Extending The Rest Api

- Adding Custom Endpoints
- Adding REST API Support For Custom Content Types
- Controller Classes
- Modifying Responses
- Routes and Endpoints
- Schema

Adding Custom Endpoints

The WordPress REST API is more than just a set of default routes. It is also a tool for creating custom routes and endpoints. The WordPress front-end provides a default set of URL mappings, but the tools used to create them (e.g. the Rewrites API, as well as the query classes: `WP_Query`, `WP_User`, etc) are also available for creating your own URL mappings, or custom queries.

This document details how to create a new and completely custom route with its own endpoints. We'll first work through a short example and then expand it out to the full controller pattern as used internally.

Bare Basics

So you want to add custom endpoints to the API? Fantastic! Let's get started with a simple example. Let's start with a simple function that looks like this:

```
<?php
/**
 * Grab latest post title by an author!
 *
 * @param array $data Options for the function.
 * @return string|null Post title for the latest,
 * or null if none.
 */
function my_awesome_func( $data ) {
  $posts = get_posts( array(
    'author' => $data['id'],
  ) );

  if ( empty( $posts ) ) {
    return null;
  }

  return $posts[0]->post_title;
}
```

To make this available via the API, we need to register a route. This tells the API to respond to a given request with our function. We do this through a function called `register_rest_route`, which should be called in a callback on `rest_api_init` to avoid doing extra work when the API isn't loaded.

We need to pass in three things to `register_rest_route`: the namespace, the route we want, and the options. We'll come back to the namespace in a bit, but for now, let's pick `myplugin/v1`. We're going to have the route match anything with `/author/{id}`, where `{id}` is an integer.

```
<?php
add_action( 'rest_api_init', function () {
  register_rest_route( 'myplugin/v1', '/author/(?P<id>\d+)', array(
    'methods' => 'GET',
    'callback' => 'my_awesome_func',
  ) );
} );
```

Right now, we're only registering the one endpoint for the route. The term "route" refers to the URL, whereas "endpoint" refers to the function behind it that corresponds to a method *and* a URL (for more information, see the Glossary).

For example, if your site domain is `example.com` and you've kept the API path of `wp-json`, then the full URL would be `http://example.com/wp-json/myplugin/v1/author/(?P\d+)`.

[alert] On sites without pretty permalinks, the route is instead added to the URL as the `rest_route` parameter. For the above example, the full URL would then be `http://example.com/?rest_route=/myplugin/v1/author/(?P\d+)` [/alert]

Each route can have any number of endpoints, and for each endpoint, you can define the HTTP methods allowed, a callback function for responding to the request and a permissions callback for creating custom permissions. In addition you can define allowed fields in the request and for each field specify a default value, a sanitization callback, a validation callback, and whether the field is required.

Namespacing

Namespaces are the first part of the URL for the endpoint. They should be used as a vendor/package prefix to prevent clashes between custom routes. Namespaces allows for two plugins to add a route of the same name, with different functionality.

Namespaces in general should follow the pattern of `vendor/v1`, where `vendor` is typically your plugin or theme slug, and `v1` represents the first version of the API. If you ever need to break compatibility with new endpoints, you can then bump this to `v2`.

The above scenario, two routes with the same name, from two different plugins, requires all vendors to use a unique namespace. Failing to do so is analogous to a failure to use a vendor function prefix, class prefix and/ or class namespace in a theme or plugin, which is **very very bad**.

An added benefit of using namespaces is that clients can detect support for your custom API. The API index lists out the available namespaces on a site:

```
{
  "name": "WordPress Site",
  "description": "Just another WordPress site",
  "url": "http://example.com/",
  "namespaces": [
    "wp/v2",
    "vendor/v1",
    "myplugin/v1",
    "myplugin/v2",
  ]
}
```

If a client wants to check that your API exists on a site, they can check against this list. (For more information, see the Discovery guide.)

Arguments

By default, routes receive all arguments passed in from the request. These are merged into a single set of parameters, then added to the Request object, which is passed in as the first parameter to your endpoint:

```
<?php
function my_awesome_func( WP_REST_Request $request ) {
  // You can access parameters via direct array access on the object:
  $param = $request['some_param'];

  // Or via the helper method:
  $param = $request->get_param( 'some_param' );

  // You can get the combined, merged set of parameters:
  $parameters = $request->get_params();

  // The individual sets of parameters are also available, if needed:
```

```
  $parameters = $request->get_url_params();
  $parameters = $request->get_query_params();
  $parameters = $request->get_body_params();
  $parameters = $request->get_json_params();
  $parameters = $request->get_default_params();

  // Uploads aren't merged in, but can be accessed separately:
  $parameters = $request->get_file_params();
}
```

(To find out exactly how parameters are merged, check the source of `WP_REST_Request::get_parameter_order()`; the basic order is body, query, URL, then defaults.)

