Regular Expressions

- *Example* (page 23)

The *pattern* (page 19) and *Pattern Properties* (page 31) keywords use regular expressions to express constraints. The regular expression syntax used is from JavaScript (ECMA 262, specifically). However, that complete syntax is not widely supported, therefore it is recommended that you stick to the subset of that syntax described below.

- A single unicode character (other than the special characters below) matches itself.
- `.`: Matches any character except line break characters. (Be aware that what constitutes a line break character is somewhat dependent on your platform and language environment, but in practice this rarely matters).
- `^`: Matches only at the beginning of the string.
- `$`: Matches only at the end of the string.
- `(...)`: Group a series of regular expressions into a single regular expression.
- `|`: Matches either the regular expression preceding or following the `|` symbol.
- `[abc]`: Matches any of the characters inside the square brackets.
- `[a-z]`: Matches the range of characters.
- `[^abc]`: Matches any character *not* listed.

- `[^a-z]`: Matches any character outside of the range.
- `+`: Matches one or more repetitions of the preceding regular expression.
- `*`: Matches zero or more repetitions of the preceding regular expression.
- `?`: Matches zero or one repetitions of the preceding regular expression.
- `+?`, `*?`, `??`: The `*`, `+`, and `?` qualifiers are all greedy; they match as much text as possible. Sometimes this behavior isn't desired and you want to match as few characters as possible.
- `(?!x)`, `(?=x)`: Negative and positive lookahead.
- `{x}`: Match exactly `x` occurrences of the preceding regular expression.
- `{x,y}`: Match at least `x` and at most `y` occurrences of the preceding regular expression.
- `{x,}`: Match `x` occurrences or more of the preceding regular expression.
- `{x}?`, `{x,y}?`, `{x,}?`: Lazy versions of the above expressions.

4.3.1 Example

The following example matches a simple North American telephone number with an optional area code:

{ json schema }

```
{
   "type": "string",
   "pattern": "^(\\([0-9]{3}\\))?[0-9]{3}-[0-9]{4}$"
}
```

✓

```
"555-1212"
```

✓

```
"(888)555-1212"
```

✗

```
"(888)555-1212 ext. 532"
```

✗

```
"(800)FLOWERS"
```

4.4 Numeric types

There are two numeric types in JSON Schema: *integer* (page 24) and *number* (page 25). They share the same validation keywords.

Note: JSON has no standard way to represent complex numbers, so there is no way to test for them in JSON Schema.

4.4.1 integer

The `integer` type is used for integral numbers. JSON does not have distinct types for integers and floating-point values. Therefore, the presence or absence of a decimal point is not enough to distinguish between integers and non-integers. For example, `1` and `1.0` are two ways to represent the same value in JSON. JSON Schema considers that value an integer no matter which representation was used.

Python

In Python, "integer" is analogous to the `int` type.

Ruby

In Ruby, "integer" is analogous to the `Integer` type.

{ json schema }

```
{ "type": "integer" }
```

✓

```
42
```

✓

```
-1
```

Numbers with a zero fractional part are considered integers

✓

```
1.0
```

Floating point numbers are rejected:

✗

```
3.1415926
```

Numbers as strings are rejected:

✗

```
"42"
```

4.4.2 number

The `number` type is used for any numeric type, either integers or floating point numbers.

Python

In Python, "number" is analogous to the `float` type.

Ruby

In Ruby, "number" is analogous to the `Float` type.

{ json schema }

```
{ "type": "number" }
```

✓

```
42
```

✓

```
-1
```

Simple floating point number:

✓

```
5.0
```

Exponential notation also works:

✓

```
2.99792458e8
```

Numbers as strings are rejected:

✗

```
"42"
```

4.4.3 Multiples

Numbers can be restricted to a multiple of a given number, using the `multipleOf` keyword. It may be set to any positive number.

{ json schema }

```
{
    "type": "number",
    "multipleOf" : 10
}
```

✓

```
0
```

✓

```
10
```

✓

```
20
```

Not a multiple of 10:

✗

```
23
```

4.4.4 Range

Ranges of numbers are specified using a combination of the `minimum` and `maximum` keywords, (or `exclusiveMinimum` and `exclusiveMaximum` for expressing exclusive range).

If x is the value being validated, the following must hold true:

- $x \geq$ `minimum`
- $x >$ `exclusiveMinimum`
- $x \leq$ `maximum`

- $x <$ `exclusiveMaximum`

While you can specify both of `minimum` and `exclusiveMinimum` or both of `maximum` and `exclusiveMaximum`, it doesn't really make sense to do so.

{ json schema }

```
{
  "type": "number",
  "minimum": 0,
  "exclusiveMaximum": 100
}
```

Less than `minimum`:

✗

```
-1
```

`minimum` is inclusive, so 0 is valid:

✓

```
0
```

✓

```
10
```

✓

```
99
```

`exclusiveMaximum` is exclusive, so 100 is not valid:

✗

```
100
```

Greater than `maximum`:

✗

```
101
```

Draft 4

In JSON Schema Draft 4, `exclusiveMinimum` and `exclusiveMaximum` work differently. There they are boolean values, that indicate whether `minimum` and `maximum` are exclusive of the value. For example:

- if `exclusiveMinimum` is `false`, $x \geq$ `minimum`.
- if `exclusiveMinimum` is `true`, $x >$ `minimum`.

This was changed to have better keyword independence.

Here is an example using the older Draft 4 convention:

{ json schema }

```
{
  "type": "number",
  "minimum": 0,
  "maximum": 100,
  "exclusiveMaximum": true
}
```

Less than `minimum`:

✗

```
-1
```

`exclusiveMinimum` was not specified, so 0 is included:

✓

```
0
```

✓

```
10
```

✓

```
99
```

`exclusiveMaximum` is `true`, so 100 is not included:

✗

```
100
```

Greater than `maximum`:

✗

```
101
```

4.5 object

Objects are the mapping type in JSON. They map “keys” to “values”. In JSON, the “keys” must always be strings. Each of these pairs is conventionally referred to as a “property”.

Python

In Python, "objects" are analogous to the `dict` type. An important difference, however, is that while Python dictionaries may use anything hashable as a key, in JSON all the keys must be strings.

Try not to be confused by the two uses of the word "object" here: Python uses the word `object` to mean the generic base class for everything, whereas in JSON it is used only to mean a mapping from string keys to values.

Ruby

In Ruby, "objects" are analogous to the `Hash` type. An important difference, however, is that all keys in JSON must be strings, and therefore any non-string keys are converted over to their string representation.

Try not to be confused by the two uses of the word "object" here: Ruby uses the word `Object` to mean the generic base class for everything, whereas in JSON it is used only to mean a mapping from string keys to values.

{ json schema }

```
{ "type": "object" }
```

✓

```
{
   "key": "value",
   "another_key": "another_value"
}
```

✓

```
{
    "Sun": 1.9891e30,
    "Jupiter": 1.8986e27,
    "Saturn": 5.6846e26,
    "Neptune": 10.243e25,
    "Uranus": 8.6810e25,
    "Earth": 5.9736e24,
    "Venus": 4.8685e24,
    "Mars": 6.4185e23,
    "Mercury": 3.3022e23,
    "Moon": 7.349e22,
    "Pluto": 1.25e22
}
```

Using non-strings as keys is invalid JSON:

✗

```
{
    0.01: "cm",
    1: "m",
    1000: "km"
}
```

✗

```
"Not an object"
```

✗

```
["An", "array", "not", "an", "object"]
```

4.5.1 Properties

The properties (key-value pairs) on an object are defined using the `properties` keyword. The value of `properties` is an object, where each key is the name of a property and each value is a schema used to validate that property. Any property that doesn't match any of the property names in the `properties` keyword is ignored by this keyword.

Note: See *Additional Properties* (page 32) and *Unevaluated Properties* (page 36) for how to disallow properties that don't match any of the property names in `properties`.

For example, let's say we want to define a simple schema for an address made up of a number, street name and street type:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": ["Street", "Avenue", "Boulevard"] }
  }
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

If we provide the number in the wrong type, it is invalid:

✗

```
{ "number": "1600", "street_name": "Pennsylvania", "street_type": "Avenue" }
```

By default, leaving out properties is valid. See *Required Properties* (page 39).

