Schema

- Overview
  - JSON
  - Resource Schema
  - Argument Schema
  - Summary
- JSON Schema Basics
  - API
  - Caching Schema
  - Schema Documents
  - Primitive Types
    - Type Juggling
  - Format
  - Strings
    - minLength and maxLength
    - pattern
  - Numbers
    - minimum and maximum
    - multipleOf
  - Arrays
    - minItems and maxItems
    - uniqueItems
      - Uniqueness
  - Objects
    - Required Properties
      - Version 3 Syntax
      - Version 4 syntax
    - additionalProperties
    - patternProperties

    - minProperties and maxProperties
  - Type agnostic keywords
    - oneOf and anyOf
- Changelog
  - WordPress 5.6
  - WordPress 5.5
  - WordPress 5.4
  - WordPress 5.3
  - WordPress 4.9

Overview

A schema is metadata that tells us how our data is structured. Most databases implement some form of schema which enables us to reason about our data in a more structured manner. The WordPress REST API utilizes JSON Schema to handle the structuring of its data. You can implement endpoints without using a schema, but you will be missing out on a lot of things. It is up to you to decide what suits you best.

JSON

First, let's talk about JSON a bit. JSON is a human readable data format that resembles JavaScript objects. JSON stands for JavaScript Object Notation. JSON is growing wildly in popularity and seems to be taking the world of data structure by storm. The WordPress REST API uses a special specification for JSON known as JSON schema. To learn more about JSON Schema please check out the JSON Schema website and this easier to understand introduction to JSON Schema. Schema affords us many benefits: improved testing, discoverability, and overall better structure. Let's look at a JSON blob of data.

```
{
    "shouldBeArray": 'LOL definitely not an array',
    "shouldBeInteger": ['lolz', 'you', 'need', 'schema'],
    "shouldBeString": 123456789
}
```

A JSON parser will go through that data no problem and won't complain about anything, because it is valid JSON. The clients and servers know nothing about the data and what to expect they just see

the JSON. By implementing schema we can actually simplify our codebase. Schema will help structure our data better so our applications can more easily reason about our interactions with the WordPress REST API. The WordPress REST API does not force you to use schema, but it is encouraged. There are two ways in which schema data is incorporated into the API; schema for resources and schema for our registered arguments.

Resource Schema

The schema for a resource indicates what fields are present for a particular object. When we register our routes we can also specify the resource schema for the route. Let’s look at what a simple comment schema might look like in a PHP representation of JSON schema.

```
// Register our routes.
function prefix_register_my_comment_route() {
    register_rest_route( 'my-namespace/v1', '/comments', array(
        // Notice how we are registering multiple endpoints the 'schema' equates to an OPTIONS
request.
        array(
            'methods'  => 'GET',
            'callback' => 'prefix_get_comment_sample',
        ),
        // Register our schema callback.
        'schema' => 'prefix_get_comment_schema',
    ) );
}

add_action( 'rest_api_init', 'prefix_register_my_comment_route' );

/**
 * Grabs the five most recent comments and outputs them as a rest response.
