TIMOTHY B. JACOBS

Just another WordPress developer

HOME PLUGINS SLIDES CONTACT

REST API

JSON Schema and the WP REST API

Written by **Timothy Jacobs** on **May 17th, 2017. 2 Comments**

JSON Schema is a *draft* ITEF specification that provides a "vocabulary that allows you to annotate and validate JSON documents" – *json-schema.org*. The WordPress REST API utilizes this system for describing the request and response formats for each endpoint. The schema for any endpoint can be seen by making an OPTIONS request to the endpoint.

```
curl -X OPTIONS https://timothybjacobs.com/wp-json/wp/v2/posts
{
  "$schema": "http://json-schema.org/schema#",
  "title": "post",
  "type": "object",
  "properties": {
    // property list...
  }
}
```

A basic schema document consists of a few properties.

- `$schema` – A reference to a meta schema that describes the version of the spec the document is using.
- `title` – The title of the schema. Normally this is a human readable label, but in WordPress this is machine readable string . The posts endpoint, for example, has a title of 'post'. Comments has a title of 'comment'.
- `type` – This refers to the type of the value being described. This can be any one of the seven primitive types: array, boolean, integer, null,

number, object, or string. In WordPress the top-level type will almost always be an `object`, even for collection endpoints that return an array of objects.
- `properties` – A list of the known properties contained in the object and their definitions. Each property definition itself is also a schema, but without the `$schema` top-level property, more accurately described as a sub-schema.

Example Property Definitions

Let's take a look at a subset of the properties in the `wp/v2/posts` endpoint.

```
          "rendered": {
            "description": "HTML content for the object, transformed for di
            "type": "string",
            "readonly": true
          },
          "protected": {
            "description": "Whether the content is protected with a passwor
            "type": "boolean",
            "readonly": true
          }
        }
      },
      "categories": {
        "description": "The terms assigned to the object in the category ta
        "type": "array",
        "items": {
          "type": "integer"
        }
      },
    }
  }
```

Let's take a look at the first property, `id`. WordPress provides a description of what the field represents. This is the `WP_Post::$id` property. It has a `type` of `integer` which means a number without a decimal point. Additionally, the property is marked as `readonly`, which means that the value cannot be set via a `POST` or `PUT` request.

The next property is `sticky`. This has a `type` of `boolean` which means `true` and `false` are the only supported values. Since PHP does not support non-string types in inputs, WordPress also allows you to pass the strings `'true'` and `'false'` to achieve a boolean value.

String Formats

The `date` property is a `string` type, but what we want is a date. JSON Schema does not have a date primitive, instead it makes use of the `format` keyword. The specification defines a number of supported formats, of note is `date-time`, `uri`, and `email`. The `date-time` format requires the value to be passed in `ISO8601` format: `Y-m-dTH:i:sP` (PHP). For example, May 31st, 2017 at 6:30pm would be, `2017-05-31T18:30:00Z`.

Similarly, the `link` property is a string, but defines a `format` of `uri`. This refers to the WordPress permalink of the post

Next up is another string property, `status`. A WordPress post status does not have any defined structure, but there are a limited number of statuses supported. The `enum` keyword defines a list of the values allowed.

Structures

The `content` property is a bit more complex. It has a `type` of `object` and defines its own list of `properties`. This is where the recursive nature of JSON Schema documents come into play. Any of the properties defined for `content` could also be an object. In this case, however, it’s two strings, `raw` and `rendered`, and a boolean value, `protected`. The `rendered` property is marked as `readonly` which means that it cannot be updated, but the `raw` property can be. This would correspond to the following JSON.

```
{
  "content": {
    "raw": "My post with a [shortcode]",
    "rendered": "My post with a rendered shortcode",
    "protected": false
  }
}
```

Last is the `categories` property. This has a `type` of `array`. That means that there isn’t a single value, but a list of values. Each value in the array should be described with a sub-schema in the `items` property. In this case, each item is defined to have a `type` of `integer`.

```
{
  "categories": [5, 10, 13]
}
```