✓

```
{ "number": 1600, "street_name": "Pennsylvania" }
```

By extension, even an empty object is valid:

✓

```
{ }
```

By default, providing additional properties is valid:

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue", "direction": "NW" }
```

4.5.2 Pattern Properties

Sometimes you want to say that, given a particular kind of property name, the value should match a particular schema. That's where `patternProperties` comes in: it maps regular expressions to schemas. If a property name matches the given regular expression, the property value must validate against the corresponding schema.

Note: Regular expressions are not anchored. This means that when defining the regular expressions for `patternProperties`, it's important to note that the expression may match anywhere within the property name. For example, the regular expression `"p"` will match any property name with a `p` in it, such as `"apple"`, not just a property whose name is simply `"p"`. It's therefore usually less confusing to surround the regular expression in `^...$`,

for example, `"^p$"`.

In this example, any properties whose names start with the prefix `S_` must be strings, and any with the prefix `I_` must be integers. Any properties that do not match either regular expression are ignored.

{ json schema }

```
{
  "type": "object",
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  }
}
```

✓

```
{ "S_25": "This is a string" }
```

✓

```
{ "I_0": 42 }
```

If the name starts with `S_`, it must be a string

✗

```
{ "S_0": 42 }
```

If the name starts with `I_`, it must be an integer

✗

```
{ "I_42": "This is a string" }
```

This is a key that doesn't match any of the regular expressions:

✓

```
{ "keyword": "value" }
```

4.5.3 Additional Properties

The `additionalProperties` keyword is used to control the handling of extra stuff, that is, properties whose names are not listed in the `properties` keyword or match any of the regular expressions in the `patternProperties` keyword. By default any additional properties are allowed.

The value of the `additionalProperties` keyword is a schema that will be used to validate any properties in the instance that are not matched by `properties` or `patternProperties`. Setting the `additionalProperties` schema to

`false` means no additional properties will be allowed.

Reusing the example from *Properties* (page 30), but this time setting `additionalProperties` to `false`.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": ["Street", "Avenue", "Boulevard"] }
  },
  "additionalProperties": false
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

Since `additionalProperties` is `false`, this extra property "direction" makes the object invalid:

✗

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue", "direction": "NW" }
```

You can use non-boolean schemas to put more complex constraints on the additional properties of an instance. For example, one can allow additional properties, but only if their values are each a string:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "number": { "type": "number" },
    "street_name": { "type": "string" },
    "street_type": { "enum": ["Street", "Avenue", "Boulevard"] }
  },
  "additionalProperties": { "type": "string" }
}
```

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue" }
```

This is valid, since the additional property's value is a string:

✓

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue", "direction": "NW" }
```

This is invalid, since the additional property's value is not a string:

✗

```
{ "number": 1600, "street_name": "Pennsylvania", "street_type": "Avenue", "office_number":␣
↪201 }
```

You can use `additionalProperties` with a combination of `properties` and `patternProperties`. In the following example, based on the example from *Pattern Properties* (page 31), we add a `"builtin"` property, which must be a number, and declare that all additional properties (that are neither defined by `properties` nor matched by `patternProperties`) must be strings:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "builtin": { "type": "number" }
  },
  "patternProperties": {
    "^S_": { "type": "string" },
    "^I_": { "type": "integer" }
  },
  "additionalProperties": { "type": "string" }
}
```

✓

```
{ "builtin": 42 }
```

This is a key that doesn't match any of the regular expressions:

✓

```
{ "keyword": "value" }
```

It must be a string:

✗

```
{ "keyword": 42 }
```

Extending Closed Schemas

It's important to note that `additionalProperties` only recognizes properties declared in the same subschema as itself. So, `additionalProperties` can restrict you from "extending" a schema using *Schema Composition* (page 55) keywords such as *allOf* (page 56). In the following example, we can see how the `additionalProperties` can cause attempts to extend the address schema example to fail.

{ json schema }

```
{
  "allOf": [
    {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" }
      },
      "required": ["street_address", "city", "state"],
      "additionalProperties": false
    }
  ],

  "properties": {
    "type": { "enum": [ "residential", "business" ] }
  },
  "required": ["type"]
}
```

Fails `additionalProperties`. "type" is considered additional.

✗

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC",
   "type": "business"
}
```

Fails `required`. "type" is required.

✗

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC"
}
```

Because `additionalProperties` only recognizes properties declared in the same subschema, it considers anything other than "street_address", "city", and "state" to be additional. Combining the schemas with *allOf* (page 56) doesn't change that. A workaround you can use is to move `additionalProperties` to the extending schema and redeclare the properties from the extended schema.

{ json schema }

```
{
  "allOf": [
    {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  ],

  "properties": {
    "street_address": true,
    "city": true,
    "state": true,
    "type": { "enum": [ "residential", "business" ] }
  },
  "required": ["type"],
  "additionalProperties": false
}
```

✓

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC",
   "type": "business"
}
```

✗

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC",
   "type": "business",
   "something that doesn't belong": "hi!"
}
```

Now the `additionalProperties` keyword is able to recognize all the necessary properties and the schema works as expected. Keep reading to see how the `unevaluatedProperties` keyword solves this problem without needing to redeclare properties.

4.5.4 Unevaluated Properties

New in draft 2019-09

In the previous section we saw the challenges with using `additionalProperties` when "extending" a schema using

Schema Composition (page 55). The `unevaluatedProperties` keyword is similar to `additionalProperties` except that it can recognize properties declared in subschemas. So, the example from the previous section can be rewritten without the need to redeclare properties.

{ json schema }

```
{
  "allOf": [
    {
      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" }
      },
      "required": ["street_address", "city", "state"]
    }
  ],

  "properties": {
    "type": { "enum": ["residential", "business"] }
  },
  "required": ["type"],
  "unevaluatedProperties": false
}
```

✓

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC",
   "type": "business"
}
```

✗

```
{
   "street_address": "1600 Pennsylvania Avenue NW",
   "city": "Washington",
   "state": "DC",
   "type": "business",
   "something that doesn't belong": "hi!"
}
```

`unevaluatedProperties` works by collecting any properties that are successfully validated when processing the schemas and using those as the allowed list of properties. This allows you to do more complex things like conditionally adding properties. The following example allows the "department" property only if the "type" of address is "business".

{ json schema }

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" },
    "type": { "enum": ["residential", "business"] }
  },
  "required": ["street_address", "city", "state", "type"],

  "if": {
    "type": "object",
    "properties": {
      "type": { "const": "business" }
    },
    "required": ["type"]
  },
  "then": {
    "properties": {
      "department": { "type": "string" }
    }
  },

  "unevaluatedProperties": false
}
```

✓

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "city": "Washington",
  "state": "DC",
  "type": "business",
  "department": "HR"
}
```

✗

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "city": "Washington",
  "state": "DC",
  "type": "residential",
  "department": "HR"
}
```

In this schema, the properties declared in the `then` schema only count as "evaluated" properties if the "type" of the address is "business".

4.5.5 Required Properties

By default, the properties defined by the `properties` keyword are not required. However, one can provide a list of required properties using the `required` keyword.

The `required` keyword takes an array of zero or more strings. Each of these strings must be unique.

Draft 4

In Draft 4, `required` must contain at least one string.

In the following example schema defining a user record, we require that each user has a name and e-mail address, but we don't mind if they don't provide their address or telephone number:

{ json schema }

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "email": { "type": "string" },
    "address": { "type": "string" },
    "telephone": { "type": "string" }
  },
  "required": ["name", "email"]
}
```

✓

```
{
  "name": "William Shakespeare",
  "email": "bill@stratford-upon-avon.co.uk"
}
```

Providing extra properties is fine, even properties not defined in the schema:

✓

```
{
  "name": "William Shakespeare",
  "email": "bill@stratford-upon-avon.co.uk",
  "address": "Henley Street, Stratford-upon-Avon, Warwickshire, England",
  "authorship": "in question"
}
```

Missing the required "email" property makes the JSON document invalid:

✗

```
{
  "name": "William Shakespeare",
  "address": "Henley Street, Stratford-upon-Avon, Warwickshire, England",
}
```

In JSON a property with value `null` is not equivalent to the property not being present. This fails because `null` is not of type "string", it's of type "null"

✗

```
{
  "name": "William Shakespeare",
  "address": "Henley Street, Stratford-upon-Avon, Warwickshire, England",
  "email": null
}
```

4.5.6 Property names

New in draft 6

The names of properties can be validated against a schema, irrespective of their values. This can be useful if you don't want to enforce specific properties, but you want to make sure that the names of those properties follow a specific convention. You might, for example, want to enforce that all names are valid ASCII tokens so they can be used as attributes in a particular programming language.

{ json schema }

