JSON Schema

Overview

Just like XSD for XML, JSON Schema allows to formally describe the elements which will compose a specific class of JSON objects. JSON Schema is nothing more than a JSON itself.

Basics of JSON Schema

The special rules when coming to JSON Schema are that an empty schema accepts anything, as long as it's valid JSON, and true can also be used in place of the empty object to represent a schema that matches anything, or false for a schema that matches nothing.

Declaring a JSON Schema: the **$schema** keyword is used to declare that something is a JSON Schema.
{ `"$schema": "http://json-schema.org/draft-07/schema#"` } must be one of the direct accessible properties, declaring that your schema was written against a specific version of the JSON Schema (in this specific case `draft-07`).
Every schema will be then identified by the specific value that will be put as id.
For example: `"$id": "pokemonJSONSchema"`

Type-specific keywords: the "type" keyword specifies the data type for an element of the schema. The type keyword may either be a string or an array:

- If it's a string, then it is the name of one of the basic types supplied by JSON Schema ("`string`", "`integer`", "`number`", "`object`", "`array`", "`boolean`", `null`)
- If it is an array, then 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.

For example,

❏ If the schema is set as { `"type": "number"` }, it will either accept 42 or 42.0, but not the string "42"
❏ If the schema is set to { `"type": ["number", "string"]` }, instead, the type can be both a number or a string, thus 42,42.0 and "42" will all be accepted (it still won't accept structured data types like [42, 43, 44]).

Restrictions: for each type, it's possible to specify eventual constraints:

string	• `minLength` • `maxLength` • `pattern` • `format`

The pattern keyword allows to restrict the string to a particular regular expression:

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

This accepts just strings representing telephone number where the prefix is composed of three digits between curve brackets followed by other three digits, a dash and then the last 4 digit (e.g. `"(888)555-1212"`).

The format keyword instead allows for basic semantic validation on certain kinds of string values that are commonly used, like date/time, email addresses, IP addresses and so on:

```
{       "type": "string",
        "format": "ipv6"
}
```

This accepts just strings representing valid ipv6 addresses.

integer **number**	• `multipleOf` • `minimum` • `maximum` • `exclusiveMinimum` • `exclusiveMaximum`

The difference between minimum and `exclusiveMinimum` is that, in the case of `{"type": "number", "exclusiveMinimum": 10}`, `exclusiveMinimum` won't accept the value 10, while `minimum` will.

object	• `properties` • `required` • `property names` • `minProperties` • `maxProperties` • `dependencies`

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 JSON schema used to validate that property. For example:

```
{

        "type": "object",
        "properties": {
                "species": { "type": "string" },
                "dex": { "type": "integer" },
                "type": { "type": "string", "enum": ["GRASS", "WATER", "FIRE"]}
        }
        "required": ["species", "dex"]
}
```

The `required` attribute specifies the properties that must be included within the object.

It is also possible to validate their names against a schema, irrespective of their values, via property names:

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

JSON Schema allows to also set `dependencies` between properties, with the value of the dependencies keyword being an object. Suppose that we have an object that contains the informations about users and we want to always have infos on the billing address of the users who also have a credit card. Then, we can do it like this:

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

	"properties": { ...},
	"dependencies": { "credit_card": ["billing_address"] }
}
```

array	• `list validation` • `tuple validation` • `minItems` • `maxItems` • `uniqueness`

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, each item of the array must be of type “number”:

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

It is also possible to use “contains” instead of “items” to specify that the array must contain at least one item of the specified type. Unlike list validation, tuple validation is useful when the array is a collection of items where each one has a different schema and the ordinal index of each item is meaningful. For example:

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

is going to accept [1600, "Pennsylvania", "Avenue", "NW"], but not [24, "Sussex", "Drive"], since “Drive” is not an accepted enum value. It’s possible to not provide all values.

boolean	Nothing, matches just to “`true`“ or “`false`“. Other values like “`0`” or “`1`” are not accepted by the schema.

null	Only one accepted value: `null`.

Combining schemas

JSON Schema includes a few keywords for combining schemas together. This allows not only the combination of schemas from multiple files or JSON trees, but also allows a value to be validated against multiple criteria at the same time.

There are four keywords used for combining schemas:

- `allOf` - the given data must be valid against all of the given subschemas
- `anyOf` - the given data must be valid against any (one or more) of the given subschemas
- `oneOf` - the given data must be valid against exactly one of the given subschemas
- `not` - declares that an instance is valid if it doesn’t validate against any of the given subschemas

An example of schema combination is:

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

In this case the object is accepted if it is an array of string and/or number, no matter how they are interleaved and without requiring both.

References

If you’re interested in more examples take a look at the JSON Schema documentation here https://json-schema.org/understanding-json-schema/