https://spacetelescope.github.io/understanding-json-schema/

Since JSON Schema is itself JSON, it’s not always easy to tell when something is JSON Schema or just an arbitrary chunk of JSON. The $schema keyword is used to declare that something is JSON Schema. 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.

{ "$schema": "http://json-schema.org/schema#" }

**Declaring a unique identifier**[**¶**](https://spacetelescope.github.io/understanding-json-schema/basics.html#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" }

## 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

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:

{

"type": "string",

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

}

"555-1212"

## integer

The integer type is used for integral numbers.

he precise treatment of the “integer” type may depend on the implementation of your JSON Schema validator. JavaScript (and thus also JSON) does not have distinct types for integers and floating-point values. Therefore, JSON Schema can not use type alone to distinguish between integers and non-integers. The JSON Schema specification recommends, but does not require, that validators use the mathematical value to determine whether a number is an integer, and not the type alone. Therefore, there is some disagreement between validators on this point. For example, a JavaScript-based may accept 1.0 as an integer, whereas the Python-based [jsonschema](https://pypi.python.org/pypi/jsonschema) does not.

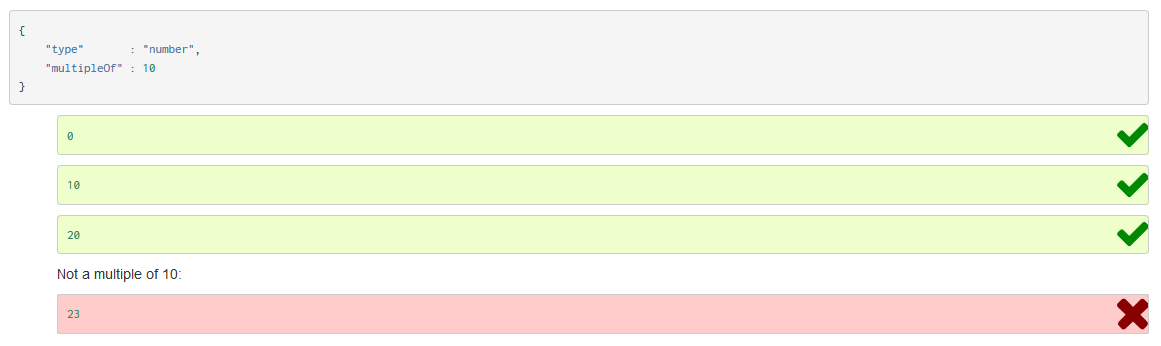
Clever use of the multipleOf keyword (see [Multiples](https://spacetelescope.github.io/understanding-json-schema/reference/numeric.html#multiples)) can be used to get around this discrepancy. For example, the following likely has the same behavior on all JSON Schema implementations:

{ "type": "number", "multipleOf": 1.0 }

42 --right

42.0 --right

3.14156926 --wrong



## Range

Ranges of numbers are specified using a combination of the minimum, maximum, exclusiveMinimum and exclusiveMaximum keywords.

* minimum specifies a minimum numeric value.
* exclusiveMinimum is a boolean. When true, it indicates that the range excludes the minimum value, i.e., *x*>min

. When false (or not included), it indicates that the range includes the minimum value, i.e., *x*≥min

 .

 maximum specifies a maximum numeric value.

 exclusiveMaximum is a boolean. When true, it indicates that the range excludes the maximum value, i.e., *x*<max. When false (or not included), it indicates that the range includes the maximum value, i.e., *x*≤max

* .