 *
 * @param WP_REST_Request $request Current request.
 */
function prefix_get_comment_sample( $request ) {
    $args = array(
        'number' => 5,
    );
    $comments = get_comments( $args );

    $data = array();
```

```
    if ( empty( $comments ) ) {
        return rest_ensure_response( $data );
    }

    foreach ( $comments as $comment ) {
        $response = prefix_rest_prepare_comment( $comment, $request );
        $data[] = prefix_prepare_for_collection( $response );
    }

    // Return all of our comment response data.
    return rest_ensure_response( $data );
}

/**
 * Matches the comment data to the schema we want.
 *
 * @param WP_Comment $comment The comment object whose response is being prepared.
 */
function prefix_rest_prepare_comment( $comment, $request ) {
    $comment_data = array();

    $schema = prefix_get_comment_schema();

    // We are also renaming the fields to more understandable names.
    if ( isset( $schema['properties']['id'] ) ) {
        $comment_data['id'] = (int) $comment->comment_ID;
    }

    if ( isset( $schema['properties']['author'] ) ) {
        $comment_data['author'] = (int) $comment->user_id;
    }

    if ( isset( $schema['properties']['content'] ) ) {
        $comment_data['content'] = apply_filters( 'comment_text', $comment->comment_content,
$comment );
    }

    return rest_ensure_response( $comment_data );
}
```

```
/**
 * Prepare a response for inserting into a collection of responses.
 *
 * This is copied from WP_REST_Controller class in the WP REST API v2 plugin.
 *
 * @param WP_REST_Response $response Response object.
 * @return array Response data, ready for insertion into collection data.
 */
function prefix_prepare_for_collection( $response ) {
    if ( ! ( $response instanceof WP_REST_Response ) ) {
        return $response;
    }

    $data  = (array) $response->get_data();
    $links = rest_get_server()::get_compact_response_links( $response );

    if ( ! empty( $links ) ) {
        $data['_links'] = $links;
    }

    return $data;
}

/**
 * Get our sample schema for comments.
 */
function prefix_get_comment_schema() {
    $schema = array(
        // This tells the spec of JSON Schema we are using which is draft 4.
        '$schema'              => 'http://json-schema.org/draft-04/schema#',
        // The title property marks the identity of the resource.
        'title'                => 'comment',
        'type'                 => 'object',
        // In JSON Schema you can specify object properties in the properties attribute.
        'properties'           => array(
            'id' => array(
                'description'  => esc_html__( 'Unique identifier for the object.', 'my-
textdomain' ),
                'type'         => 'integer',
```

```
                'context'      => array( 'view', 'edit', 'embed' ),
                'readonly'     => true,
            ),
            'author' => array(
                'description'  => esc_html__( 'The id of the user object, if author was a
user.', 'my-textdomain' ),
                'type'         => 'integer',
            ),
            'content' => array(
                'description'  => esc_html__( 'The content for the object.', 'my-textdomain'
),
                'type'         => 'string',
            ),
        ),
    );

    return $schema;
}
```