Normally, you'll get every parameter brought in unaltered. However, you can register your arguments when registering your route, which allows you to run sanitization and validation on these.

If the request has the `Content-type: application/json` header set and valid JSON in the body, `get_json_params()` will return the parsed JSON body as an associative array.

Arguments are defined as a map in the key `args` for each endpoint (next to your `callback` option). This map uses the name of the argument of the key, with the value being a map of options for that argument. This array can contain a key for `default`, `required`, `sanitize_callback` and `validate_callback`.

- `default`: Used as the default value for the argument, if none is supplied.
- `required`: If defined as true, and no value is passed for that argument, an error will be returned. No effect if a default value is set, as the argument will always have a value.
- `validate_callback`: Used to pass a function that will be passed the value of the argument. That function should return true if the value is valid, and false if not.
- `sanitize_callback`: Used to pass a function that is used to sanitize the value of the argument before passing it to the main callback.

Using `sanitize_callback` and `validate_callback` allows the main callback to act only to process the request, and prepare data to be returned using the `WP_REST_Response` class. By using these two callbacks, you will be able to safely assume your inputs are valid and safe when processing.

Taking our previous example, we can ensure the parameter passed in is always a number:

```
<?php
add_action( 'rest_api_init', function () {
  register_rest_route( 'myplugin/v1', '/author/(?P<id>\d+)', array(
    'methods' => 'GET',
    'callback' => 'my_awesome_func',
```

```
    'args' => array(
      'id' => array(
        'validate_callback' => function($param, $request, $key) {
          return is_numeric( $param );
        }
      ),
    ),
  ) );
} );
```

You could also pass in a function name to `validate_callback`, but passing certain functions like `is_numeric` directly will not only throw a warning about having extra parameters passed to it, but will also return `NULL` causing the callback function to be called with invalid data. We hope to eventually solve this problem in WordPress core.

We could also use something like `'sanitize_callback' => 'absint'` instead, but validation will throw an error, allowing clients to understand what they're doing wrong. Sanitization is useful when you would rather change the data being input rather than throwing an error (such as invalid HTML).

Return Value

After your callback is called, the return value is then converted to JSON, and returned to the client. This allows you to return basically any form of data. In our example above, we're returning either a string or null, which are automatically handled by the API and converted to JSON.

Like any other WordPress function, you can also return a `WP_Error` instance. This error information will be passed along to the client, along with a 500 Internal Service Error status code. You can further customise your error by setting the `status` option in the `WP_Error` instance data to a code, such as `400` for bad input data. Taking our example from before, we can now return an error instance:

```
<?php
/**
 * Grab latest post title by an author!
 *
 * @param array $data Options for the function.
 * @return string|null Post title for the latest,
 * or null if none.
 */
function my_awesome_func( $data ) {
  $posts = get_posts( array(
```

```
    'author' => $data['id'],
  ) );

  if ( empty( $posts ) ) {
    return new WP_Error( 'no_author', 'Invalid author', array( 'status' => 404 ) );
  }

  return $posts[0]->post_title;
}
```

When an author doesn't have any posts belonging to them, this will return a 404 Not Found error to the client:

```
HTTP/1.1 404 Not Found

[{
   "code": "no_author",
   "message": "Invalid author",
   "data": { "status": 404 }
}]
```

For more advanced usage, you can return a `WP_REST_Response` object. This object "wraps" normal body data, but allows you to return a custom status code, or custom headers. You can also add links to your response. The quickest way to use this is via the constructor:

```
<?php
$data = array( 'some', 'response', 'data' );

// Create the response object
$response = new WP_REST_Response( $data );

// Add a custom status code
$response->set_status( 201 );

// Add a custom header
$response->header( 'Location', 'http://example.com/' );
```

When wrapping existing callbacks, you should always use `rest_ensure_response()` on the return value. This will take raw data returned from an endpoint and automatically turn it into a `WP_REST_Response` for you. (Note that `WP_Error` is not converted to a `WP_REST_Response` to allow proper error handling.)