```
{
  "type": "object",
  "propertyNames": {
    "pattern": "^[A-Za-z_][A-Za-z0-9_]*$"
  }
}
```

✓

```
{
  "_a_proper_token_001": "value"
}
```

✗

```
{
  "001 invalid": "value"
}
```

Since object keys must always be strings anyway, it is implied that the schema given to `propertyNames` is always at least:

```
{ "type": "string" }
```

4.5.7 Size

The number of properties on an object can be restricted using the `minProperties` and `maxProperties` keywords. Each of these must be a non-negative integer.

{ json schema }

```
{
  "type": "object",
  "minProperties": 2,
  "maxProperties": 3
}
```

✗

```
{}
```

✗

```
{ "a": 0 }
```

✓

```
{ "a": 0, "b": 1 }
```

✓

```
{ "a": 0, "b": 1, "c": 2 }
```

✗

```
{ "a": 0, "b": 1, "c": 2, "d": 3 }
```

4.6 array

Arrays are used for ordered elements. In JSON, each element in an array may be of a different type.

Python

In Python, "array" is analogous to a `list` or `tuple` type, depending on usage. However, the `json` module in the Python standard library will always use Python lists to represent JSON arrays.

Ruby

In Ruby, "array" is analogous to a `Array` type.

{ json schema }

```
{ "type": "array" }
```

✓

```
[1, 2, 3, 4, 5]
```

✓

```
[3, "different", { "types" : "of values" }]
```

✗

```
{"Not": "an array"}
```

There are two ways in which arrays are generally used in JSON:

- **List validation:** a sequence of arbitrary length where each item matches the same schema.
- **Tuple validation:** a sequence of fixed length where each item may have a different schema. In this usage, the index (or location) of each item is meaningful as to how the value is interpreted. (This usage is often given a whole separate type in some programming languages, such as Python's `tuple`).

4.6.1 Items

List validation is useful for arrays of arbitrary length where each item matches the same schema. For this kind of array, set the `items` keyword to a single schema that will be used to validate all of the items in the array.

In the following example, we define that each item in an array is a number:

{ json schema }

```
{
  "type": "array",
  "items": {
    "type": "number"
  }
}
```

✓

```
[1, 2, 3, 4, 5]
```

A single "non-number" causes the whole array to be invalid:

✗

```
[1, 2, "3", 4, 5]
```

The empty array is always valid:

✓

```
[]
```

4.6.2 Tuple validation

Tuple validation is useful when the array is a collection of items where each has a different schema and the ordinal index of each item is meaningful.

For example, you may represent a street address such as:

```
1600 Pennsylvania Avenue NW
```

as a 4-tuple of the form:

> [number, street_name, street_type, direction]

Each of these fields will have a different schema:

- `number`: The address number. Must be a number.
- `street_name`: The name of the street. Must be a string.
- `street_type`: The type of street. Should be a string from a fixed set of values.
- `direction`: The city quadrant of the address. Should be a string from a different set of values.

To do this, we use the `prefixItems` keyword. `prefixItems` is an array, where each item is a schema that corresponds to each index of the document's array. That is, an array where the first element validates the first element of the input array, the second element validates the second element of the input array, etc.

Draft 4 - 2019-09

In Draft 4 - 2019-09, tuple validation was handled by an alternate form of the `items` keyword. When `items` was an array of schemas instead of a single schema, it behaved the way `prefixItems` behaves.

Here's the example schema:

{ json schema }

```
{
  "type": "array",
  "prefixItems": [
    { "type": "number" },
    { "type": "string" },
    { "enum": ["Street", "Avenue", "Boulevard"] },
    { "enum": ["NW", "NE", "SW", "SE"] }
  ]
}
```

✓

```
[1600, "Pennsylvania", "Avenue", "NW"]
```

"Drive" is not one of the acceptable street types:

✗

```
[24, "Sussex", "Drive"]
```

This address is missing a street number

✗

```
["Palais de l'Élysée"]
```

It's okay to not provide all of the items:

✓

```
[10, "Downing", "Street"]
```

And, by default, it's also okay to add additional items to end:

✓

```
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

Additional Items

The `items` keyword can be used to control whether it's valid to have additional items in a tuple beyond what is defined in `prefixItems`. The value of the `items` keyword is a schema that all additional items must pass in order for the keyword to validate.

Draft 4 - 2019-09

Before to Draft 2020-12, you would use the `additionalItems` keyword to constrain additional items on a tuple. It works the same as `items`, only the name has changed.

Draft 6 - 2019-09

In Draft 6 - 2019-09, the `additionalItems` keyword is ignored if there is not a "tuple validation" `items` keyword present in the same schema.

Here, we'll reuse the example schema above, but set `items` to `false`, which has the effect of disallowing extra items in the tuple.

{ json schema }

```
{
  "type": "array",
  "prefixItems": [
    { "type": "number" },
    { "type": "string" },
    { "enum": ["Street", "Avenue", "Boulevard"] },
    { "enum": ["NW", "NE", "SW", "SE"] }
  ],
  "items": false
}
```

✓

```
[1600, "Pennsylvania", "Avenue", "NW"]
```

It's ok to not provide all of the items:

✓

```
[1600, "Pennsylvania", "Avenue"]
```

But, since `items` is `false`, we can't provide extra items:

✗

```
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

You can express more complex constraints by using a non-boolean schema to constrain what value additional items can have. In that case, we could say that additional items are allowed, as long as they are all strings:

{ json schema }

```
{
  "type": "array",
  "prefixItems": [
    { "type": "number" },
    { "type": "string" },
    { "enum": ["Street", "Avenue", "Boulevard"] },
    { "enum": ["NW", "NE", "SW", "SE"] }
  ],
  "items": { "type": "string" }
}
```

Extra string items are ok …

✓

```
[1600, "Pennsylvania", "Avenue", "NW", "Washington"]
```

… but not anything else

✗

```
[1600, "Pennsylvania", "Avenue", "NW", 20500]
```

4.6.3 Unevaluated Items

New in draft 2019-09

Documentation Coming Soon

4.6.4 Contains

New in draft 6

While the `items` schema must be valid for every item in the array, the `contains` schema only needs to validate against one or more items in the array.

{ json schema }

```
{
   "type": "array",
   "contains": {
     "type": "number"
   }
}
```

A single “number” is enough to make this pass:

✓

```
["life", "universe", "everything", 42]
```

But if we have no number, it fails:

✗

```
["life", "universe", "everything", "forty-two"]
```

All numbers is, of course, also okay:

✓

```
[1, 2, 3, 4, 5]
```

minContains / maxContains

New in draft 2019-09

`minContains` and `maxContains` can be used with `contains` to further specify how many times a schema matches a `contains` constraint. These keywords can be any non-negative number including zero.

{ json schema }

```
{
  "type": "array",
  "contains": {
    "type": "number"
  },
  "minContains": 2,
  "maxContains": 3
}
```

Fails `minContains`

✗

```
["apple", "orange", 2]
```

✓

```
["apple", "orange", 2, 4]
```

✓

```
["apple", "orange", 2, 4, 8]
```

Fails `maxContains`

✗

```
["apple", "orange", 2, 4, 8, 16]
```

4.6.5 Length

The length of the array can be specified using the `minItems` and `maxItems` keywords. The value of each keyword must be a non-negative number. These keywords work whether doing *list validation* (page 42) or *Tuple validation* (page 43).

{ json schema }

```
{
  "type": "array",
  "minItems": 2,
  "maxItems": 3
}
```

✗

```
[]
```

✗

```
[1]
```

✓

```
[1, 2]
```

✓

```
[1, 2, 3]
```

✗

```
[1, 2, 3, 4]
```

4.6.6 Uniqueness

A schema can ensure that each of the items in an array is unique. Simply set the `uniqueItems` keyword to `true`.

{ json schema }

```
{
  "type": "array",
  "uniqueItems": true
}
```

✓

```
[1, 2, 3, 4, 5]
```

✗

```
[1, 2, 3, 3, 4]
```

The empty array always passes:

✓

```
[]
```

4.7 boolean

The boolean type matches only two special values: `true` and `false`. Note that values that *evaluate* to `true` or `false`, such as 1 and 0, are not accepted by the schema.

Python

In Python, "boolean" is analogous to `bool`. Note that in JSON, `true` and `false` are lower case, whereas in Python they are capitalized (`True` and `False`).

Ruby

In Ruby, "boolean" is analogous to `TrueClass` and `FalseClass`. Note that in Ruby there is no `Boolean` class.

{ json schema }