If you notice, each comment resource now matches up to our schema that we specified. We made this switch in `prefix_rest_prepare_comment()`. By creating schema for our resources, we can now view this schema by making `OPTIONS` requests. Why is this useful? If we wanted other languages, JavaScript for example, to interpret our data and validate the data from our endpoint, JavaScript would need to know how our data is structured. When we provide schema, we open the doors for other authors, and ourselves, to build on top of our endpoints in a consistent manner. Schema provides machine readable data, so potentially anything that can read JSON can understand what kind of data it is looking at. When we look at the API index by making a `GET` request to `https://ourawesomesite.com/wp-json/`, we are returned the schema of our API, enabling others to write client libraries to interpret our data. This process of reading schema data is known as discovery. When we have provided schema for a resource we make that resource discoverable via OPTIONS` requests to that route. Exposing resource schema is only one part of our schema puzzle. We also want to use schema for our registered arguments.

Argument Schema

When we register request arguments for an endpoint, we can also use JSON Schema to provide us data about what the arguments should be. This enables us to write validation libraries that can be reused as our endpoints expand. Schema is more work upfront, but if you are going to write a production application that will grow, you should definitely consider using schema. Let’s look at an example of using argument schema and validation.

```
// Register our routes.
function prefix_register_my_arg_route() {
    register_rest_route( 'my-namespace/v1', '/schema-arg', array(
        // Here we register our endpoint.
        array(
            'methods'  => 'GET',
            'callback' => 'prefix_get_item',
            'args' => prefix_get_endpoint_args(),
        ),
    ) );
}

// Hook registration into 'rest_api_init' hook.
add_action( 'rest_api_init', 'prefix_register_my_arg_route' );

/**
 * Returns the request argument `my-arg` as a rest response.
 *
 * @param WP_REST_Request $request Current request.
 */
function prefix_get_item( $request ) {
    // If we didn't use required in the schema this would throw an error when my arg is not
set.
    return rest_ensure_response( $request['my-arg'] );
}

/**
 * Get the argument schema for this example endpoint.
 */
function prefix_get_endpoint_args() {
    $args = array();

    // Here we add our PHP representation of JSON Schema.
    $args['my-arg'] = array(
        'description'       => esc_html__( 'This is the argument our endpoint returns.', 'my-
textdomain' ),
        'type'              => 'string',
        'validate_callback' => 'prefix_validate_my_arg',
        'sanitize_callback' => 'prefix_sanitize_my_arg',
        'required'          => true,
```

```
    );

    return $args;
}

/**
 * Our validation callback for `my-arg` parameter.
 *
 * @param mixed           $value   Value of the my-arg parameter.
 * @param WP_REST_Request $request Current request object.
 * @param string          $param   The name of the parameter in this case, 'my-arg'.
 * @return true|WP_Error True if the data is valid, WP_Error otherwise.
 */
function prefix_validate_my_arg( $value, $request, $param ) {
    $attributes = $request->get_attributes();

    if ( isset( $attributes['args'][ $param ] ) ) {
        $argument = $attributes['args'][ $param ];
        // Check to make sure our argument is a string.
        if ( 'string' === $argument['type'] && ! is_string( $value ) ) {
            return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%1$s is not of
type %2$s', 'my-textdomain' ), $param, 'string' ), array( 'status' => 400 ) );
        }
    } else {
        // This code won't execute because we have specified this argument as required.
        // If we reused this validation callback and did not have required args then this
would fire.
        return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%s was not registered
as a request argument.', 'my-textdomain' ), $param ), array( 'status' => 400 ) );
    }

    // If we got this far then the data is valid.
    return true;
}

/**
 * Our sanitization callback for `my-arg` parameter.
 *
 * @param mixed           $value   Value of the my-arg parameter.
 * @param WP_REST_Request $request Current request object.
```

```
 * @param string          $param   The name of the parameter in this case, 'my-arg'.
 * @return mixed|WP_Error The sanitize value, or a WP_Error if the data could not be
sanitized.
 */
function prefix_sanitize_my_arg( $value, $request, $param ) {
    $attributes = $request->get_attributes();

    if ( isset( $attributes['args'][ $param ] ) ) {
        $argument = $attributes['args'][ $param ];
        // Check to make sure our argument is a string.
        if ( 'string' === $argument['type'] ) {
            return sanitize_text_field( $value );
        }
    } else {
        // This code won't execute because we have specified this argument as required.
        // If we reused this validation callback and did not have required args then this
would fire.
        return new WP_Error( 'rest_invalid_param', sprintf( esc_html__( '%s was not registered
as a request argument.', 'my-textdomain' ), $param ), array( 'status' => 400 ) );
    }

    // If we got this far then something went wrong don't use user input.
    return new WP_Error( 'rest_api_sad', esc_html__( 'Something went terribly wrong.', 'my-
textdomain' ), array( 'status' => 500 ) );
}
```

In the example above we have abstracted away from using the `'my-arg'` name. We can use these validation and sanitizing functions for any other argument that should be a string we have specified schema for. As your codebase and endpoints grow, schema will help keep your code lightweight and maintainable. Without schema you can validate and sanitize, however it will be more difficult to keep track of which functions should be validating what. By adding schema to request arguments we can also expose our argument schema to clients, so validation libraries can be built client side which can help performance by preventing invalid requests from ever being sent to the API.

If you are uncomfortable with using schema, it is still possible to have validate/sanitize callbacks for each of your arguments, and in some cases it will make the most sense to do a custom validation.

Summary

Schema can seem silly at points and possibly like unnecessary work, but if you want maintainable, discoverable, and easily extensible endpoints, it is essential to use schema. Schema also helps to self document your endpoints both for humans and computers!

JSON Schema Basics

WordPress implements a validator that uses a subset of the JSON Schema Version 4 specification. The RFC is recommended reading to gain a deeper understanding of how JSON Schema works, but this article will describe the basics of JSON Schema and what features WordPress supports.