Importantly, a REST API route's callback should *always* return data; it shouldn't attempt to send the response body itself. This ensures that the additional processing that the REST API server does, like handling linking/embedding, sending headers, etc... takes place. In other words, don't call `die( wp_json_encode( $data ) );` or `wp_send_json( $data )`. As of WordPress 5.5, a `_doing_it_wrong` notice is issued if the `wp_send_json()` family of functions is used during a REST API request.

> Return a WP_REST_Response or WP_Error object from your callback when using the REST API.

Permissions Callback

You must also register a permissions callback for the endpoint. This is a function that checks if the user can perform the action (reading, updating, etc) before the real callback is called. This allows the API to tell the client what actions they can perform on a given URL without needing to attempt the request first.

This callback can be registered as `permission_callback`, again in the endpoint options next to your `callback` option. This callback should return a boolean or a `WP_Error` instance. If this function returns true, the response will be processed. If it returns false, a default error message will be returned and the request will not proceed with processing. If it returns a `WP_Error`, that error will be returned to the client.

The permissions callback is run after remote authentication, which sets the current user. This means you can use `current_user_can` to check if the user that has been authenticated has the appropriate capability for the action, or any other check based on current user ID. Where possible, you should always use `current_user_can`; instead of checking if the user is logged in (authentication), check whether they can perform the action (authorization).

Once you register a `permission_callback`, you will need to authenticate your requests (for example by including a nonce parameter) or you will receive a `rest_forbidden` error. See Authentication for more details.

Continuing with our previous example, we can make it so that only editors or above can view this author data. We can check a number of different capabilities here, but the best is `edit_others_posts`, which is really the core of what an editor is. To do this, we just need a callback here:

```
<?php
add_action( 'rest_api_init', function () {
  register_rest_route( 'myplugin/v1', '/author/(?P<id>\d+)', array(
    'methods' => 'GET',
```

```
    'callback' => 'my_awesome_func',
    'args' => array(
      'id' => array(
        'validate_callback' => 'is_numeric'
      ),
    ),
    'permission_callback' => function () {
      return current_user_can( 'edit_others_posts' );
    }
  ) );
} );
```

Note that the permission callback also receives the Request object as the first parameter, so you can do checks based on request arguments if you need to.

As of WordPress 5.5, if a `permission_callback` is not provided, the REST API will issue a `_doing_it_wrong` notice.

> The REST API route definition for myplugin/v1/author is missing the required permission_callback argument. For REST API routes that are intended to be public, use __return_true as the permission callback.

If your REST API endpoint is public, you can use `__return_true` as the permission callback.

```
<?php
add_action( 'rest_api_init', function () {
  register_rest_route( 'myplugin/v1', '/author/(?P<id>\d+)', array(
    'methods' => 'GET',
    'callback' => 'my_awesome_func',
    'permission_callback' => '__return_true',
  ) );
} );
```

Discovery

If you'd like to enable Resource Discovery for your custom endpoint, you can do so using the `rest_queried_resource_route` filter. For example, consider a custom query var `my-route` that contains the id of your custom resource. The following code snippet would add a discovery link whenever the `my-route` query var is used.

```
function my_plugin_rest_queried_resource_route( $route ) {
    $id = get_query_var( 'my-route' );
    if ( ! $route && $id ) {
        $route = '/my-ns/v1/items/' . $id;
    }

    return $route;
}
add_filter( 'rest_queried_resource_route', 'my_plugin_rest_queried_resource_route' );
```

Note: If your endpoint is describing a custom post type or custom taxonomy you will most likely want to be using the `rest_route_for_post` or `rest_route_for_term` filters instead.

The Controller Pattern

The controller pattern is a best practice for working with complex endpoints with the API.

It is recommended that you read "Extending Internal Classes" before reading this section. Doing so will familiarize you with the patterns used by the default routes, which is the best practice. While it is not required that the class you use to process your request extends the `WP_REST_Controller` class or a class that extends it, doing so allows you to inherit work done in those classes. In addition, you can rest assured that you're following best practices based on the controller methods you're using.

At their core, controllers are nothing more than a set of commonly named methods to match up with REST conventions, along with some handy helpers. Controllers register their routes in a `register_routes` method, respond to requests with `get_items`, `get_item`, `create_item`, `update_item` and `delete_item`, and have similarly named permission check methods. Following this pattern will ensure you don't miss any steps or functionality in your endpoints.

To use controllers, you first need to subclass the base controller. This gives you a base set of methods, ready for you to add your own behaviour into.