```
{ "type": "boolean" }
```

✓

```
true
```

✓

```
false
```

✗

```
"true"
```

Values that evaluate to `true` or `false` are still not accepted by the schema:

✗

```
0
```

4.8 null

When a schema specifies a `type` of `null`, it has only one acceptable value: `null`.

Note: It's important to remember that in JSON, `null` isn't equivalent to something being absent. See *Required Properties* (page 39) for an example.

Python

In Python, `null` is analogous to `None`.

Ruby

In Ruby, `null` is analogous to `nil`.

{ json schema }

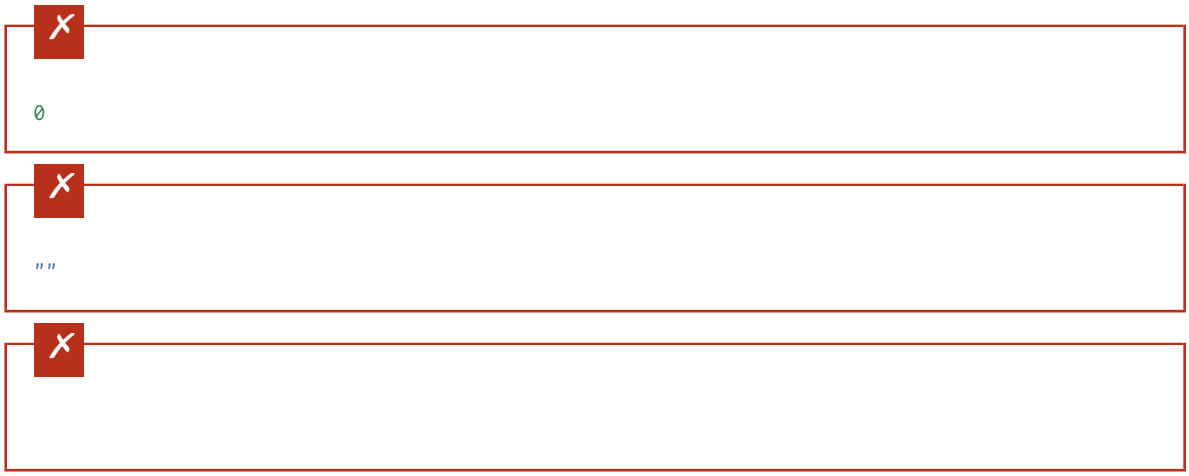
```
{ "type": "null" }
```

✓

```
null
```

✗

```
false
```



4.9 Generic keywords

This chapter lists some miscellaneous properties that are available for all JSON types.

4.9.1 Annotations

JSON Schema includes a few keywords, that aren't strictly used for validation, but are used to describe parts of a schema. None of these "annotation" keywords are required, but they are encouraged for good practice, and can make your schema "self-documenting".

The `title` and `description` keywords must be strings. A "title" will preferably be short, whereas a "description" will provide a more lengthy explanation about the purpose of the data described by the schema.

The `default` keyword specifies a default value. This value is not used to fill in missing values during the validation process. Non-validation tools such as documentation generators or form generators may use this value to give hints to users about how to use a value. However, `default` is typically used to express that if a value is missing, then the value is semantically the same as if the value was present with the default value. The value of `default` should validate against the schema in which it resides, but that isn't required.

New in draft 6 The `examples` keyword is a place to provide an array of examples that validate against the schema. This isn't used for validation, but may help with explaining the effect and purpose of the schema to a reader. Each entry should validate against the schema in which it resides, but that isn't strictly required. There is no need to duplicate the `default` value in the `examples` array, since `default` will be treated as another example.

New in draft 7 The boolean keywords `readOnly` and `writeOnly` are typically used in an API context. `readOnly` indicates that a value should not be modified. It could be used to indicate that a `PUT` request that changes a value would result in a `400 Bad Request` response. `writeOnly` indicates that a value may be set, but will remain hidden. In could be used to indicate you can set a value with a `PUT` request, but it would not be included when retrieving that record with a `GET` request.

New in draft 2019-09 The `deprecated` keyword is a boolean that indicates that the instance value the keyword applies to should not be used and may be removed in the future.

{ json schema }

```
{
  "title": "Match anything",
  "description": "This is a schema that matches anything.",
  "default": "Default value",
  "examples": [
    "Anything",
    4035
  ],
  "deprecated": true,
  "readOnly": true,
  "writeOnly": false
}
```

4.9.2 Comments

New in draft 7 `$comment`

The `$comment` keyword is strictly intended for adding comments to a schema. Its value must always be a string. Unlike the annotations `title`, `description`, and `examples`, JSON schema implementations aren't allowed to attach any meaning or behavior to it whatsoever, and may even strip them at any time. Therefore, they are useful for leaving notes to future editors of a JSON schema, but should not be used to communicate to users of the schema.

4.9.3 Enumerated values

The `enum` keyword is used to restrict a value to a fixed set of values. It must be an array with at least one element, where each element is unique.

The following is an example for validating street light colors:

{ json schema }

```
{
  "enum": ["red", "amber", "green"]
}
```

✓

```
"red"
```

✗

```
"blue"
```

You can use `enum` even without a type, to accept values of different types. Let's extend the example to use `null` to indicate "off", and also add 42, just for fun.

{ json schema }

```
{
  "enum": ["red", "amber", "green", null, 42]
}
```

✓

```
"red"
```

✓

```
null
```

✓

```
42
```

✗

```
0
```

4.9.4 Constant values

New in draft 6

The `const` keyword is used to restrict a value to a single value.

For example, if you only support shipping to the United States for export reasons:

{ json schema }

```
{
  "properties": {
    "country": {
      "const": "United States of America"
    }
  }
}
```

✓

```
{ "country": "United States of America" }
```

✗

```
{ "country": "Canada" }
```

4.10 Media: string-encoding non-JSON data

New in draft 7

JSON schema has a set of keywords to describe and optionally validate non-JSON data stored inside JSON strings. Since it would be difficult to write validators for many media types, JSON schema validators are not required to validate the contents of JSON strings based on these keywords. However, these keywords are still useful for an application that consumes validated JSON.

4.10.1 contentMediaType

The `contentMediaType` keyword specifies the MIME type of the contents of a string, as described in RFC 2046. There is a list of MIME types officially registered by the IANA, but the set of types supported will be application and operating system dependent. Mozilla Developer Network also maintains a shorter list of MIME types that are important for the web

4.10.2 contentEncoding

The `contentEncoding` keyword specifies the encoding used to store the contents, as specified in RFC 2054, part 6.1 and RFC 4648.

The acceptable values are `7bit`, `8bit`, `binary`, `quoted-printable`, `base16`, `base32`, and `base64`. If not specified, the encoding is the same as the containing JSON document.

Without getting into the low-level details of each of these encodings, there are really only two options useful for modern usage:

- If the content is encoded in the same encoding as the enclosing JSON document (which for practical purposes, is almost always UTF-8), leave `contentEncoding` unspecified, and include the content in a string as-is. This includes text-based content types, such as `text/html` or `application/xml`.
- If the content is binary data, set `contentEncoding` to `base64` and encode the contents using Base64. This would include many image types, such as `image/png` or audio types, such as `audio/mpeg`.

4.10.3 contentSchema

New in draft 2019-09

Documentation Coming soon

4.10.4 Examples

The following schema indicates the string contains an HTML document, encoded using the same encoding as the surrounding document:

{ json schema }

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

✓

```
"<!DOCTYPE html><html xmlns=\"http://www.w3.org/1999/xhtml\"><head></head></html>"
```

The following schema indicates that a string contains a PNG image, encoded using Base64:

{ json schema }

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

✓

