Elegant APIs with JSON Schema *was published on* ***May 27, 2014*** *from* ***Berlin****.*

Find me on Twitter at ***@brandur****.*

Elegant APIs with JSON Schema

Contents

We've recently gone on record indicating our commitment to using JSON Schema as the format for describing our API's, then even further by releasing a set of tools to improve the process of building and working with schema-based HTTP API's. With the recent rise of great API description formats over the last few years like Swagger, Blueprint, and RAML (among others), I wanted to write a few words on what JSON Schema is, why it's a neat technology, and how it can be applied specifically to building great APIs.

At any time, you can jump into more documentation over at jsonschema.org, which includes detailed draft specifications for both JSON Schema and JSON Hyper-schema.

The basic case

At its essence, JSON Schema is simply a declarative language for validating the format and structure of a JSON object. It allows you to specify a number of special primitives to describe exactly what a valid JSON object will look like, and provides a powerful nesting concept that allows you to extend these primitives to a document of any complexity. This idea hails back to the days of XML, when it was common to see XML documents linking to the XSD's (XML Schema Definition) that should be used to validate them.

Let's start with one of the most basic schemas possible. The following describes a single value inside a JSON object:

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

The value `"foo"` would validate successfully while `123` or `false` would not.

More complex rules can be mixed into the object as well. This will validate that the string matches a particular regex pattern:

```
{
  "pattern": "^[a-z][a-z0-9-]{2,30}$",
  "type": "string"
}
```

Nesting schemas

While the above lets us validate a single value, it's more interesting to validate a complex JSON object. We can build on the above by nesting our single value validation into another schema using the `properties` keyword, which describes the keys that a JSON object might have, and the schema that validates their values:

```
{
  "properties": {
    "name": {
      "pattern": "^[a-z][a-z0-9-]{2,30}$",
      "type": "string"
    }
  },
  "required": ["name"],
  "type": "object"
}
```

The `required` keyword indicates that the property `name` is expected, so while the object `{"name":"foo"}` is valid, `{}` is not.

Note how the `type` keyword is present in both of the objects in our schema above. This is where the elegance of JSON Schema starts to emerge: **both objects are JSON Schemas that are defined to precisely the same specification**. We could give the `name` object its own `definitions`, but that would be non-sensical because it's defined as a `string` rather than an `object`.

A very common convention in cases like this is to define subschemas under `definitions` and reference them from elsewhere, which allows those schema definitions to be re-used. Like `properties`, `definitions` also maps object keys to schemas, but doesn't suggest that those keys should actually be properties

on an object being validated; it's simply a useful mechanism for defining schemas in a common place. The above could be re-written to use `definitions` like so:

```
{
  "definitions": {
    "name": {
      "pattern": "^[a-z][a-z0-9-]{2,30}$",
      "type": "string"
    }
  },
  "properties": {
    "name": {
      "$ref": "#/definitions/name"
    }
  },
  "required": ["name"],
  "type": "object"
}
```

The strange `$ref` keyword is a JSON Reference. It tells schema parsers that the definition is not a schema itself, but rather references a schema elsewhere in the document (or in a different document). The `#` denotes the root of the JSON document, and the slashes are keys that should be descended through until the appropriate value is reached.

We need to go deeper

Let's think of our schema above as the definition of a simple app, which has a name, but might later have some other properties as well. A very common scenario (especially in an API) might be to define another type of object as well, and to have these objects reference each other.

Along with our app, let's define a domain:

```
{
  "definitions": {
    "name": {
      "format": "hostname",
      "type": "string"
    }
  },
  "properties": {
    "name": {
      "$ref": "#/definitions/name"
    }
  },
  "required": ["name"],
  "type": "object"
}
```

Domain looks a lot like an app, with its own `name` and property definitions. Note above that we've defined that domain's `name` is in the `hostname` format, which is a special string validation built into JSON Schema.

Now, remember how I told you that schemas nest? They do, and we've already seen how they can be nested one level deep above. To make this even better though, we can actually nest them to *any* level. Let's put app and domain into the same root schema which will eventually be used to define our entire API. Note how the references below change to reflect the greater depth of nesting.

```
{
  "definitions": {
    "app": {
      "definitions": {
        "domains": {
          "items": {
            "$ref": "#/definitions/domain"
          },
          "type": "array"
        },
        "name": {
          "pattern": "^[a-z][a-z0-9-]{2,30}$",
          "type": "string"
        }
      },
      "properties": {
        "domains": {
          "$ref": "#/definitions/app/definitions/domains"
        },
        "name": {
          "$ref": "#/definitions/app/definitions/name"
        }
      },
      "required": ["name"],
      "type": "object"
    },
    "domain": {
      "definitions": {
        "name": {
          "format": "hostname",
          "type": "string"
        }
      },
      "properties": {
        "name": {
          "$ref": "#/definitions/domain/definitions/name"
        }
      },
      "required": ["name"],
      "type": "object"
    }
  },
  "properties": {
    "app": {
      "$ref": "#/definitions/app"
    },
    "domain": {
      "$ref": "#/definitions/domain"
    }
```

```
  },
  "type": "object"
}
```

Phew! We've managed to build out a pretty significant schema already. Astute readers may have noticed that along with the new domains resource, we've defined a new property for app:

```
"domains": {
  "items": {
    "$ref": "#/definitions/domain",
  },
  "type": "array"
}
```

`items` is another special keyword that applies specifically to the `array` type. It indicates that all items in the array should conform to the referenced schema; in this case, that means that `domains` should be an array of objects that validate according to the `domain` schema. For example, this array validates correctly:

```
[
  { "name": "example.com" },
  { "name": "heroku.com" }
]
```

We've now demonstrated not only how schemas can be nested to as many levels as we need, but also how subschemas can start to reference each other to do build out more complex validation rules in a modular way.

