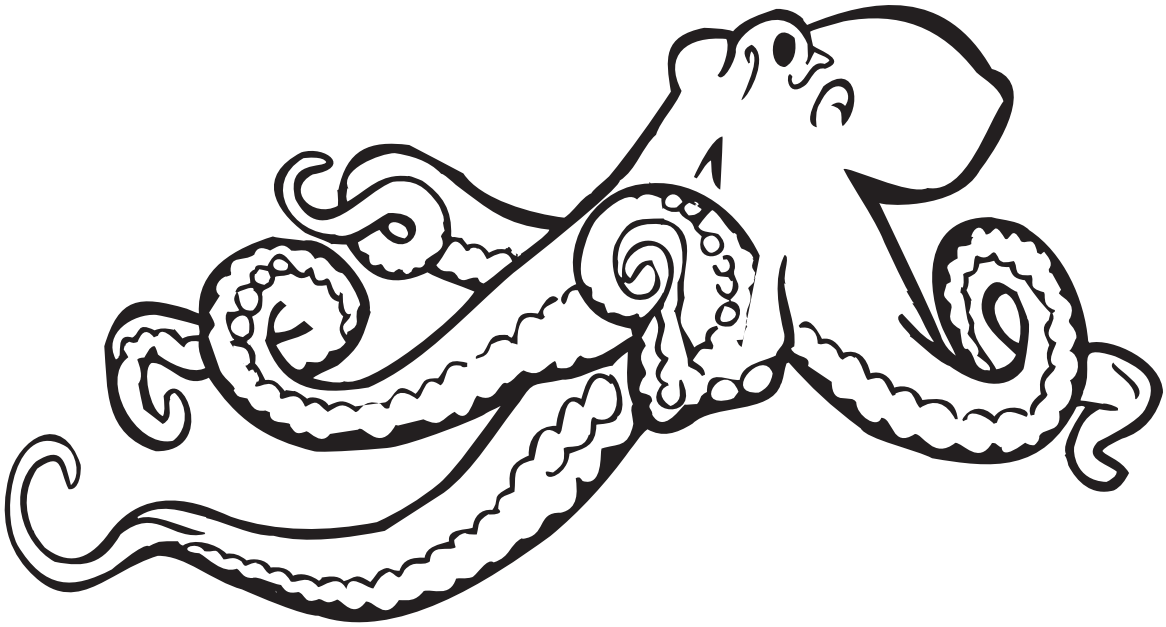

Understanding JSON Schema

Release 1.0

Michael Droettboom, et al
Space Telescope Science Institute

Dec 21, 2016

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 internal combustion engine 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 4. Earlier versions of JSON Schema are not completely compatible with the format described here.

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 5).
- *The basics* (page 9) chapter should be enough to get you started with understanding the core *JSON Schema Reference* (page 13).
- When you start developing large schemas with many nested and repeated sections, check out *Structuring a complex schema* (page 25).
- json-schema.org has a number of resources, including the official specification and tools for working with JSON Schema from various programming languages.
- jsonschema.net is an online application run your own JSON schemas against example documents. If you want to try things out without installing any software, it's a very handy resource.

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 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 JavaScript types to their analogous types in Python:

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

Ruby

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

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

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",
  "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-time" },
    "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": "22-02-1732",
  "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 5), 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" }
```

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 13).

3.3 Declaring a 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.

{ json schema }

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

You can also use this keyword to declare which version of the JSON Schema specification that the schema is written to. See *The $schema keyword* (page 22) for more information.

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 *The id property* (page 27) become more apparent when you start *Structuring a complex schema* (page 25).

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 ??)
- *Numeric types* (page ??)
- *object* (page ??)
- *array* (page ??)
- *boolean* (page ??)
- *null* (page ??)

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

Python

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

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

Ruby

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

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

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 Generic keywords

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

4.2.1 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.

{ json schema }

```
{
  "title" : "Match anything",
  "description" : "This is a schema that matches anything.",
  "default" : "Default value"
}
```

4.2.2 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 }

```
{
  "type": "string",
  "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
```

However, in most cases, the elements in the `enum` array should also be valid against the enclosing schema:

{ json schema }

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

✓

```
"red"
```

This is in the `enum`, but it's invalid against `{ "type": "string" }`, so it's ultimately invalid:

✗

```
null
```

4.3 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* (page 25). 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.

{ json schema }

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

✓

```
"short"
```

✗

```
"too long"
```

✓

```
12
```

✗

```
-5
```

The keywords used to combine schemas are:

- *allOf* (page 18): Must be valid against *all* of the subschemas
- *anyOf* (page 20): Must be valid against *any* of the subschemas
- *oneOf* (page 21): 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 22): Must *not* be valid against the given schema

4.3.1 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 that it's quite easy to create schemas that are logical impossibilities with `allOf`. 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
```