Once we've subclassed the controller, we need to instantiate the class to get it working. This should be done inside of a callback hooked into `rest_api_init`, which ensures we only instantiate the class when we need to. The normal controller pattern is to call `$controller->register_routes()` inside this callback, where the class can then register its endpoints.

Examples

The following is a "starter" custom route:

```php
<?php

class Slug_Custom_Route extends WP_REST_Controller {

  /**
   * Register the routes for the objects of the controller.
   */
  public function register_routes() {
    $version = '1';
    $namespace = 'vendor/v' . $version;
    $base = 'route';
    register_rest_route( $namespace, '/' . $base, array(
      array(
        'methods'             => WP_REST_Server::READABLE,
        'callback'            => array( $this, 'get_items' ),
        'permission_callback' => array( $this, 'get_items_permissions_check' ),
        'args'                => array(

        ),
      ),
      array(
        'methods'             => WP_REST_Server::CREATABLE,
        'callback'            => array( $this, 'create_item' ),
        'permission_callback' => array( $this, 'create_item_permissions_check' ),
        'args'                => $this->get_endpoint_args_for_item_schema( true ),
      ),
    ) );
    register_rest_route( $namespace, '/' . $base . '/(?P<id>[\d]+)', array(
      array(
        'methods'             => WP_REST_Server::READABLE,
        'callback'            => array( $this, 'get_item' ),
        'permission_callback' => array( $this, 'get_item_permissions_check' ),
        'args'                => array(
          'context' => array(
            'default' => 'view',
          ),
        ),
      ),
      array(
        'methods'             => WP_REST_Server::EDITABLE,
```

```
      'callback'            => array( $this, 'update_item' ),
      'permission_callback' => array( $this, 'update_item_permissions_check' ),
      'args'                => $this->get_endpoint_args_for_item_schema( false ),
    ),
    array(
      'methods'             => WP_REST_Server::DELETABLE,
      'callback'            => array( $this, 'delete_item' ),
      'permission_callback' => array( $this, 'delete_item_permissions_check' ),
      'args'                => array(
        'force' => array(
          'default' => false,
        ),
      ),
    ),
  ) );
  register_rest_route( $namespace, '/' . $base . '/schema', array(
    'methods'  => WP_REST_Server::READABLE,
    'callback' => array( $this, 'get_public_item_schema' ),
  ) );
}

/**
 * Get a collection of items
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|WP_REST_Response
 */
public function get_items( $request ) {
  $items = array(); //do a query, call another class, etc
  $data = array();
  foreach( $items as $item ) {
    $itemdata = $this->prepare_item_for_response( $item, $request );
    $data[] = $this->prepare_response_for_collection( $itemdata );
  }

  return new WP_REST_Response( $data, 200 );
}

/**
 * Get one item from the collection
```

```
   *
   * @param WP_REST_Request $request Full data about the request.
   * @return WP_Error|WP_REST_Response
   */
  public function get_item( $request ) {
    //get parameters from request
    $params = $request->get_params();
    $item = array();//do a query, call another class, etc
    $data = $this->prepare_item_for_response( $item, $request );

    //return a response or error based on some conditional
    if ( 1 == 1 ) {
      return new WP_REST_Response( $data, 200 );
    } else {
      return new WP_Error( 'code', __( 'message', 'text-domain' ) );
    }
  }

  /**
   * Create one item from the collection
   *
   * @param WP_REST_Request $request Full data about the request.
   * @return WP_Error|WP_REST_Response
   */
  public function create_item( $request ) {
    $item = $this->prepare_item_for_database( $request );

    if ( function_exists( 'slug_some_function_to_create_item' ) ) {
      $data = slug_some_function_to_create_item( $item );
      if ( is_array( $data ) ) {
        return new WP_REST_Response( $data, 200 );
      }
    }

    return new WP_Error( 'cant-create', __( 'message', 'text-domain' ), array( 'status' => 500
) );
  }

  /**
   * Update one item from the collection
```

```
   *
   * @param WP_REST_Request $request Full data about the request.
   * @return WP_Error|WP_REST_Response
   */
  public function update_item( $request ) {
    $item = $this->prepare_item_for_database( $request );

    if ( function_exists( 'slug_some_function_to_update_item' ) ) {
      $data = slug_some_function_to_update_item( $item );
      if ( is_array( $data ) ) {
        return new WP_REST_Response( $data, 200 );
      }
    }

    return new WP_Error( 'cant-update', __( 'message', 'text-domain' ), array( 'status' => 500
) );
  }

  /**
   * Delete one item from the collection
   *
   * @param WP_REST_Request $request Full data about the request.
   * @return WP_Error|WP_REST_Response
   */
  public function delete_item( $request ) {
    $item = $this->prepare_item_for_database( $request );

    if ( function_exists( 'slug_some_function_to_delete_item' ) ) {
      $deleted = slug_some_function_to_delete_item( $item );
      if ( $deleted ) {
        return new WP_REST_Response( true, 200 );
      }
    }

    return new WP_Error( 'cant-delete', __( 'message', 'text-domain' ), array( 'status' => 500
) );
  }

  /**
   * Check if a given request has access to get items
```

```
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|bool
 */
public function get_items_permissions_check( $request ) {
  //return true; <--use to make readable by all
  return current_user_can( 'edit_something' );
}

/**
 * Check if a given request has access to get a specific item
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|bool
 */
public function get_item_permissions_check( $request ) {
  return $this->get_items_permissions_check( $request );
}

/**
 * Check if a given request has access to create items
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|bool
 */
public function create_item_permissions_check( $request ) {
  return current_user_can( 'edit_something' );
}

/**
 * Check if a given request has access to update a specific item
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|bool
 */
public function update_item_permissions_check( $request ) {
  return $this->create_item_permissions_check( $request );
}

/**
```

```
 * Check if a given request has access to delete a specific item
 *
 * @param WP_REST_Request $request Full data about the request.
 * @return WP_Error|bool
 */
public function delete_item_permissions_check( $request ) {
  return $this->create_item_permissions_check( $request );
}

/**
 * Prepare the item for create or update operation
 *
 * @param WP_REST_Request $request Request object
 * @return WP_Error|object $prepared_item
 */
protected function prepare_item_for_database( $request ) {
  return array();
}

/**
 * Prepare the item for the REST response
 *
 * @param mixed $item WordPress representation of the item.
 * @param WP_REST_Request $request Request object.
 * @return mixed
 */
public function prepare_item_for_response( $item, $request ) {
  return array();
}

/**
 * Get the query params for collections
 *
 * @return array
 */
public function get_collection_params() {
  return array(
    'page'     => array(
      'description'       => 'Current page of the collection.',
      'type'              => 'integer',
```

```
        'default'           => 1,
        'sanitize_callback' => 'absint',
      ),
      'per_page' => array(
        'description'       => 'Maximum number of items to be returned in result set.',
        'type'              => 'integer',
        'default'           => 10,
        'sanitize_callback' => 'absint',
      ),
      'search'   => array(
        'description'       => 'Limit results to those matching a string.',
        'type'              => 'string',
        'sanitize_callback' => 'sanitize_text_field',
      ),
    );
  }
}
```