```
"iVBORw0KGgoAAAANSUhEUgAAABgAAAAYCAYAAADgdz34AAAABmJLR0QA/wD/AP+gvaeTAAAA..."
```

4.11 Schema Composition

JSON Schema includes a few keywords for combining schemas together. Note that this doesn't necessarily mean combining schemas from multiple files or JSON trees, though these facilities help to enable that and are described in *Structuring a complex schema* (page 73). Combining schemas may be as simple as allowing a value to be validated

against multiple criteria at the same time.

These keywords correspond to well known boolean algebra concepts like AND, OR, XOR, and NOT. You can often use these keywords to express complex constraints that can't otherwise be expressed with standard JSON Schema keywords.

The keywords used to combine schemas are:

- *allOf* (page 56): (AND) Must be valid against *all* of the subschemas
- *anyOf* (page 56): (OR) Must be valid against *any* of the subschemas
- *oneOf* (page 57): (XOR) Must be valid against *exactly one* of the subschemas

All of these keywords must be set to an array, where each item is a schema.

In addition, there is:

- *not* (page 58): (NOT) Must *not* be valid against the given schema

4.11.1 allOf

To validate against `allOf`, the given data must be valid against all of the given subschemas.

{ json schema }

```
{
  "allOf": [
    { "type": "string" },
    { "maxLength": 5 }
  ]
}
```

✓

```
"short"
```

✗

```
"too long"
```

Note: *allOf* (page 56) can not be used to "extend" a schema to add more details to it in the sense of object-oriented inheritance. Instances must independently be valid against "all of" the schemas in the `allOf`. See the section on *Extending Closed Schemas* (page 34) for more information.

4.11.2 anyOf

To validate against `anyOf`, the given data must be valid against any (one or more) of the given subschemas.

{ json schema }

```
{
  "anyOf": [
    { "type": "string", "maxLength": 5 },
    { "type": "number", "minimum": 0 }
  ]
}
```

✓

```
"short"
```

✗

```
"too long"
```

✓

```
12
```

✗

```
-5
```

4.11.3 oneOf

To validate against `oneOf`, the given data must be valid against exactly one of the given subschemas.

{ json schema }

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}
```

✓

```
10
```

✓

```
9
```

Not a multiple of either 5 or 3.

✗

```
2
```

Multiple of *both* 5 and 3 is rejected.

✗

```
15
```

4.11.4 not

The `not` keyword declares that an instance validates if it doesn't validate against the given subschema.

For example, the following schema validates against anything that is not a string:

{ json schema }

```
{ "not": { "type": "string" } }
```

✓

```
42
```

✓

```
{ "key": "value" }
```

✗

```
"I am a string"
```

4.11.5 Properties of Schema Composition

Illogical Schemas

Note that it's quite easy to create schemas that are logical impossibilities with these keywords. The following example creates a schema that won't validate against anything (since something may not be both a string and a number at the same time):

{ json schema }

```
{
  "allOf": [
    { "type": "string" },
    { "type": "number" }
  ]
}
```

✗

```
"No way"
```

✗

```
-1
```

Factoring Schemas

Note that it's possible to "factor" out the common parts of the subschemas. The following two schemas are equivalent.

{ json schema }

```
{
  "oneOf": [
    { "type": "number", "multipleOf": 5 },
    { "type": "number", "multipleOf": 3 }
  ]
}
```

{ json schema }

```
{
   "type": "number",
   "oneOf": [
     { "multipleOf": 5 },
     { "multipleOf": 3 }
   ]
 }
```

4.12 Applying Subschemas Conditionally

4.12.1 dependentRequired

The `dependentRequired` keyword conditionally requires that certain properties must be present if a given property is present in an object. For example, suppose we have a schema representing a customer. If you have their credit card number, you also want to ensure you have a billing address. If you don't have their credit card number, a billing address would not be required. We represent this dependency of one property on another using the `dependentRequired` keyword. The value of the `dependentRequired` keyword is an object. Each entry in the object maps from the name of a property, *p*, to an array of strings listing properties that are required if *p* is present.

In the following example, whenever a `credit_card` property is provided, a `billing_address` property must also be present:

{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },

  "required": ["name"],

  "dependentRequired": {
    "credit_card": ["billing_address"]
  }
}
```

✓

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

This instance has a `credit_card`, but it's missing a `billing_address`.

✗

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This is okay, since we have neither a `credit_card`, or a `billing_address`.

✓

```
{
  "name": "John Doe"
}
```

Note that dependencies are not bidirectional. It's okay to have a billing address without a credit card number.

✓

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

To fix the last issue above (that dependencies are not bidirectional), you can, of course, define the bidirectional dependencies explicitly:

{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" },
    "billing_address": { "type": "string" }
  },

  "required": ["name"],

  "dependentRequired": {
    "credit_card": ["billing_address"],
    "billing_address": ["credit_card"]
  }
}
```

This instance has a `credit_card`, but it's missing a `billing_address`.

✗

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This has a `billing_address`, but is missing a `credit_card`.

✗

```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

Draft 4-7

Previously to Draft 2019-09, `dependentRequired` and `dependentSchemas` were one keyword called `dependencies`. If the dependency value was an array, it would behave like `dependentRequired` and if the dependency value was a schema, it would behave like `dependentSchemas`.

4.12.2 dependentSchemas

The `dependentSchemas` keyword conditionally applies a subschema when a given property is present. This schema is applied in the same way *allOf* (page 56) applies schemas. Nothing is merged or extended. Both schemas apply independently.

For example, here is another way to write the above:

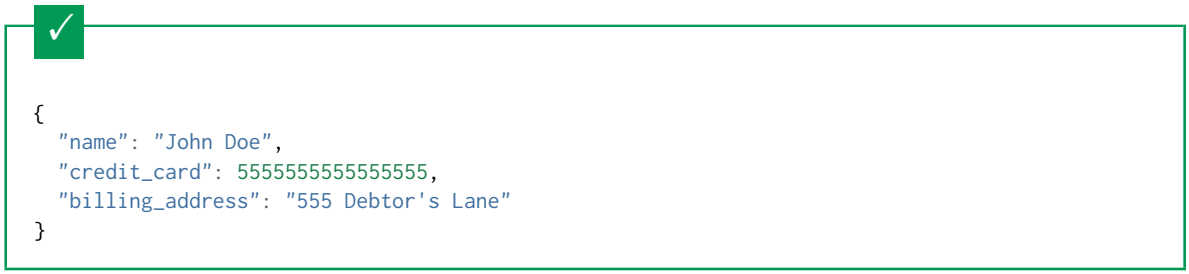
{ json schema }

```
{
  "type": "object",

  "properties": {
    "name": { "type": "string" },
    "credit_card": { "type": "number" }
  },

  "required": ["name"],

  "dependentSchemas": {
    "credit_card": {
      "properties": {
        "billing_address": { "type": "string" }
      },
      "required": ["billing_address"]
    }
  }
}
```

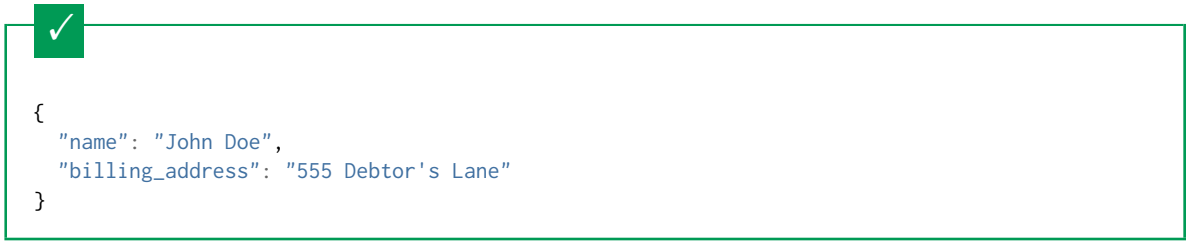


```
{
  "name": "John Doe",
  "credit_card": 5555555555555555,
  "billing_address": "555 Debtor's Lane"
}
```

This instance has a `credit_card`, but it's missing a `billing_address`:

```
{
  "name": "John Doe",
  "credit_card": 5555555555555555
}
```

This has a `billing_address`, but is missing a `credit_card`. This passes, because here `billing_address` just looks like an additional property:



```
{
  "name": "John Doe",
  "billing_address": "555 Debtor's Lane"
}
```

Draft 4-7

Previously to Draft 2019-09, `dependentRequired` and `dependentSchemas` were one keyword called `dependencies`. If the dependency value was an array, it would behave like `dependentRequired` and if the dependency value was a schema, it would behave like `dependentSchemas`.

4.12.3 If-Then-Else

New in draft 7 The `if`, `then` and `else` keywords allow the application of a subschema based on the outcome of another schema, much like the `if`/`then`/`else` constructs you've probably seen in traditional programming languages.

If `if` is valid, `then` must also be valid (and `else` is ignored.) If `if` is invalid, `else` must also be valid (and `then` is ignored).

If `then` or `else` is not defined, `if` behaves as if they have a value of `true`.

If `then` and/or `else` appear in a schema without `if`, `then` and `else` are ignored.

We can put this in the form of a truth table, showing the combinations of when `if`, `then`, and `else` are valid and the resulting validity of the entire schema:

if	then	else	whole schema
T	T	n/a	T
T	F	n/a	F
F	n/a	T	T
F	n/a	F	F
n/a	n/a	n/a	T

For example, let's say you wanted to write a schema to handle addresses in the United States and Canada. These countries have different postal code formats, and we want to select which format to validate against based on the country. If the address is in the United States, the `postal_code` field is a "zipcode": five numeric digits followed by an optional four digit suffix. If the address is in Canada, the `postal_code` field is a six digit alphanumeric string where letters and numbers alternate.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "street_address": {
      "type": "string"
    },
    "country": {
      "default": "United States of America",
      "enum": ["United States of America", "Canada"]
    }
  },
  "if": {
    "properties": { "country": { "const": "United States of America" } }
  },
  "then": {
    "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
  },
  "else": {
    "properties": { "postal_code": { "pattern": "[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" } }
  }
}
```

✓

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "country": "United States of America",
  "postal_code": "20500"
}
```

✓

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "postal_code": "20500"
}
```

✓

```
{
  "street_address": "24 Sussex Drive",
  "country": "Canada",
  "postal_code": "K1M 1M4"
}
```

✗

```
{
  "street_address": "24 Sussex Drive",
  "country": "Canada",
  "postal_code": "10000"
}
```

✗

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "postal_code": "K1M 1M4"
}
```

Note: In this example, "country" is not a required property. Because the "if" schema also doesn't require the "country" property, it will pass and the "then" schema will apply. Therefore, if the "country" property is not defined, the default behavior is to validate "postal_code" as a USA postal code. The "default" keyword doesn't have an effect, but is nice to include for readers of the schema to more easily recognize the default behavior.

Unfortunately, this approach above doesn't scale to more than two countries. You can, however, wrap pairs of `if` and `then` inside an `allOf` to create something that would scale. In this example, we'll use United States and Canadian postal codes, but also add Netherlands postal codes, which are 4 digits followed by two letters. It's left as an exercise to the reader to expand this to the remaining postal codes of the world.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "street_address": {
      "type": "string"
    },
    "country": {
      "default": "United States of America",
      "enum": ["United States of America", "Canada", "Netherlands"]
    }
  },
  "allOf": [
    {
      "if": {
        "properties": { "country": { "const": "United States of America" } }
      },
      "then": {
        "properties": { "postal_code": { "pattern": "[0-9]{5}(-[0-9]{4})?" } }
      }
    },
    {
      "if": {
        "properties": { "country": { "const": "Canada" } },
        "required": ["country"]
      },
      "then": {
        "properties": { "postal_code": { "pattern": "[A-Z][0-9][A-Z] [0-9][A-Z][0-9]" } }
      }
    },
    {
      "if": {
        "properties": { "country": { "const": "Netherlands" } },
        "required": ["country"]
      },
      "then": {
        "properties": { "postal_code": { "pattern": "[0-9]{4} [A-Z]{2}" } }
      }
    }
  ]
}
```

✓

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "country": "United States of America",
  "postal_code": "20500"
}
```

✓

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "postal_code": "20500"
}
```

✓

```
{
  "street_address": "24 Sussex Drive",
  "country": "Canada",
  "postal_code": "K1M 1M4"
}
```

✓

```
{
  "street_address": "Adriaan Goekooplaan",
  "country": "Netherlands",
  "postal_code": "2517 JX"
}
```

✗

```
{
  "street_address": "24 Sussex Drive",
  "country": "Canada",
  "postal_code": "10000"
}
```

✗

```
{
  "street_address": "1600 Pennsylvania Avenue NW",
  "postal_code": "K1M 1M4"
}
```

Note: The "required" keyword is necessary in the "if" schemas or they would all apply if the "country" is not defined. Leaving "required" off of the "United States of America" "if" schema makes it effectively the default if no "country" is defined.

Note: Even if "country" was a required field, it's still recommended to have the "required" keyword in each "if" schema. The validation result will be the same because "required" will fail, but not including it will add noise to error results because it will validate the "postal_code" against all three of the "then" schemas leading to irrelevant errors.

4.12.4 Implication

Before Draft 7, you can express an "if-then" conditional using the *Schema Composition* (page 55) keywords and a boolean algebra concept called "implication". `A -> B` (pronounced, A implies B) means that if A is true, then B must also be true. It can be expressed as `!A || B` which can be expressed as a JSON Schema.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "restaurantType": { "enum": ["fast-food", "sit-down"] },
    "total": { "type": "number" },
    "tip": { "type": "number" }
  },
  "anyOf": [
    {
      "not": {
        "properties": { "restaurantType": { "const": "sit-down" } },
        "required": ["restaurantType"]
      }
    },
    { "required": ["tip"] }
  ]
}
```

✓

```
{
  "restaurantType": "sit-down",
  "total": 16.99,
  "tip": 3.4
}
```

✗

```
{
  "restaurantType": "sit-down",
  "total": 16.99
}
```

✓

```
{
  "restaurantType": "fast-food",
  "total": 6.99
}
```

✓

```
{ "total": 5.25 }
```

Variations of implication can be used to express the same things you can express with the `if/then/else` keywords. `if/then` can be expressed as `A -> B`, `if/else` can be expressed as `!A -> B`, and `if/then/else` can be expressed as `A -> B AND !A -> C`.

Note: Since this pattern is not very intuitive, it's recommended to put your conditionals in `$defs` with a descriptive name and `$ref` it into your schema with `"allOf": [{ "$ref": "#/$defs/sit-down-restaurant-implies-tip-is-required" }]`.

4.13 Declaring a Dialect

A version of JSON Schema is called a dialect. A dialect represents the set of keywords and semantics that can be used to evaluate a schema. Each JSON Schema release is a new dialect of JSON Schema. JSON Schema provides a way for you to declare which dialect a schema conforms to and provides ways to describe your own custom dialects.

4.13.1 $schema

The `$schema` keyword is used to declare which dialect of JSON Schema the schema was written for. The value of the `$schema` keyword is also the identifier for a schema that can be used to verify that the schema is valid according to the dialect `$schema` identifies. A schema that describes another schema is called a "meta-schema".

`$schema` applies to the entire document and must be at the root level. It does not apply to externally referenced (`$ref`, `$dynamicRef`) documents. Those schemas need to declare their own `$schema`.

If `$schema` is not used, an implementation might allow you to specify a value externally or it might make assumptions about which specification version should be used to evaluate the schema. It's recommended that all JSON Schemas have a `$schema` keyword to communicate to readers and tooling which specification version is intended. Therefore most of the time, you'll want this at the root of your schema:

```
"$schema": "https://json-schema.org/draft/2020-12/schema"
```

Draft 4

The identifier for Draft 4 is `http://json-schema.org/draft-04/schema#`.

Draft 4 defined a value for `$schema` without a specific dialect (`http://json-schema.org/schema#`) which meant, use the latest dialect. This has since been deprecated and should no longer be used.

You might come across references to Draft 5. There is no Draft 5 release of JSON Schema. Draft 5 refers to a no-change revision of the Draft 4 release. It does not add, remove, or change any functionality. It only updates references, makes clarifications, and fixes bugs. Draft 5 describes the Draft 4 release. If you came here looking for information about Draft 5, you'll find it under Draft 4. We no longer use the "draft" terminology to refer to patch releases to avoid this confusion.

Draft 6

The identifier for Draft 6 is `http://json-schema.org/draft-06/schema#`.

Draft 7

The identifier for Draft 7 is `http://json-schema.org/draft-07/schema#`.

Draft 2019-09

The identifier for Draft 2019-09 is `https://json-schema.org/draft/2019-09/schema`.

4.13.2 Vocabularies

New in draft 2019-09

Documentation Coming Soon

Draft 4-7

Before the introduction of Vocabularies, you could still extend JSON Schema with your custom keywords but the process was much less formalized. The first thing you'll need is a meta-schema that includes your custom keywords. The best way to do this is to make a copy of the meta-schema for the version you want to extend and make your changes to your copy. You will need to choose a custom URI to identify your custom version. This URI must not be one of the URIs used to identify official JSON Schema specification drafts and should probably include a domain name you own. You can use this URI with the `$schema` keyword to declare that your schemas use your custom version.

Note: Not all implementations support custom meta-schemas and custom keyword implementations.

Guidelines

One of the strengths of JSON Schema is that it can be written in JSON and used in a variety of environments. For example, it can be used for both front-end and back-end HTML Form validation. The problem with using custom vocabularies is that every environment where you want to use your schemas needs to understand how to evaluate your vocabulary's keywords. Meta-schemas can be used to ensure that schemas are written correctly, but each implementation will need custom code to understand how to evaluate the vocabulary's keywords.

Meta-data keywords are the most interoperable because they don't affect validation. For example, you could add a `units` keyword. This will always work as expecting with an compliant validator.

{ json schema }