But this could just as easily be an array of objects, or even an array of arrays.

```
// Schema
{
  "categories": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "propA": { "type": "string" },
        "propB": { "type": "integer" }
      }
    }
  }
}
// JSON
{
  "categories": [
    { "propA": "hi", "propB": 5 },
    { "propA": "hello", "propB": 12 }
  ]
}
```

Validation

Schema documents provide a great way to inform API clients of the response format for every endpoint. The structure of the document means it can be read natively by humans or transformed into always accurate documentation. Schemas aren't only used for documentation, though, they also provide a platform-independent means of validating data.

Procedural Validation

Normally in PHP, validation of data is performed procedurally. For example:

```
if ( ! isset( $_GET['status'], $_GET['date'], $_GET['content'] ) )
  return new \WP_Error( 'missing_params' );

$status  = in_array( $_GET['status'], [ 'draft', 'published' ], true ) ? $_
$date    = strtotime( $_GET['date'] );
$content = $_GET['content'];

if ( ! $status ) return new \WP_Error( 'invalid_status' );
if ( ! $date ) return new \WP_Error( 'invalid_date' );
```

While this works for simple validation needs, it has a number of drawbacks.

1. Separation of concerns. The code concerns itself with performing validation as opposed to describing the validation requirements. This leads to long line lengths and long methods.

2. Client side validation requires duplicating business logic which leads to unmaintainable code.
3. Duplication of data when seeding input fields. For example, providing options to a `<select>` dropdown or specifying the field type or other attributes of an `<input>` element.

JSON Schema can alleviate these issues.

1. The schema describes the format of a valid value. The validator interprets the schema and performs the actual validation.
2. JSON Schema is portable. Validator implementations exist for PHP, JavaScript, and numerous other languages.
3. A well structured Schema can be used to automatically generate UIs in their entirety or in part.

Schema Validation

When registering a REST route with WordPress, you can provide a function that returns the schema for that route in the `schema` option. This is the schema that is returned when making an `OPTIONS` request. In addition to the `schema` argument, an `args` argument is provided which, when subclassing `WP_REST_Controller`, is built from the schema using `get_endpoint_args_for_item_schema()`.

Once the REST server matches a request to a given route, it instructs the request to validate itself according to the `args` for the matched route. The `rest_validate_request_arg` function performs this validation by default.

For example:

```
PUT https://timothybjacobs.com/wp-json/wp/v2/posts/1
{
  "id": 1,
  "status": "invalid",
  "sticky": false,
  "categories": [
    "my-category"
  ]
}
```

This request is invalid because `status` is not one of the supported enum values and `categories` can only contain `integer` items, not `string` items. WordPress will return an error in this case.

```
{
  "code": "rest_invalid_param",
  "message": "Invalid parameter(s): status, categories",
  "data": {
    "status": 400,
    "params": {
      "status": "status is not one of publish, future, draft, pending, priv
      "categories": "categories[0] is not of type integer."
    }
  }
}
```

If you start to play with the endpoint, or take a peek at the code, it will become clear that this validator only supports a small subset of the JSON Schema. Take the `content` property for example. It is defined to be an `object` with a set of three `properties`. However, you might notice that if you can pass a `string` value, that will be used to update the post content. Any other value and WordPress will silently discard it.

On the face of it, this doesn't make sense. The schema specifies that the content should be updated via the `content.raw` property and the API successfully fails when other invalid parameters are given. So what is going on here?

1. WordPress does not provide a validator for any kind of objects. This becomes clear upon examining the source of `rest_validate_value_from_schema()`.
2. To ease implementation, the API is designed to accept the post content as either just a plain `string` value *or* a `string` in `content.raw`. This isn't evident from the schema documentation, however.