Adding REST API Support For Custom Content Types

The REST API can create routes for custom post types and custom taxonomies inside of the `wp/v2` namespace, using the same controllers as the default post type or taxonomy term controllers. Alternatively, you can use your own controllers and namespace. This document will cover using the default controllers for your custom content type's API routes. This is the easiest way and ensures the highest chance of compatibility with third parties.

Registering A Custom Post Type With REST API Support

When registering a custom post type, if you want it to be available via the REST API you should set `'show_in_rest' => true` in the arguments passed to `register_post_type`. Setting this argument to true will add a route in the `wp/v2` namespace.

```
/**
 * Register a book post type, with REST API support
 *
 * Based on example at: https://developer.wordpress.org/reference/functions/register_post_type
 */
add_action( 'init', 'my_book_cpt' );
function my_book_cpt() {
    $args = array(
      'public'       => true,
      'show_in_rest' => true,
      'label'        => 'Books'
    );
    register_post_type( 'book', $args );
}
```

You can optionally set the `rest_base` argument to change the base url, which will otherwise default to the post type's name. In the example below, "books" is used as the value of `rest_base`. This will

make the URL for the route `wp-json/wp/v2/books` instead of `wp-json/wp/v2/book/`, which would have been the default.

In addition, you can pass an argument for `rest_controller_class`. This class must be a subclass of `WP_REST_Controller`. By default, `WP_REST_Posts_Controller` is used as the controller. If you are using a custom controller, then you likely will not be within the `wp/v2` namespace.