{

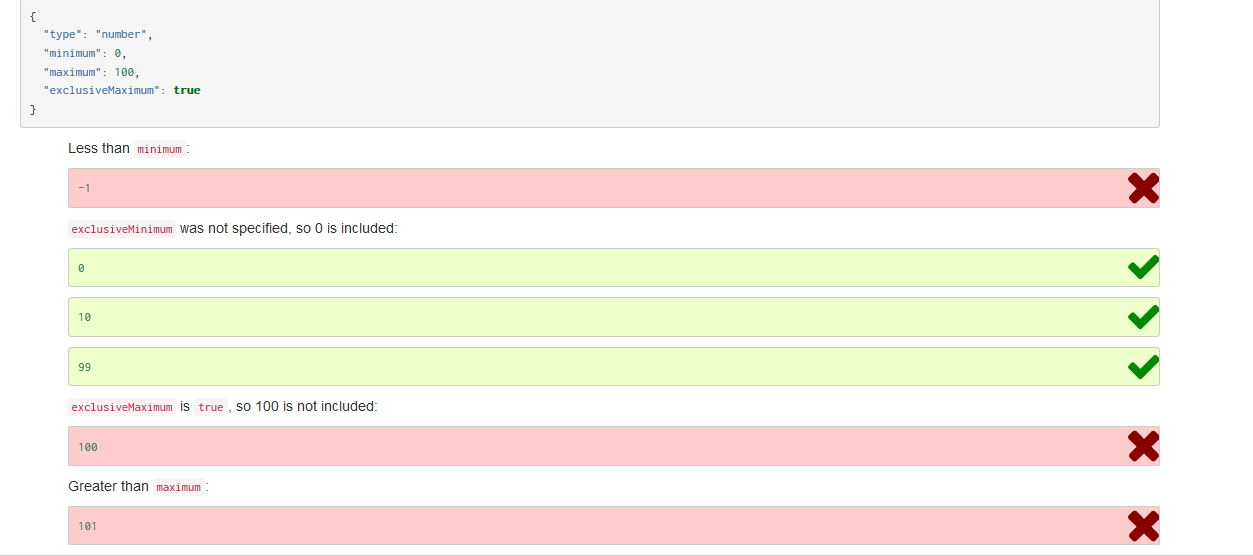
"type": "number",

"minimum": 0,

"maximum": 100,

"exclusiveMaximum": true

}



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.



## Items

By default, the elements of the array may be anything at all. However, it’s often useful to validate the items of the array against some schema as well. This is done using the items and additionalItems keywords.

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.

### List validation

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.

Note

When items is a single schema, the additionalItems keyword is meaningless, and it should not be used.

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

{

"type": "array",

"items": {

"type": "number"

}

}

### 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 set the items keyword to 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.

Here’s the example schema:

{

"type": "array",

"items": [

{

"type": "number"

},

{

"type": "string"

},

{

"type": "string",

"enum": ["Street", "Avenue", "Boulevard"]

},

{

"type": "string",

"enum": ["NW", "NE", "SW", "SE"]

}

],

"additionalItems": false

}

The additionalItems keyword controls whether it’s valid to have additional items in the array beyond what is defined in the schema. Here, we’ll reuse the example schema above, but set additionalItems to false, which has the effect of disallowing extra items in the array.

## 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](https://spacetelescope.github.io/understanding-json-schema/reference/array.html#list-validation) or [Tuple validation](https://spacetelescope.github.io/understanding-json-schema/reference/array.html#tuple-validation).

{

"type": "array",

"minItems": 2,

"maxItems": 3

}

## Uniqueness

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

{

"type": "array",

"uniqueItems": true

}

## Metadata

JSON Schema includes a few keywords, title, description and default, that aren’t strictly used for validation, but are used to describe parts of a schema.

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. Neither are required, but they are encouraged for good practice.

The default keyword specifies a default value for an item. JSON processing tools may use this information to provide a default value for a missing key/value pair, though many JSON schema validators simply ignore the default keyword. It should validate against the schema in which it resides, but that isn’t required.

{

"title" : "Match anything",

"description" : "This is a schema that matches anything.",

"default" : "Default value"

}

## 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:

{

"type": "string",

"enum": ["red", "amber", "green"]

}

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.

{

"enum": ["red", "amber", "green", null, 42]

}

# Combining schemas

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](https://spacetelescope.github.io/understanding-json-schema/structuring.html#structuring). Combining schemas may be as simple as allowing a value to be validated against multiple criteria at the same time.

For example, in the following schema, the anyOf keyword is used to say that the given value may be valid against any of the given subschemas. The first subschema requires a string with maximum length 5. The second subschema requires a number with a minimum value of 0. As long as a value validates against either of these schemas, it is considered valid against the entire combined schema.

{

"anyOf": [

{ "type": "string", "maxLength": 5 },

{ "type": "number", "minimum": 0 }

]

}

The keywords used to combine schemas are:

* [allOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#allof): Must be valid against *all* of the subschemas
* [anyOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#anyof): Must be valid against *any* of the subschemas
* [oneOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#oneof): Must be valid against *exactly one* of the subschemas
* [not](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#not): Must *not* be valid against the given schema

It is important to note that the schemas listed in an [allOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#allof), [anyOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#anyof) or [oneOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#oneof) array know nothing of one another. While it might be surprising, [allOf](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#allof) can not be used to “extend” a schema to add more details to it in the sense of object-oriented inheritance. For example, say you had a schema for an address in a efinitions section, and want to extend it to include an address type: 

## oneOf

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

# The $schema keyword

The $schema keyword is used to declare that a JSON fragment is actually a piece of JSON Schema. It also declares which version of the JSON Schema standard that the schema was written against.