API

The REST API defines two main functions for using JSON Schema: `rest_validate_value_from_schema` and `rest_sanitize_value_from_schema`. Both functions accept the request data as the first parameter, the parameter's schema definition as the second parameter, and optionally the parameter's name as the third parameter. The validate function returns either `true` or a `WP_Error` instance depending on if the data successfully validates against the schema. The sanitize function returns a sanitized form of the data passed to the function, or a `WP_Error` instance if the data cannot be safely sanitized. When calling these functions, you should take care to *always* first validate the data using `rest_validate_value_from_schema`, and then if that function returns `true`, sanitize the data using `rest_sanitize_value_from_schema`. Not using both can open up your endpoint to security vulnerabilities. If your endpoint is implemented using a subclass of `WP_REST_Controller`, the `WP_REST_Controller::get_endpoint_args_for_item_schema` method will automatically mark your arguments as using the built-in validate and sanitize callbacks. As such, there is no need to manually specify the callbacks. If your endpoint does not follow the controller class pattern, args returned from `WP_REST_Controller::get_collection_params()`, or any other instance where callbacks are not specified, the `WP_REST_Request` object will apply sanitization and validation using the `rest_parse_request_arg` function. Importantly, this is only applied when the `sanitize_callback` is not defined. As such, if you specify a custom `sanitize_callback` for your argument definition, the built-in JSON Schema validation will not apply. If you need this validation, you should manually specify `rest_validate_request_arg` as the `validate_callback` in your argument definition.

Caching Schema

Schema may be complex, and can take time to generate. You should consider caching generated schema in your plugin's custom endpoints to avoid repeatedly generating the same schema object.

If you are defining your endpoint using a a subclass of `WP_REST_Controller`, that might look like this:

```
    /**
```

```
     * Retrieves the attachment's schema, conforming to JSON Schema.
     *
     * @return array Item schema as an array.
     */
    public function get_item_schema() {
        // Returned cached copy whenever available.
        if ( $this->schema ) {
            return $this->add_additional_fields_schema( $this->schema );
        }

        $schema = parent::get_item_schema();
        // Add endpoint-specific properties to Schema.
        $schema['properties']['field_name'] = array( /* ... */ );
        $schema['properties']['etcetera'] = array( /* ... */ );

        // Cache generated schema on endpoint instance.
        $this->schema = $schema;

        return $this->add_additional_fields_schema( $this->schema );
    }
```

This pattern was introduced into WordPress core in version 5.3 in #47871, and yielded up to 40% speed improvement when generating some API responses.

Schema Documents

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. 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.

Primitive Types

JSON Schema defines a list of seven allowed primitive types.

- `string` A string value.
- `null` A `null` value.
- `number` Any number. Decimals are allowed. Equivalent to a `float` in PHP.
- `integer` A number, but decimals or exponents are not allowed.
- `boolean` A `true` or `false` value.
- `array` A list of values. This is equivalent to a JavaScript array. In PHP this is referred to as a numerical array, or an array without defined keys.
- `object` A map of keys to values. This is equivalent to a JavaScript object. In PHP this is referred to as an associative array, or an array with defined keys.

A value's primitive type is specified using the `type` keyword. For example, this is how you would define a `string` value with JSON Schema.

```
array(
    'type' => 'string',
);
```

JSON Schema allows for defining values that can be multiple types. For example, this is how you would define a value that can be either a `string` or a `boolean`.

```
array(
    'type' => array( 'boolean', 'string' ),
);
```

Type Juggling

Because the WordPress REST API accepts URL form encoded data both in the `POST` body or as the query portion of the URL, many primtive types perform type juggling to convert these string values into their proper native types.

- `string` Only strings according to `is_string` are allowed.
- `null` Only a properly typed `null` is accepted. This means that submitting a null value in the URL or as a URL form encoded post body is not possible, a JSON request body must be used.
- `number` Floats, integers, and strings that pass `is_numeric` are allowed. Values will be casted to a `float`.

- `integer` Integers or strings that can be casted to a `float` with a fractional part that is equivalent to 0.
- `boolean` Booleans, the integers `0` and `1`, or the strings `"0"`, `"1"`, `"false"`, and `"true"`. `0` is treated as `false` and `1` is treated as `true`.
- `array` Numeric arrays according to `wp_is_numeric_array` or a string. If the string is comma separated it will be split into an array, otherwise it will be an array containing the string value. For example: `"red,yellow"` becomes `array( "red", "yellow" )` and `"blue"` becomes `array( "blue" )`.
- `object` An array, an `stdClass` object, an object implementing `JsonSerializable` or an empty string. Values will be converted to a native PHP array.