Here is an example of registering a post type, with full labels, support for the REST API, a customized rest_base, and explicit registry of the default controller:

```
/**
 * Register a book post type, with REST API support
 *
 * Based on example at: https://developer.wordpress.org/reference/functions/register_post_type
 */
add_action( 'init', 'my_book_cpt' );
function my_book_cpt() {
  $labels = array(
    'name'               => _x( 'Books', 'post type general name', 'your-plugin-textdomain' ),
    'singular_name'      => _x( 'Book', 'post type singular name', 'your-plugin-textdomain' ),
    'menu_name'          => _x( 'Books', 'admin menu', 'your-plugin-textdomain' ),
    'name_admin_bar'     => _x( 'Book', 'add new on admin bar', 'your-plugin-textdomain' ),
    'add_new'            => _x( 'Add New', 'book', 'your-plugin-textdomain' ),
    'add_new_item'       => __( 'Add New Book', 'your-plugin-textdomain' ),
    'new_item'           => __( 'New Book', 'your-plugin-textdomain' ),
    'edit_item'          => __( 'Edit Book', 'your-plugin-textdomain' ),
    'view_item'          => __( 'View Book', 'your-plugin-textdomain' ),
    'all_items'          => __( 'All Books', 'your-plugin-textdomain' ),
    'search_items'       => __( 'Search Books', 'your-plugin-textdomain' ),
    'parent_item_colon'  => __( 'Parent Books:', 'your-plugin-textdomain' ),
    'not_found'          => __( 'No books found.', 'your-plugin-textdomain' ),
    'not_found_in_trash' => __( 'No books found in Trash.', 'your-plugin-textdomain' )
  );

  $args = array(
    'labels'             => $labels,
    'description'        => __( 'Description.', 'your-plugin-textdomain' ),
    'public'             => true,
    'publicly_queryable' => true,
    'show_ui'            => true,
    'show_in_menu'       => true,
```

```
    'query_var'          => true,
    'rewrite'            => array( 'slug' => 'book' ),
    'capability_type'    => 'post',
    'has_archive'        => true,
    'hierarchical'       => false,
    'menu_position'      => null,
    'show_in_rest'       => true,
    'rest_base'          => 'books',
    'rest_controller_class' => 'WP_REST_Posts_Controller',
    'supports'           => array( 'title', 'editor', 'author', 'thumbnail', 'excerpt',
'comments' )
  );

  register_post_type( 'book', $args );
}
```

If you are using a custom `rest_controller_class`, then the REST API is unable to automatically determine the route for a given post. In this case, you can use the `rest_route_for_post` filter to provide this information. This allows for your custom post type to be properly formatted in the Search endpoint and enables automated discovery links.

```
function my_plugin_rest_route_for_post( $route, $post ) {
    if ( $post->post_type === 'book' ) {
        $route = '/wp/v2/books/' . $post->ID;
    }

    return $route;
}
add_filter( 'rest_route_for_post', 'my_plugin_rest_route_for_post', 10, 2 );
```

Registering A Custom Taxonomy With REST API Support

Registering a custom taxonomy with REST API support is very similar to registering a custom post type: pass `'show_in_rest' => true` in the arguments passed to `register_taxonomy`. You may optionally pass `rest_base` to change the base url for the taxonomy's routes.

The default controller for taxonomies is `WP_REST_Terms_Controller`. You may modify this with the `rest_controller_class` if you choose to use a custom controller.

Here is an example of how to register a custom taxonomy with REST API support:

```
/**
 * Register a genre post type, with REST API support
 *
 * Based on example at: https://developer.wordpress.org/reference/functions/register_taxonomy/
 */
add_action( 'init', 'my_book_taxonomy', 30 );
function my_book_taxonomy() {

  $labels = array(
    'name'              => _x( 'Genres', 'taxonomy general name' ),
    'singular_name'     => _x( 'Genre', 'taxonomy singular name' ),
    'search_items'      => __( 'Search Genres' ),
    'all_items'         => __( 'All Genres' ),
    'parent_item'       => __( 'Parent Genre' ),
    'parent_item_colon' => __( 'Parent Genre:' ),
    'edit_item'         => __( 'Edit Genre' ),
    'update_item'       => __( 'Update Genre' ),
    'add_new_item'      => __( 'Add New Genre' ),
    'new_item_name'     => __( 'New Genre Name' ),
    'menu_name'         => __( 'Genre' ),
  );

  $args = array(
    'hierarchical'          => true,
    'labels'                => $labels,
    'show_ui'               => true,
    'show_admin_column'     => true,
    'query_var'             => true,
    'rewrite'               => array( 'slug' => 'genre' ),
    'show_in_rest'          => true,
    'rest_base'             => 'genre',
    'rest_controller_class' => 'WP_REST_Terms_Controller',
  );

  register_taxonomy( 'genre', array( 'book' ), $args );
```

```
}
```