It is important to note that the schemas listed in an *allOf* (page 18), *anyOf* (page 20) or *oneOf* (page 21) array know nothing of one another. While it might be surprising, *allOf* (page 18) 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 `definitions` section, and want to extend it to include an address type:

{ json schema }

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

  "allOf": [
    { "$ref": "#/definitions/address" },
    { "properties": {
        "type": { "enum": [ "residential", "business" ] }
      }
    }
  ]
}
```

✓

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

This works, but what if we wanted to restrict the schema so no additional properties are allowed? One might try adding the highlighted line below:

{ json schema }

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

  "allOf": [
    { "$ref": "#/definitions/address" },
    { "properties": {
        "type": { "enum": [ "residential", "business" ] }
      }
    }
  ],

  "additionalProperties": false
}
```

✗

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

Unfortunately, now the schema will reject *everything*. This is because the *Properties* (page ??) refers to the entire schema. And that entire schema includes no properties, and knows nothing about the properties in the subschemas inside of the *allOf* (page 18) array.

This shortcoming is perhaps one of the biggest surprises of the combining operations in JSON schema: it does not behave like inheritance in an object-oriented language. There are some proposals to address this in the next version of the JSON schema specification.

4.3.2 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" },
    { "type": "number" }
  ]
}
```

✓

```
"Yes"
```

✓

```
42
```

✗

```
{ "Not a": "string or number" }
```

4.3.3 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
```

Note that it's possible to "factor" out the common parts of the subschemas. The following schema is equivalent to the one above:

{ json schema }

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

4.3.4 not

This doesn't strictly combine schemas, but it belongs in this chapter along with other things that help to modify the effect of schemas in some way. The `not` keyword declares that a 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"
```

4.4 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#"
```

4.4.1 Advanced

If you need to declare that your schema was written against a specific version of the JSON Schema standard, and not just the latest version, you can use one of these predefined values:

- `http://json-schema.org/schema#`

  JSON Schema written against the current version of the specification.

- `http://json-schema.org/hyper-schema#`

  JSON Schema hyperschema written against the current version of the specification.

- `http://json-schema.org/draft-04/schema#`

  JSON Schema written against this version.

- `http://json-schema.org/draft-04/hyper-schema#`

  JSON Schema hyperschema written against this version.

- `http://json-schema.org/draft-03/schema#`

  JSON Schema written against JSON Schema, draft v3

- `http://json-schema.org/draft-03/hyper-schema#`

  JSON Schema hyperschema written against JSON Schema, draft v3

Additionally, if you have extended the JSON Schema language to include your own custom keywords for validating values, you can use a custom URI for `$schema`. It must not be one of the predefined values above.

4.5 Regular Expressions

The *pattern* (page ??) and *Pattern Properties* (page ??) 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 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.

Python

This subset of JavaScript regular expressions is compatible with Python regular expressions. Pay close attention to what is missing, however. Notably, it is not recommended to use `.` to match any character.

4.5.1 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"
```

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

5.1 Reuse

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 does it make the schema more verbose, but it makes updating it in the future more difficult. If our imaginary company were to start 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.

Note: This is part of the draft 4 spec only, and does not exist in draft 3.

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.

Note: JSON Pointer aims to serve the same purpose as 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:

{ json schema }

```
{
  "$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" }
  }
}
```

✓

```
{
  "shipping_address": {
    "street_address": "1600 Pennsylvania Avenue NW",
    "city": "Washington",
    "state": "DC"
  },
  "billing_address": {
    "street_address": "1st Street SE",
    "city": "Washington",
    "state": "DC"
  }
}
```

5.2 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.

5.3 Extending

The power of `$ref` really shines when it is combined with the combining keywords `allOf`, `anyOf` and `oneOf` (see *Combining schemas* (page 17)).

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"]
    }
  ]
}
```

Tying this all together,

{ json schema }

```
{
  "$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": { "$ref": "#/definitions/address" },
    "shipping_address": {
      "allOf": [
        { "$ref": "#/definitions/address" },
        { "properties":
          { "type": { "enum": [ "residential", "business" ] } },
          "required": ["type"]
        }
      ]
    }
  }
}
```

This fails, because it's missing an address type:

✗

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

✓

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

From these basic pieces, it's possible to build very powerful constructions without a lot of duplication.

CHAPTER 6

ACKNOWLEDGMENTS

Michael Droettboom wishes to thank the following contributors:

- Anders D. Johnson
- Armand Abric
- Ben Hutton
- btubbs
- Chris Carpenter
- Christopher Mark Gore
- David Branner
- David Worth
- Erik Bray
- forevermatt
- goldaxe
- Jesse Claven
- Vincent Jacques

INDEX