Complex Schemas

JSON Schema provides a mechanism to define a multi-type schema. For example, to properly describe the `content` property we'd have a schema like the following.

```
{
  "content": {
    "description": "The content for the object.",
    "oneOf": [
      {
        "type": "string",
        "description": "Content for the object, as it exists in the databas
      },
      {
        "type": "object",
        "properties": {
          "raw": {
            "description": "Content for the object, as it exists in the dat
            "type": "string"
          },
          "rendered": {
            "description": "HTML content for the object, transformed for di
            "type": "string",
            "readonly": true
          },
          "protected": {
            "description": "Whether the content is protected with a passwor
            "type": "boolean",
            "readonly": true
          }
        }
      }
    ]
  }
}
```

This introduces a new keyword, `oneOf`. A value is valid according to `oneOf` if it matches against *exactly one* of the provided definitions. This means a client could pass either a plain `string` value *or* an `object` with the `raw` property to update the post's content.

WP REST API Schema Validator

As WordPress does not ship with a compliant Schema validator, we'll have to use an external library. To make this easier, I developed a package, `ironbound/wp-rest-api-schema-validator` which utilizes the fantastic `justinrainbow/json-schema` library to perform the validation.

Using the validator is simple.

```
( new \IronBound\WP_REST_API\SchemaValidator\Middleware( 'namespace/v1', [
  'methodParamDescription' => __( 'HTTP method to get the schema for. If no
  'schemaNotFound'         => __( 'Schema not found.', 'text-domain' ),
] ) )->initialize();
```

This will substitute WordPress core's validation functions with a more complete JSON Schema library, without effecting endpoints in other namespaces.

Documentation for the library is available on GitHub.

Advanced Schema Features

The JSON Schema website provides a human readable, but highly technical specification describing all of the supported validation mechanisms. I hope to provide some additional details/examples on the supported validation mechanisms in a WordPress context. For readability purposes, I'll utilize `@` to represent the base REST API url, i.e. `https://timothybjacobs.com/wp-json/`.

Referencing Other Schemas with $ref

One of the major benefits of JSON Schema is its ability to compose larger schemas by reusing or combining existing schemas. Other schema definitions can be referenced using the `$ref` keyword. This means a schema can be referenced from an external document or the same document using a JSON pointer.

```
{
  "$schema": "http://json-schema.org/schema#",
  "title": "customer",
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "address": { "type": "string" },
        "city": { "type": "string" },
        "state": { "type": "string" },
        "country": { "type": "string" },
      }
    }
  },
  "type": "object",
  "properties": {
    "billing": { "$ref": "#/definitions/address" },
    "shipping": { "$ref": "#/definitions/address" }
  }
}
```

In this example, a customer has a `shipping` and `billing` property which holds their billing and shipping address. The structure for each address is the same, so instead of duplicating the validation constraints for each address type, we reference another schema in the same document. By convention, these locally shared schemas should be defined under the `definitions` property.

The reference can also be a URL to an external Schema document. For example, if we had a `/addresses` route with a schema `title` of 'address'.

```
{
  "$schema": "http://json-schema.org/schema#",
  "title": "customer",
  "type": "object",
  "properties": {
    "billing": { "$ref": "@namespace/v1/schemas/address#" },
    "shipping": { "$ref": "@namespace/v1/schemas/address#" }
  }
}
```

A Note on JSON Pointers

JSON pointers are used to communicate the position of a specific value within a document. When used with `$ref` the pointer must begin with `#`. The pound sign refers to the root of the document, i.e. the part with `$schema`, `title`, etc… The pointer then moves down the document another step after every `/`. The specification covers a number of edge cases, but the following are some simple examples.