If you are using a custom `rest_controller_class`, then the REST API is unable to automatically determine the route for a given term. In this case, you can use the `rest_route_for_term` filter to provide this information. This allows for your custom taxonomy to be properly formatted in the Search endpoint and enables automated discovery links.

```
function my_plugin_rest_route_for_term( $route, $term ) {
    if ( $term->taxonomy === 'genre' ) {
        $route = '/wp/v2/genre/' . $term->term_id;
    }

    return $route;
}
add_filter( 'rest_route_for_term', 'my_plugin_rest_route_for_term', 10, 2 );
```

Adding REST API Support To Existing Content Types

If you need to add REST API support for a custom post type or custom taxonomy you do not control, for example a theme or plugin you are using, you can use the `register_post_type_args` filter hook that exists since WordPress version 4.6.0.

```
/**
 * Add REST API support to an already registered post type.
 */
add_filter( 'register_post_type_args', 'my_post_type_args', 10, 2 );

function my_post_type_args( $args, $post_type ) {

    if ( 'book' === $post_type ) {
        $args['show_in_rest'] = true;

        // Optionally customize the rest_base or rest_controller_class
        $args['rest_base']             = 'books';
        $args['rest_controller_class'] = 'WP_REST_Posts_Controller';
    }
```

```
    return $args;
}
```

For custom taxonomies it is almost the same. You can use the `register_taxonomy_args` filter that exists since WordPress version 4.4.0.

```
/**
 * Add REST API support to an already registered taxonomy.
 */
add_filter( 'register_taxonomy_args', 'my_taxonomy_args', 10, 2 );

function my_taxonomy_args( $args, $taxonomy_name ) {

    if ( 'genre' === $taxonomy_name ) {
        $args['show_in_rest'] = true;

        // Optionally customize the rest_base or rest_controller_class
        $args['rest_base']             = 'genres';
        $args['rest_controller_class'] = 'WP_REST_Terms_Controller';
    }

    return $args;
}
```

Custom Link Relationships

Taxonomies & custom post types have a built-in association within WordPress, but what if you want to establish a link between two custom post types? This is not supported formally within WordPress itself, but we can create our own connections between arbitrary content types using the `_link` relation.

Controller Classes

Overview

To register a new REST route, you must specify a number of callback functions to control endpoint behavior such as how a request is fulfilled, how permissions checks are applied, and how the schema for your resource gets generated. While it is possible to declare all of these methods in an ordinary PHP file without any wrapping namespace or class, all functions declared in that manner coexist in the same global scope. If you decide to use a common function name for your endpoint logic like `get_items()` and another plugin (or another endpoint in your own plugin) also registers a function with that same name, PHP will fail with a fatal error because the function `get_items()` is being declared twice.

You can avoid this issue by naming your callback functions using a unique prefix such as `myplugin_myendpoint_` to avoid any potential conflics:

```
function myplugin_myendpoint_register_routes() { /* ... */ }
function myplugin_myendpoint_get_item() { /* ... */ }
function myplugin_myendpoint_get_item_schema() { /* ... */ }
// etcetera

add_action( 'rest_api_init', 'myplugin_myendpoint_register_routes' );
```

You may already be familiar with this approach because it is commonly used within theme `functions.php` files. However these prefixes are unnecessarily verbose, and several better options exist to group and encapsulate your endpoint's logic in a more maintainable way.

WordPress currently requires PHP 5.6 or greater. PHP 5.6 supports namespaces, which provide an easy way to encapsulate your endpoint's functionality. By declaring a `namespace` at the top of your endpoint's PHP file, all methods within that namespace will be declared within that namespace and will no longer conflict with global functions. You may then use shorter, more-readable names for your endpoint callbacks.

```
namespace MyPlugin\API\MyEndpoint;

function register_routes() { /* ... */ }
function get_item() { /* ... */ }
function get_item_schema() { /* ... */ }
```

```
// and so on

add_action( 'rest_api_init', __NAMESPACE__ . '\\register_routes' );
```

While these shorter function names are simpler to work with, they don't provide any other benefits over declaring global functions. For this reason the core REST API endpoints within WordPress are all implemented using a *controller class*.

The remainder of this page details how to write your own controller class and explains the advantages of doing so.