Once again, discerning readers may have noticed that our top-level schema actually defines a non-sensical object that has both an app and a domain like `{"app":..., "domain":...}`. This is true, but we'll see that it's not important as we move onto building an API in the next section.

Adding hyper-schema

Along with the JSON Schema, a companion draft also defines JSON Hyper-schema, which builds off the original specification to define a schema that can host a collection of links. This allows us to move beyond the realm of basic JSON validation, and into the more interesting area of using schema to build APIs.

Let's define two simple links on our app schema for creating a new app (`POST /apps`) and listing existing ones (`GET /apps`):

```
{
  "definitions": ...,
  "links": [
    {
      "description": "Create a new app.",
      "href": "/apps",
      "method": "POST",
      "rel": "create",
      "title": "Create"
    },
    {
      "description": "List apps.",
      "href": "/apps",
      "method": "GET",
      "rel": "instances",
      "title": "List"
    }
  ],
  "properties": ...,
  "required": ["name"],
  "type": "object"
}
```

Notice how these define individual HTTP endpoints: an access verb is specified in `method`, along with a URI in `href`. We've also tagged each link with some other metadata that tells us a little more about what it does and how we should describe it; this can be supremely useful for tasks like generating code and documentation.

Request schemas

The links above are useful in that we now know a little bit about how to interact with an apps resource, but they don't tell us much beyond that. For example, how do we know what parameters to send in while creating an app?

Luckily, hyper-schema also allows us to nest schemas to describe just that. Let's leverage references once again, and define a create app link that requires a valid app object to be sent in along with a request:

```
{
  "description": "Create a new app.",
  "href": "/apps",
  "method": "POST",
  "rel": "create",
  "schema": {
    "$ref": "#/definitions/app"
  },
  "title": "Create"
}
```

Note that although the above is fine in this trivial case, we often want to define required request parameters to be a subset of what we might see in a fully valid object. Because we're defining a schema like any other, we can de-construct it and reference particular properties that we want to see in the incoming request:

```
{
  "description": "Create a new app.",
  "href": "/apps",
  "method": "POST",
  "rel": "create",
  "schema": {
    "properties": {
      "name": {
        "$ref": "#/definitions/app/definitions/name"
      }
```

```
    },
    "required": ["name"],
    "type": "object"
  },
  "title": "Create"
}
```

A request to an API implementing this schema might look like the following:

```
curl -X POST http://example.com/apps \
  -H "Content-Type: application/json" \
  -d '{"name":"my-app"}'
```

We could also remove the name requirement (`"required": ["name"]`) if we wanted to generate a name for the new app unless the user explicitly overrides it. In that case, an empty JSON object `{}` would be a valid request for this endpoint.

Once again, I'd like to draw your attention to the elegant modularity of JSON Schema here. We've defined a property on our app object (`name`) one time, then referenced it to describe what a valid app looks like, then used the same technique to reference it again to describe a valid request.

A declarative definition of incoming requests can be supremely useful for sanitizing data and generating errors for malformed data automatically. A tool like Committee, which provides a collection of schema-related middleware, can help with this in Ruby.

Note that the API I'm building above is a little like the Heroku API in that it expects input as `application/json` rather than the more commonly seen `application/x-www-form-urlencoded` (e.g. `name=my-app&foo=bar`). Hyper-schema doesn't necessarily stipulate that incoming requests have to be JSON, in fact it defines an `encType` that allows a link to specify its format, but the symmetry of a request and response that are both in JSON is a clean model worthy of consideration (in my humble opinion).

Response schemas

Much like the incoming request, Hyper-schema allows us to specify a schema for the outgoing response as well with the `targetSchema` keyword. Within the confines of our simple example API above, this one is easy; given a request to create an app, let's respond with an app:

```
{
  "description": "Create a new app.",
  "href": "/apps",
  "method": "POST",
  "rel": "create",
  "targetSchema": {
    "$ref": "#/definitions/app"
  },
  "title": "Create"
}
```

For the list endpoint, we'd like to describe the response as an array of apps:

```
{
  "description": "List apps.",
  "href": "/apps",
  "method": "GET",
  "rel": "instances",
  "targetSchema": {
    "items": {
      "$ref": "#/definitions/app"
    },
    "type": "array"
  },
  "title": "List"
}
```

And we've managed to re-use our basic object definitions yet again! Knowing what responses are supposed to look like can be very handy for testing for API regressions in acceptance-level tests. Once again, Committee can help with that in Ruby by providing test helpers for use with rack-test.