It is recommended that all JSON Schemas have a $schema entry, which must be at the root. Therefore most of the time, you’ll want this at the root of your schema:

"$schema": "http://json-schema.org/schema#"

# Regular Expressions

The [pattern](https://spacetelescope.github.io/understanding-json-schema/reference/string.html#pattern) and [Pattern Properties](https://spacetelescope.github.io/understanding-json-schema/reference/object.html#patternproperties) keywords use regular expressions to express constraints. The regular expression syntax used is from JavaScript ([ECMA 262](http://www.ecma-international.org/publications/standards/Ecma-262.htm), 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 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}: 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.

{

"type": "string",

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

}

# 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 some practical examples that use the tools available for reusing and structuring schemas.

So let’s start with the schema that defines an address:

{

"type": "object",

"properties": {

"street\_address": { "type": "string" },

"city": { "type": "string" },

"state": { "type": "string" }

},

"required": ["street\_address", "city", "state"]

}

Since we are going to reuse this schema, it is customary (but not required) to put it in the parent schema under a key called definitions:

{

"definitions": {

"address": {

"type": "object",

"properties": {

"street\_address": { "type": "string" },

"city": { "type": "string" },

"state": { "type": "string" }

},

"required": ["street\_address", "city", "state"]

}

}

}

We can then refer to this schema snippet from elsewhere using the $ref keyword. The easiest way to describe $ref is that it gets logically replaced with the thing that it points to. So, to refer to the above, we would include:

{ "$ref": "#/definitions/address" }

The value of $ref is a string in a format called [JSON Pointer](https://tools.ietf.org/html/rfc6901).

Note

JSON Pointer aims to serve the same purpose as [XPath](http://www.w3.org/TR/xpath/) from the XML world, but it is much simpler.

The pound symbol (#) refers to the current document, and then the slash (/) separated keys thereafter just traverse the keys in the objects in the document. Therefore, in our example "#/definitions/address" means:

1. go to the root of the document
2. find the value of the key "definitions"
3. within that object, find the value of the key "address"

$ref can also be a relative or absolute URI, so if you prefer to include your definitions in separate files, you can also do that. For example:

{ "$ref": "definitions.json#/address" }

would load the address schema from another file residing alongside this one.

Now let’s put this together and use our address schema to create a schema for a customer:

{

"$schema": "http://json-schema.org/draft-04/schema#",

"definitions": {

"address": {

"type": "object",

"properties": {

"street\_address": { "type": "string" },

"city": { "type": "string" },

"state": { "type": "string" }

},

"required": ["street\_address", "city", "state"]

}

},

"type": "object",

"properties": {

"billing\_address": { "$ref": "#/definitions/address" },

"shipping\_address": { "$ref": "#/definitions/address" }

}

}

## The id property

The id property serves two purposes:

* It declares a unique identifier for the schema.
* It declares a base URL against which $ref URLs are resolved.

It is best practice that id is a URL, preferably in a domain that you control. For example, if you own the foo.bar domain, and you had a schema for addresses, you may set its id as follows:

"id": "http://foo.bar/schemas/address.json"

This provides a unique identifier for the schema, as well as, in most cases, indicating where it may be downloaded.

But be aware of the second purpose of the id property: that it declares a base URL for relative $ref URLs elsewhere in the file. For example, if you had:

{ "$ref": "person.json" }

in the same file, a JSON schema validation library would fetch person.json from http://foo.bar/schemas/person.json, even if address.json was loaded from the local filesystem.

## Extending

The power of $ref really shines when it is combined with the combining keywords allOf, anyOf and oneOf (see [Combining schemas](https://spacetelescope.github.io/understanding-json-schema/reference/combining.html#combining)).

Let’s say that for shipping address, we want to know whether the address is a residential or business address, because the shipping method used may depend on that. For the billing address, we don’t want to store that information, because it’s not applicable.

To handle this, we’ll update our definition of shipping address:

"shipping\_address": { "$ref": "#/definitions/address" }

to instead use an allOf keyword entry combining both the core address schema definition and an extra schema snippet for the address type:

"shipping\_address": {

"allOf": [

// Here, we include our "core" address schema...

{ "$ref": "#/definitions/address" },

// ...and then extend it with stuff specific to a shipping

// address

{ "properties": {

"type": { "enum": [ "residential", "business" ] }

},

"required": ["type"]

}

]

}