- `#/title`
  - customer
- `#/properties/billing/$ref`
  - @namespace/v1/schemas/address#
- `@namespace/v1/schemas/address#/$title`
  - address

Multiple Formats with oneOf and anyOf

An in-context example of `oneOf` can be seen with the `content` property. Crucial to `oneOf` is that only one schema can be matched. For example, this request is not valid because it matches both of the provided definitions.

```
          "propA": { "type": "string" },
          "propB": { "type": "string" }
        }
      },
      {
        "type": "object",
        "properties": {
          "propA": { "type": "string" },

          "propC": { "type": "string" }
        }
      }
    ]
  }
}
// Request
{
  "prop": {
    "propA": "value"
  }
}
```

If `anyOf` was used instead, this would successfully validate because `anyOf` only requires *at least one* schema match, not *exactly one*.

Combing Schemas with allOf

`allOf` is an especially cool keyword that requires a value match *all* of the schemas provided. This allows us to combine schemas to minimize repetition of business logic. For example, given a shared address schema, the `/addresses` schema can be defined as a combination of two schemas.

```
          "country": { "type": "string" },
        }
      }
    }
  }
  // Schema for /addresses
  {
    "$schema": "http://json-schema.org/schema#",
    "title": "saved-address",
    "allOf": [
      { "$ref": "@namespace/v1/schemas/address" },
      {
        "type": "object",
        "properties": {
          "id": { "type": "integer" },
          "label": { "type": "string" }
        }
      }
    ]
  }
```

This means that the full properties list for an object returned from `/addresses` is `address`, `city`, `state`, `country`, `id` and `label`.

Regex Matching with the Pattern Keyword

JSON Schema supports more complex `string` validation by using the `pattern` keyword. This keyword allows you to validate strings according to a regex pattern. According to the specification, the regex should be JavaScript flavor compatible. Since the main validation will be done server side, however, it is vital that the regex be PHP compatible as well. Most expressions should be compatible between the two flavors, but you should check with a tool like regex101 to be sure. For example, a simple zip code validator.

```
{
  "zipCode": {
    "type": "string",
    "pattern": "[0-9]{5}(?:-[0-9]{4})?"
  }
}
```

Validating with regular expressions are highly preferred over a procedural function definition to maintain platform interoperability. In general, PHP validation functions should be limited to checks that require runtime access like querying the database or applying filters. If possible, you should still provide an accurate schema definition so clients can still do first level validation without querying the server.

I hope this has helped to explain how the WordPress REST API and JSON Schema work together to build documentation and validate requests. Feel free to leave a comment if there are other JSON Schema keywords you'd like me to explore.

JSON Schema WP-API

Previous Post:

IronBound-DB v2

---

2 comments:

Harry says:
November 9, 2017 at 1:45 pm

Hi, good technical insight.
So, I would take advantage to ask you a question.
Do you know if there is a way to relate an object to an already defined "definition" in wp rest controller pattern?

IE:

```
// event schema
$schema = array(
    '$schema' => 'http://json-schema.org/draft-04/schema#',
    'title' => 'event',
    'type' => 'object',
    'properties' => array(...)
);

// other schema (which would like to contain the schema "event")
$other_schema = array(
    '$schema' => 'http://json-schema.org/draft-04/schema#',
    'title' => 'something_needing_an_array_of_events',
    'type' => 'object',
    'properties' => array(
        'listing' => array(
            'description' => __( 'listing of events.' )
            'type' => 'array',
            'items' => array(
                // something like...
                //'type' => 'event',
                // or
                //'$ref' => '#/definitions/event'
            ),
        )
    )
);
```

Thank you in advance

Reply

Timothy Jacobs says:
November 15, 2017 at 1:07 pm

Yep, you definitely can. If you check out the Github repo I mentioned, you can see an example of that under **Reusing Schemas**.

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT

Search …

I'm a WordPress developer at iThemes where I work full-time on our security plugin, **iThemes Security**. You can find my premium plugins and services at **Iron Bound Designs**.

Signup for my Newsletter

Receive development posts in your inbox. I'll never spam and you can unsubscribe at anytime.

Fields marked with an * are required
First Name *

Email *

SUBSCRIBE

Recent Posts

- **JSON Schema and the WP REST API**
- **IronBound-DB v2**

Proudly powered by WordPress | Theme: Simone by mor10.com