When using multiple types, types will be evaluated in the order they are specified. This can have an impact on the sanitized data received by your REST API endpoint. For instance, in the previous example, if the value submitted was `"1"`, it would be sanitized to the boolean `true` value. However, if the order was flipped, the value would remain as the string `"1"`.

The JSON Schema specification allows for defining schemas without a `type` field. The WordPress implementation however requires a `type` to be defined, and will issue a `_doing_it_wrong` notice if a type is ommitted.

Format

The seven primitive types are just that, primitive, so how do you define more complex value types? One of the ways to do that is using the `format` keyword. The `format` keyword allows for defining additional semantic-level validation of values that have a well-defined structure. For example, to require a date value, you would use the `date-time` format.

```
array(
    'type'   => 'string',
    'format' => 'date-time',
);
```

WordPress supports the following formats:

- `date-time` A date formatted according to RFC3339.
- `uri` A uri according to `esc_url_raw`.
- `email` An email address according to `is_email`.
- `ip` A v4 or v6 ip address.
- `uuid` A uuid of any version.
- `hex-color` A 3 or 6 character hex color with a leading `#`.

A value must match its format even if the value is an empty string. If allowing an “empty” value is desired, add `null` as a possible type. For example, the following schema would allow `127.0.0.1` or `null` as possible values.

```
array(
    'type'   => array( 'string', 'null' ),
    'format' => 'ip',
);
```

Strings

The `string` types supports three additional keywords.

minLength and maxLength

The `minLength` and `maxLength` keywords can be used to constrain the acceptable length of a string. Importantly multi-byte characters are counted as a single character and bounds are inclusive. For instance, given the following schema, `ab`, `abc`, and `abcd` are valid, while `a`, and `abcde` are invalid.

```
array(
    'type'      => 'string',
    'minLength' => 2,
    'maxLength' => 4,
);
```

The `exclusiveMinimum` and `exclusiveMaximum` keywords do not apply, they are only valid for numbers.

pattern

The JSON Schema keyword `pattern` can be used to validate that a string field matches a regular expression. For instance, given the following schema, `#123` would be valid, but `#abc` would not.

```
array(
    'type'    => 'string',
    'pattern' => '#[0-9]+',
);
```

The regex is not automatically anchored. Regex flags, for instance `/i` to make the match case insensitive are not supported. The JSON Schema RFC recommends limiting yourself to the following regex features so the schema can be interoperable between as many different programming languages as possible.

- individual Unicode characters, as defined by the JSON specification [RFC4627].
- simple character classes `[abc]`, range character classes `[a-z]`.
- complemented character classes `[^abc]`, `[^a-z]`.
- simple quantifiers: `+` (one or more), `*` (zero or more), `?` (zero or one), and their lazy versions `+?`, `*?`, `??`.
- range quantifiers: `{x}` (exactly x occurrences), `{x,y}` (at least x, at most y, occurrences), `{x,}` (x occurrences or more), and their lazy versions.
- the beginning-of-input `^` and end-of-input `$` anchors.
- simple grouping `(...)` and alternation `|`.

The pattern should be valid according to the ECMA 262 regex dialect.

Numbers

The `number` and `integer` types support four additional keywords.

minimum and maximum

The `minimum` and `maximum` keyword allow constraining the range of acceptable numbers. For example, `2` would be valid according to this schema, but `0` and `4` would not be.

```
array(
    'type' => 'integer',
    'minimum' => 1,
    'maximum' => 3,
);
```

JSON Schema also allows using the `exclusiveMinimum` and `exclusiveMaximum` keywords to denote that the value cannot equal the defined `minimum` or `maximum` respectively. For example, in this case only `2` would be an acceptable value.

```
array(
    'type'             => 'integer',
    'minimum'          => 1,
    'exclusiveMinimum' => true,
    'maximum'          => 3,
    'exclusiveMaximum' => true,
);
```

multipleOf

The `multipleOf` keyword allows for asserting an integer or number type is a multiple of the given number. For example, this schema will only accept even integers.

```
array(
    'type'       => 'integer',
    'multipleOf' => 2,
);
```

`multipleOf` also supports decimals. For example, this schema could be used to accept a percentage with a maximum of 1 decimal point.

```
array(
    'type'       => 'number',
    'minimum'    => 0,
    'maximum'    => 100,
    'multipleOf' => 0.1,
);
```