```
{
  "type": "number",
  "units": "kg"
}
```

✓

```
42
```

✗

```
"42"
```

The next best candidates for custom keywords are keywords that don't apply other schemas and don't modify the behavior of existing keywords. An `isEven` keyword is an example. In contexts where some validation is better than no validation such as validating an HTML Form in the browser, this schema will perform as well as can be expected. Full validation would still be required and should use a validator that understands the custom keyword.

{ json schema }

```
{
  "type": "integer",
  "isEven": true
}
```

✓

```
2
```

This passes because the validator doesn't understand `isEven`

✓

```
3
```

The schema isn't completely impaired because it doesn't understand `isEven`

✗

```
"3"
```

The least interoperable type of custom keyword is one that applies other schemas or modifies the behavior of existing keywords. An example would be something like `requiredProperties` that declares properties and makes them required. This example shows how the schema becomes almost completely useless when evaluated with a validator that doesn't understand the custom keyword. That doesn't necessarily mean that `requiredProperties` is a bad idea for a keyword, it's just not the right choice if the schema might need to be used in a context that doesn't understand custom keywords.

{ json schema }

```
{
  "type": "object",
  "requiredProperties": {
    "foo": { "type": "string" }
  }
}
```

✓

```
{ "foo": "bar" }
```

This passes because `requiredProperties` is not understood

✓

```
{}
```

This passes because `requiredProperties` is not understood

✓

```
{ "foo": 42 }
```

CHAPTER 5

Structuring a complex schema

When writing computer programs of even moderate complexity, it's commonly accepted that "structuring" the program into reusable functions is better than copying-and-pasting duplicate bits of code everywhere they are used. Likewise in JSON Schema, for anything but the most trivial schema, it's really useful to structure the schema into parts that can be reused in a number of places. This chapter will present the tools available for reusing and structuring schemas as well as some practical examples that use those tools.

5.1 Schema Identification

Like any other code, schemas are easier to maintain if they can be broken down into logical units that reference each other as necessary. In order to reference a schema, we need a way to identify a schema. Schema documents are identified by non-relative URIs.

Schema documents are not required to have an identifier, but you will need one if you want to reference one schema from another. In this documentation, we will refer to schemas with no identifier as "anonymous schemas".

In the following sections we will see how the "identifier" for a schema is determined.

Note: URI terminology can sometimes be unintuitive. In this document, the following definitions are used.

- **URI** [1] or **non-relative URI**: A full URI containing a scheme (`https`). It may contain a URI fragment (`#foo`). Sometimes this document will use "non-relative URI" to make it extra clear that relative URIs are not allowed.
- **relative reference** [2]: A partial URI that does not contain a scheme (`https`). It may contain a fragment (`#foo`).
- **URI-reference** [3]: A relative reference or non-relative URI. It may contain a URI fragment (`#foo`).
- **absolute URI** [4] A full URI containing a scheme (`https`) but not a URI fragment (`#foo`).

Note: Even though schemas are identified by URIs, those identifiers are not necessarily network-addressable. They are just identifiers. Generally, implementations don't make HTTP requests (`https://`) or read from the file system (`file://`) to fetch schemas. Instead, they provide a way to load schemas into an internal schema database. When a schema is referenced by it's URI identifier, the schema is retrieved from the internal schema database.

5.2 Base URI

Using non-relative URIs can be cumbersome, so any URIs used in JSON Schema can be URI-references that resolve against the schema's base URI resulting in a non-relative URI. This section describes how a schema's base URI is determined.

Note: Base URI determination and relative reference resolution is defined by RFC-3986. If you are familiar with how this works in HTML, this section should feel very familiar.

5.2.1 Retrieval URI

The URI used to fetch a schema is known as the "retrieval URI". It's often possible to pass an anonymous schema to an implementation in which case that schema would have no retrieval URI.

Let's assume a schema is referenced using the URI `https://example.com/schemas/address` and the following schema is retrieved.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

The base URI for this schema is the same as the retrieval URI, `https://example.com/schemas/address`.

5.2.2 $id

You can set the base URI by using the `$id` keyword at the root of the schema. The value of `$id` is a URI-reference without a fragment that resolves against the *Retrieval URI* (page 74). The resulting URI is the base URI for the schema.

Draft 4

In Draft 4, `$id` is just `id` (without the dollar sign).

Draft 4-7

In Draft 4-7 it was allowed to have fragments in an `$id` (or `id` in Draft 4). However, the behavior when setting a base URI that contains a URI fragment is undefined and should not be used because implementations may treat them differently.

Note: This is analogous to the `<base>` tag in HTML.

Note: When the `$id` keyword appears in a subschema, it means something slightly different. See the *Bundling* (page 80) section for more.

Let's assume the URIs `https://example.com/schema/address` and `https://example.com/schema/billing-address` both identify the following schema.

{ json schema }

```
{
  "$id": "/schemas/address",

  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

No matter which of the two URIs is used to retrieve this schema, the base URI will be `https://example.com/schemas/address`, which is the result of the `$id` URI-reference resolving against the *Retrieval URI* (page 74).

However, using a relative reference when setting a base URI can be problematic. For example, we couldn't use this schema as an anonymous schema because there would be no *Retrieval URI* (page 74) and you can't resolve a relative reference against nothing. For this and other reasons, it's recommended that you always use an absolute URI when declaring a base URI with `$id`.

The base URI of the following schema will always be `https://example.com/schemas/address` no matter what the *Retrieval URI* (page 74) was or if it's used as an anonymous schema.

{ json schema }

```
{
  "$id": "https://example.com/schemas/address",

  "type": "object",
  "properties": {
    "street_address": { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

5.2.3 JSON Pointer

In addition to identifying a schema document, you can also identify subschemas. The most common way to do that is to use a JSON Pointer in the URI fragment that points to the subschema.

A JSON Pointer describes a slash-separated path to traverse the keys in the objects in the document. Therefore, `/properties/street_address` means:

1) find the value of the key `properties`
2) within that object, find the value of the key `street_address`

The URI `https://example.com/schemas/address#/properties/street_address` identifies the highlighted subschema in the following schema.

{ json schema }

```
{
  "$id": "https://example.com/schemas/address",

  "type": "object",
  "properties": {
    "street_address":
      { "type": "string" },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

5.2.4 $anchor

A less common way to identify a subschema is to create a named anchor in the schema using the `$anchor` keyword and using that name in the URI fragment. Anchors must start with a letter followed by any number of letters, digits, `-`, `_`, `:`, or `.`.

Draft 4

In Draft 4, you declare an anchor the same way you do in Draft 6-7 except that `$id` is just `id` (without the dollar sign).

Draft 6-7

In Draft 6-7, a named anchor is defined using an `$id` that contains only a URI fragment. The value of the URI fragment is the name of the anchor.

JSON Schema doesn't define how `$id` should be interpreted when it contains both fragment and non-fragment URI parts. Therefore, when setting a named anchor, you should not use non-fragment URI parts in the URI-reference.

Note: If a named anchor is defined that doesn't follow these naming rules, then behavior is undefined. Your anchors might work in some implementation, but not others.

The URI `https://example.com/schemas/address#street_address` identifies the subschema on the highlighted part of the following schema.