Let's get meta

An interesting set of products that both JSON Schema and Hyper-schema provide are their own meta-schemas. Because a schema is itself just a JSON document, a schema can be written for a schema! For example, take a look at the JSON Hyper-schema meta-schema. Note how the special `$schema` keyword points back to its own `id`. This schema can be used to validate the format of your own Hyper-schema with a tool like `json_schema`:

```
validate-schema --detect my-schema.json
```

As we all know, convention can be a very challenging problem, especially when working within a larger team of people who all have their own ideas of what a good API looks like. One possible solution to this problem is to start defining convention declaratively by writing a meta-schema that enforces a layer of constraints on top of what's already dictated by the schema and hyper-schema specifications themselves.

For example, a hyper-schema only dictates that a link specifies the `href` and `rel` attributes. We could require that a few more keys are present as well:

```
{
  "$schema": "http://example.com/my-hyper-schema",
  "definitions": {
    "resource": {
      "properties": {
        "links": {
          "items": {
            "$ref": "#/definitions/link"
          },
          "type": "array"
        }
      }
    },
    "link": {
      "required": [ "href", "method", "rel", "targetSchema" ],
      "type": "object"
    }
  },
  "id": "http://example.com/my-hyper-schema#",
  "title": "My JSON Hyper-Schema Variant",
  "properties": {
```

```
    "definitions": {
      "additionalProperties": {
        "$ref": "#/definitions/resource"
      }
    }
  }
}
```

It may be necessary to read some documentation to understand all the specific keywords in use here, but in essence what we're declaring here is that everything under definitions in our hyper-schema is an API resource (see resource under definitions), and that those resources may have links (link under definitions). Those links should have the properties href, method, rel, and targetSchema.

Checking the validity of our schema above with validate-schema from the json_schema, we get this:

```
validate-schema -d -s meta.json schema.json
schema.json is valid.
```

But if we leave targetSchema out of our first link, we get this instead:

```
validate-schema -d -s meta.json schema.json
schema.json#/definitions/app/links/0: failed schema #/definitions/resource/properties/links/items: Missing required keys "targetSchema" in object; keys are "description, href, method, rel, schema, title".
```

We could also mandate that all resource property names should be lowercase only with underscores allowed:

```
"resource": {
  "properties": {
    ...,
    "properties": {
      "additionalProperties": false,
      "patternProperties": {
        "^[a-z][a-z_]+[a-z]$": {}
      }
    }
  }
},
```

Note that the patternProperties keyword allows us to match on a schema based on the name of a property in an object, and additionalProperties set to false dictates that properties that are not in the properties object or defined in patternProperties are not valid. Re-running again we see that all the property names we defined are okay:

```
validate-schema -d -s meta.json schema.json
schema.json is valid.
```

Mixing in hyper-schema's meta-schema

You may also notice that the hyper-schema meta-schema uses an allOf attribute to make sure that in addition to the constraints it defines, data should also validate against the JSON Schema meta-schema as well. We can do the same thing for our variant except for hyper-schema:

```
{
  "$schema": "http://example.com/my-hyper-schema#",
  "allOf": [
    {
      "$ref": "http://json-schema.org/draft-04/hyper-schema#"
    }
  ],
  ...
}
```

Schema endpoint

A convention that we have at Heroku is to serve the schema itself when a request is made to GET /schema. One neat trick is to define the /schema link in the schema itself and that its response should validate according to its meta-schema. Using the same mechanism that you'd use to check that a JSON response conforms to its schema, this allows the schema to validate itself against its own meta-schema from your acceptance test suite!

```
{
  "href": "/schema",
  "method": "GET",
  "rel": "self",
  "targetSchema": {
    "$ref": "http://example.com/my-hyper-schema#",
  }
}
```

All the code for the simple hyper-schema and the meta-schema that we've built here are available on GitHub.

Schemas for other media types

A final point worth mentioning is that even a Hyper-schema API isn't your thing, Hyperschema.org has a set of schemas available for other media types, including today's popular hypermedia formats like Collection+JSON, HAL, and UBER.

Summary

To recap, we've used JSON Schema to define the following:

- Individual API resources (app and domain).
- An API "super schema" that contains all resources in a single document.
- Hyper-schema links that describe actions on those resources.
- Schemas that validate incoming requests on each link.
- Schemas that describe the JSON returned by each link.
- A meta-schema that validates the conventions of our API's schema.

Although the API itself still needs to be implemented, by combining this schema with the various packages from the HTTP toolchain, we get some nice features for free:

- Generate API documentation with Prmd.
- Generate a Ruby client with Heroics.
- Generate a Go client with Schematic, like the one used in Heroku's new CLI, hk.
- Boot a working stub with Committee that will validate incoming requests.
- Insert a request validation middleware with Committee that will validate incoming request data according to schema before it reaches our stack.
- Use Committee's test helpers to verify that the responses from our stack conform to the schema.

Elegant APIs with JSON Schema *was published on* ***May 27, 2014*** *from* ***Berlin****.*

Find me on Twitter at ***@brandur****.*

Did I make a mistake? Please consider sending a pull request.