Arrays

Specifying a `type` of `array` requires data to be an array, but that is only half the validation story. You'll also want to enforce the format of each item in the array. This is done by specifying a JSON Schema that each array item must conform to using the `items` keyword. For example, the following schema requires an array of IP addresses.

```
array(
    'type'  => 'array',
    'items' => array(
        'type'   => 'string',
        'format' => 'ip',
    ),
);
```

This would pass validation.

```
[ "127.0.0.1", "255.255.255.255" ]
```

While this would fail validation.

```
[ "127.0.0.1", 5 ]
```

The `items` schema can be any schema, it could even be an array itself!

```
array(
    'type'  => 'array',
    'items' => array(
        'type'  => 'array',
        'items' => array(
            'type'   => 'string',
            'format' => 'hex-color',
        ),
    ),
);
```

This would pass validation.

```
[
  [ "#ff6d69", "#fecc50" ],
  [ "#0be7fb" ]
]
```

While this would fail validation.

```
[
  [ "#ff6d69", "#fecc50" ],
  "george"
]
```

minItems and maxItems

The `minItems` and `maxItems` keywords can be used to constrain the acceptable number of items included in the array. For instance, given the following schema, `[ "a" ]` and `[ "a", "b" ]` are valid, while `[]` and `[ "a", "b", "c" ]` are invalid.

```
array(
    'type'     => 'array',
    'minItems' => 1,
    'maxItems' => 2,
    'items'    => array(
        'type' => 'string',
    ),
);
```

The `exclusiveMinimum` and `exclusiveMaximum` keywords do not apply, they are only valid for the `number` and `integer` types.

uniqueItems

The `uniqueItems` keyword can be used to require that all items in the array are unique. For instance, given the following schema, `[ "a", "b" ]` is valid, while `[ "a", "a" ]` is not.

```
array(
    'type'        => 'array',
    'uniqueItems' => true,
    'items'       => array(
        'type' => 'string',
    ),
);
```

Uniqueness

Items of different types are considered unique, for instance, `"1"`, `1` and `1.0` are different values. When arrays are compared, the order of items matters. So the given array is considered to have all unique items.

```
[
  [ "a", "b" ],
  [ "b", "a" ]
]
```

When objects are compared, the order the members appear in does not matter. So the given array is considered to have duplicate items since the values are the same, they just appear in a different order.

```
[
  {
    "a": 1,
    "b": 2
  },
  {
    "b": 2,
    "a": 1
  }
]
```

Uniqueness is checked in both `rest_validate_value_from_schema` and `rest_sanitize_value_from_schema`. This is to prevent instances where items would be considered unique before sanitization is applied, but after sanitization the items would converge to identical values. Take for instance the following schema:

```
array(
    'type' => 'array',
    'uniqueItems' => true,
    'items' => array(
        'type' => 'string',
        'format' => 'uri',
    ),
);
```

A request with `[ "https://example.org/hello world", "https://example.org/hello%20world" ]` would pass validation because each string value is different. However, after `esc_url_raw` converted the space in the first url to `%20` the values would be identical. In this case `rest_sanitize_value_from_schema` would return an error. As such, you should always validate and sanitize parameters.

Objects

Specifying a `type` of `object` requires data to be an object, but that is only half the validation story. You'll also want to enforce the format of each object's property. This is done by specifying a map of property names to JSON Schemas that object members must conform to using the `properties` keyword. For example, the following schema requires an object where the property `name` is a string and `color` is a hex color.

```
array(
    'type'       => 'object',
    'properties' => array(
        'name'  => array(
            'type' => 'string',
        ),
        'color' => array(
            'type'   => 'string',
            'format' => 'hex-color',
        ),
    ),
);
```

This would pass validation.

```
{
  "name": "Primary",
  "color": "#ff6d69"
}
```

While this would fail validation.

```
{
  "name": "Primary",
  "color": "orange"
}
```

Required Properties

By default, any properties listed for an object are optional, so while it might be unexpected, the following would also pass validation for the previous schema.

```
{
  "name": "Primary"
}
```

There are two mechanisms for requiring a property to be provided.