{ json schema }

```
{
  "$id": "https://example.com/schemas/address",

  "type": "object",
  "properties": {
    "street_address":
      {
        "$anchor": "street_address",
        "type": "string"
      },
    "city": { "type": "string" },
    "state": { "type": "string" }
  },
  "required": ["street_address", "city", "state"]
}
```

5.3 $ref

A schema can reference another schema using the `$ref` keyword. The value of `$ref` is a URI-reference that is resolved against the schema's *Base URI* (page 74). When evaluating a `$ref`, an implementation uses the resolved identifier to retrieve the referenced schema and applies that schema to the instance.

Draft 4-7

In Draft 4-7, `$ref` behaves a little differently. When an object contains a `$ref` property, the object is considered a reference, not a schema. Therefore, any other properties you put in that object will not be treated as JSON Schema keywords and will be ignored by the validator. `$ref` can only be used where a schema is expected.

For this example, let's say we want to define a customer record, where each customer may have both a shipping and

a billing address. Addresses are always the same—they have a street address, city and state—so we don't want to duplicate that part of the schema everywhere we want to store an address. Not only would that make the schema more verbose, but it makes updating it in the future more difficult. If our imaginary company were to start doing international business in the future and we wanted to add a country field to all the addresses, it would be better to do this in a single place rather than everywhere that addresses are used.

{ json schema }

```
{
  "$id": "https://example.com/schemas/customer",

  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "shipping_address": { "$ref": "/schemas/address" },
    "billing_address": { "$ref": "/schemas/address" }
  },
  "required": ["first_name", "last_name", "shipping_address", "billing_address"]
}
```

The URI-references in `$ref` resolve against the schema's *Base URI* (page 74) (`https://example.com/schemas/customer`) which results in `https://example.com/schemas/address`. The implementation retrieves that schema and uses it to evaluate the "shipping_address" and "billing_address" properties.

Note: When using `$ref` in an anonymous schema, relative references may not be resolvable. Let's assume this example is used as an anonymous schema.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "shipping_address": { "$ref": "https://example.com/schemas/address" },
    "billing_address": { "$ref": "/schemas/address" }
  },
  "required": ["first_name", "last_name", "shipping_address", "billing_address"]
}
```

The `$ref` at `/properties/shipping_address` can resolve just fine without a non-relative base URI to resolve against, but the `$ref` at `/properties/billing_address` can't resolve to a non-relative URI and therefore can't can be used to retrieve the address schema.

5.4 $defs

Sometimes we have small subschemas that are only intended for use in the current schema and it doesn't make sense to define them as separate schemas. Although we can identify any subschema using JSON Pointers or named anchors, the `$defs` keyword gives us a standardized place to keep subschemas intended for reuse in the current schema document.

Let's extend the previous customer schema example to use a common schema for the name properties. It doesn't make sense to define a new schema for this and it will only be used in this schema, so it's a good candidate for using `$defs`.

{ json schema }

```
{
  "$id": "https://example.com/schemas/customer",

  "type": "object",
  "properties": {
    "first_name": { "$ref": "#/$defs/name" },
    "last_name": { "$ref": "#/$defs/name" },
    "shipping_address": { "$ref": "/schemas/address" },
    "billing_address": { "$ref": "/schemas/address" }
  },
  "required": ["first_name", "last_name", "shipping_address", "billing_address"],

  "$defs": {
    "name": { "type": "string" }
  }
}
```

`$ref` isn't just good for avoiding duplication. It can also be useful for writing schemas that are easier to read and maintain. Complex parts of the schema can be defined in `$defs` with descriptive names and referenced where it's needed. This allows readers of the schema to more quickly and easily understand the schema at a high level before diving into the more complex parts.

Note: It's possible to reference an external subschema, but generally you want to limit a `$ref` to referencing either an external schema or an internal subschema defined in `$defs`.

5.5 Recursion

The `$ref` keyword may be used to create recursive schemas that refer to themselves. For example, you might have a `person` schema that has an array of `children`, each of which are also `person` instances.

{ json schema }

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "children": {
      "type": "array",
      "items": { "$ref": "#" }
    }
  }
}
```

A snippet of the British royal family tree

✓

```
{
  "name": "Elizabeth",
  "children": [
    {
      "name": "Charles",
      "children": [
        {
          "name": "William",
          "children": [
            { "name": "George" },
            { "name": "Charlotte" }
          ]
        },
        {
          "name": "Harry"
        }
      ]
    }
  ]
}
```

Above, we created a schema that refers to itself, effectively creating a "loop" in the validator, which is both allowed and useful. Note, however, that a `$ref` referring to another `$ref` could cause an infinite loop in the resolver, and is explicitly disallowed.

{ json schema }

```
{
  "$defs": {
    "alice": { "$ref": "#/$defs/bob" },
    "bob": { "$ref": "#/$defs/alice" }
  }
}
```

5.6 Extending Recursive Schemas

New in draft 2019-09

Documentation Coming Soon

5.7 Bundling

Working with multiple schema documents is convenient for development, but it's often more convenient for distribution to bundle all of your schemas into a single schema document. This can be done using the `$id` keyword in a subschema. When `$id` is used in a subschema, it indicates an embedded schema. The identifier for the embedded schema is the value of `$id` resolved against the *Base URI* (page 74) of the schema it appears in. A schema document that includes embedded schemas is called a Compound Schema Document. Each schema with an `$id` in a Compound Schema Document is called a Schema Resource.

Draft 4

In Draft 4, `$id` is just `id` (without the dollar sign).

Draft 4-7

In Draft 4-7, an `$id` in a subschema did not indicate an embedded schema. Instead it was simply a base URI change in a single schema document.

Note: This is analogous to the `<iframe>` tag in HTML.

Note: It is unusual to use embedded schemas when developing schemas. It's generally best not to use this feature explicitly and use schema bundling tools to construct bundled schemas if such a thing is needed.

This example shows the customer schema example and the address schema example bundled into a Compound Schema Document.

{ json schema }

```
{
  "$id": "https://example.com/schemas/customer",
  "$schema": "https://json-schema.org/draft/2020-12/schema",

  "type": "object",
  "properties": {
    "first_name": { "type": "string" },
    "last_name": { "type": "string" },
    "shipping_address": { "$ref": "/schemas/address" },
    "billing_address": { "$ref": "/schemas/address" }
  },
  "required": ["first_name", "last_name", "shipping_address", "billing_address"],

  "$defs": {
    "address": {
      "$id": "/schemas/address",
      "$schema": "http://json-schema.org/draft-07/schema#",

      "type": "object",
      "properties": {
        "street_address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "$ref": "#/definitions/state" }
      },
      "required": ["street_address", "city", "state"],

      "definitions": {
        "state": { "enum": ["CA", "NY", "... etc ..."] }
      }
    }
  }
}
```

All references in a Compound Schema Document need to be the same whether the Schema Resources are bundled or not. Notice that the `$ref` keywords from the customer schema have not changed. The only difference is that the address schema is now defined at `/$defs/address` instead of a separate schema document. You couldn't use `#/$defs/address` to reference the address schema because if you unbundled the schema, that reference would no longer point to the address schema.

Draft 4-7

In Draft 4-7, both of these URIs are valid because a subschema `$id` only represented a base URI change, not an embedded schema. However, even though it's allowed, it's still highly recommended that JSON Pointers don't cross a schema with a base URI change.

You should also see that `"$ref": "#/definitions/state"` resolves to the `definitions` keyword in the address schema rather than the one at the top level schema like it would if the embedded schema wasn't used.

Each Schema Resource is evaluated independently and may use different JSON Schema dialects. The example above has the address Schema Resource using Draft 7 while the customer Schema Resource uses Draft 2020-12. If no `$schema` is declared in an embedded schema, it defaults to using the dialect of the parent schema.

Draft 4-7

In Draft 4-7, a subschema `$id` is just a base URI change and not considered an independent Schema Resource. Because `$schema` is only allowed at the root of a Schema Resource, all schemas bundled using subschema `$id` must use the same dialect.

CHAPTER 6

Acknowledgments

Michael Droettboom wishes to thank the following contributors:

- Alexander Kjeldaas
- Alexander Lang
- Anders D. Johnson
- Armand Abric
- Ben Hutton
- Brandon Wright
- Brent Tubbs
- Chris Carpenter
- Christopher Mark Gore
- David Branner
- David Michael Karr
- David Worth
- E. M. Bray
- Fenhl
- forevermatt
- goldaxe
- Henry Andrews
- Hervé
- Hongwei
- Jesse Claven
- Koen Rouwhorst

- Mike Kobit
- Oliver Kurmis
- Sam Blackman
- Vincent Jacques

Index

U

W