Version 3 Syntax

While WordPress mostly follows JSON Schema Version 4, one way that it doesn't is with the syntax for defining required properties. The predominant way is using JSON Schema Version 3 syntax by adding the `required` keyword to each property's definition.

```
array(
    'type'       => 'object',
    'properties' => array(
        'name'  => array(
            'type'     => 'string',
            'required' => true,
        ),
        'color' => array(
            'type'     => 'string',
            'format'   => 'hex-color',
            'required' => true,
```

```
        ),
    ),
);
```

Version 4 syntax

WordPress also supports JSON Schema Version 4 required property syntax where the list of required properties for an object is defined as an array of property names. This can be particularly helpful when specifying that a meta value has a list of required properties. Given the following meta field.

```
register_post_meta( 'post', 'fixed_in', array(
    'type'         => 'object',
    'show_in_rest' => array(
        'single' => true,
        'schema' => array(
            'required'   => array( 'revision', 'version' ),
            'type'       => 'object',
            'properties' => array(
                'revision' => array(
                    'type' => 'integer',
                ),
                'version'  => array(
                    'type' => 'string',
                ),
            ),
        ),
    ),
) );
```

The following request would fail validation.

```
{
    "title": "Check required properties",
    "content": "We should check that required properties are provided",
    "meta": {
        "fixed_in": {
            "revision": 47089
        }
    }
}
```

If the `fixed_in` meta field was omitted entirely, no error would be generated. An object that defines a list of required properties does not indicate that the object itself is required to be submitted. Just that if the object is included, that the listed properties must also be included as well. Version 4 syntax is not supported for an endpoint's top level schema in `WP_REST_Controller::get_item_schema()`. Given the following schema, a user could successfully submit a request without the title or content properties. This is because the schema document is not itself used for validation, but instead transformed to a list of parameter definitions.

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

additionalProperties

Perhaps unintuitively, by default JSON Schema also allows providing additional properties that are not specified in the schema. As such, the following would pass validation.

```
{
  "name": "Primary",
  "color": "#ff6d69",
  "description": "The primary color to use in the theme."
}
```

This can be customized using the `additionalProperties` keyword. Setting `additionalProperties` to false will reject data that has unknown properties.

```
array(
    'type'                 => 'object',
    'additionalProperties' => false,
    'properties'           => array(
        'name'  => array(
```

```
            'type' => 'string',
        ),
        'color' => array(
            'type'   => 'string',
            'format' => 'hex-color',
        ),
    ),
);
```

The `additionalProperties` keyword can also be set to a schema of its own. This would require that the values for any unknown properties match the given schema. One way this can be helpful is when you want to accept a list of values that each have their own unique key. For example:

```
array(
    'type'                 => 'object',
    'properties'           => array(),
    'additionalProperties' => array(
        'type'       => 'object',
        'properties' => array(
            'name'  => array(
                'type'     => 'string',
                'required' => true,
            ),
            'color' => array(
                'type'     => 'string',
                'format'   => 'hex-color',
                'required' => true,
            ),
        ),
    ),
);
```

This would pass validation.

```
{
  "primary": {
    "name": "Primary",
    "color": "#ff6d69"
  },
  "secondary": {
    "name": "Secondary",
```

```
    "color": "#fecc50"
  }
}
```

While this would fail validation.

```
{
  "primary": {
    "name": "Primary",
    "color": "#ff6d69"
  },
  "secondary": "#fecc50"
}
```

patternProperties

The `patternProperties` keyword is similar to the `additionalProperties` keyword, but allows for asserting the property matches a regex pattern. The keyword is an object where each property is a regex pattern and its value is the JSON Schema used to validate properties that match that pattern. For example, this schema requires that each value is a hex color and the property must only contain “word” characters.

```
array(
  'type'                 => 'object',
  'patternProperties'    => array(
    '^\\w+$' => array(
      'type'   => 'string',
      'format' => 'hex-color',
    ),
  ),
  'additionalProperties' => false,
);
```

This would pass validation.

```
{
  "primary": "#ff6d69",
  "secondary": "#fecc50"
}
```

While this would fail validation.

```
{
  "primary": "blue",
  "$secondary": "#fecc50"
}
```

When the REST API validates the `patternProperties` schema, if a property doesn't match any of the patterns, the property will be allowed and not have any validation applied to its contents. While this may be confusing, it behaves the same as the `properties` keyword. If this logic isn't desired, add `additionalProperties` to the schema to disallow non-matching properties.

minProperties and maxProperties

The `minItems` and `maxItems` keywords can be used for the `array` type. The `minProperties` and `maxProperties` introduces this same functionality for the `object` type. This can be helpful when using `additionalProperties` to have a list of objects with unique keys.

```
array(
  'type'                 => 'object',
  'additionalProperties' => array(
    'type'   => 'string',
    'format' => 'hex-color',
  ),
  'minProperties'        => 1,
  'maxProperties'        => 3,
);
```

This would pass validation.

```
{
  "primary": "#52accc",
  "secondary": "#096484"
}
```

While this would fail validation.

```
{
  "primary": "#52accc",
  "secondary": "#096484",
  "tertiary": "#07526c"
}
```

The `exclusiveMinimum` and `exclusiveMaximum` keywords do not apply, they are only valid for the `number` and `integer` types.

Type agnostic keywords

oneOf and anyOf

These are advanced keywords that allow for the JSON Schema validator to choose one of many schemas to use when validating a value. The `anyOf` keyword allows for a value to match at least one of the given schemas. Whereas, the `oneOf` keyword requires the value match *exactly* one schema. For example, this schema allows for submitting an array of “operations” to an endpoint. Each operation can either be a “crop” or a “rotation”.

```
array(
    'type'  => 'array',
    'items' => array(
        'oneOf' => array(
            array(
                'title'      => 'Crop',
                'type'       => 'object',
                'properties' => array(
                    'operation' => array(
                        'type' => 'string',
                        'enum' => array(
                            'crop',
                        ),
                    ),
                    'x'         => array(
                        'type' => 'integer',
                    ),
                    'y'         => array(
                        'type' => 'integer',
                    ),
                ),
            ),
            array(
                'title'      => 'Rotation',
                'type'       => 'object',
                'properties' => array(
                    'operation' => array(
```

```
                        'type' => 'string',
                        'enum' => array(
                            'rotate',
                        ),
                    ),
                    'degrees'   => array(
                        'type'    => 'integer',
                        'minimum' => 0,
                        'maximum' => 360,
                    ),
                ),
            ),
        ),
    ),
);
```

The REST API will loop over each schema specified in the `oneOf` array and look for a match. If exactly one schema matches, then validation will succeed. If more than one schema matches, validation will fail. If no schemas match, then the validator will try to find the closest matching schema and return an appropriate error message.

> operations[0] is not a valid Rotation. Reason: operations[0][degrees] must be between 0 (inclusive) and 360 (inclusive)

To generate more helpful error messages, it is highly recommended to give each `oneOf` or `anyOf` schema a `title` property.

Changelog

WordPress 5.6

- Support the `multipleOf` JSON Schema keyword. r49063
- Support the `minProperties` and `maxProperties` JSON Schema keywords. r49053
- Support the `patternProperties` JSON Schema keyword. r49082
- Support the `anyOf` and `oneOf` JSON Schema keywords. r49246

WordPress 5.5

- Improve multi-type JSON Schema support. r48306
- Check required properties are provided when validating an object. r47809
- Only validate the `format` keyword if the `type` is a `string`. r48300
- Support the `uuid` JSON Schema format. 47753
- Support the `hex-color` JSON Schema format. r47450
- Support the `pattern` JSON Schema keyword. r47810
- Support the `minItems`, `maxItems`, and `uniqueItems` JSON Schema keywords. r47923 r48357
- Support the `minLength` and `maxLength` JSON Schema keywords. r47627

WordPress 5.4

- Support type juggling an empty string to an empty object. r47362

WordPress 5.3

- Support the `null` primitive type and implement basic multi-type handling. r46249
- Support validating `additionalProperties` against a schema. r45807

WordPress 4.9

- Support the `object` primitive type. r41727

□□□ #2
□ □□□ □□□ 2026-10-31 13:40:19 UTC
□ □□□ □□□ 2026-10-31 13:40:20 UTC