Controllers

A controller receives input (a `WP_REST_Request` object, in the case of the WordPress REST API) and generates response output as `WP_REST_Response` objects. Let's look at an example controller class:

```
class My_REST_Posts_Controller {

	// Here initialize our namespace and resource name.
	public function __construct() {
		$this->namespace     = '/my-namespace/v1';
		$this->resource_name = 'posts';
	}

	// Register our routes.
	public function register_routes() {
		register_rest_route( $this->namespace, '/' . $this->resource_name, array(
			// Here we register the readable endpoint for collections.
			array(
				'methods'   => 'GET',
				'callback'  => array( $this, 'get_items' ),
				'permission_callback' => array( $this, 'get_items_permissions_check' ),
			),
			// Register our schema callback.
			'schema' => array( $this, 'get_item_schema' ),
		) );
		register_rest_route( $this->namespace, '/' . $this->resource_name . '/(?P<id>[\d]+)', array(
			// Notice how we are registering multiple endpoints the 'schema' equates to an OPTIONS
request.
```

```
		array(
			'methods'   => 'GET',
			'callback'  => array( $this, 'get_item' ),
			'permission_callback' => array( $this, 'get_item_permissions_check' ),
		),
		// Register our schema callback.
		'schema' => array( $this, 'get_item_schema' ),
	) );
}

/**
 * Check permissions for the posts.
 *
 * @param WP_REST_Request $request Current request.
 */
public function get_items_permissions_check( $request ) {
	if ( ! current_user_can( 'read' ) ) {
		return new WP_Error( 'rest_forbidden', esc_html__( 'You cannot view the post resource.' ),
array( 'status' => $this->authorization_status_code() ) );
	}
	return true;
}

/**
 * Grabs the five most recent posts and outputs them as a rest response.
 *
 * @param WP_REST_Request $request Current request.
 */
public function get_items( $request ) {
	$args = array(
		'post_per_page' => 5,
	);
	$posts = get_posts( $args );

	$data = array();

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

```
□□foreach ( $posts as $post ) {
□□□$response = $this->prepare_item_for_response( $post, $request );
□□□$data[] = $this->prepare_response_for_collection( $response );
□□}

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

□/**
□ * Check permissions for the posts.
□ *
□ * @param WP_REST_Request $request Current request.
□ */
□public function get_item_permissions_check( $request ) {
□□if ( ! current_user_can( 'read' ) ) {
□□□return new WP_Error( 'rest_forbidden', esc_html__( 'You cannot view the post resource.' ),
array( 'status' => $this->authorization_status_code() ) );
□□}
□□return true;
□}

□/**
□ * Gets post data of requested post id and outputs it as a rest response.
□ *
□ * @param WP_REST_Request $request Current request.
□ */
□public function get_item( $request ) {
□□$id = (int) $request['id'];
□□$post = get_post( $id );

□□if ( empty( $post ) ) {
□□□return rest_ensure_response( array() );
□□}

□□$response = $this->prepare_item_for_response( $post, $request );

□□// Return all of our post response data.
```

```
        return $response;
    }

    /**
     * Matches the post data to the schema we want.
     *
     * @param WP_Post $post The comment object whose response is being prepared.
     */
    public function prepare_item_for_response( $post, $request ) {
        $post_data = array();

        $schema = $this->get_item_schema( $request );

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

        if ( isset( $schema['properties']['content'] ) ) {
            $post_data['content'] = apply_filters( 'the_content', $post->post_content, $post );
        }

        return rest_ensure_response( $post_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.
     */
    public function prepare_response_for_collection( $response ) {
        if ( ! ( $response instanceof WP_REST_Response ) ) {
            return $response;
        }

        $data = (array) $response->get_data();
```

```
□□$server = rest_get_server();

□□if ( method_exists( $server, 'get_compact_response_links' ) ) {
□□□$links = call_user_func( array( $server, 'get_compact_response_links' ), $response );
□□} else {
□□□$links = call_user_func( array( $server, 'get_response_links' ), $response );
□□}

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

□□return $data;
□}

□/**
□ * Get our sample schema for a post.
□ *
□ * @return array The sample schema for a post
□ */
□public function get_item_schema() {
□□if ( $this->schema ) {
□□□// Since WordPress 5.3, the schema can be cached in the $schema property.
□□□return $this->schema;
□□}
□
□□$this->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'                => 'post',
□□□'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,
```

```
        ),
        'content' => array(
            'description'  => esc_html__( 'The content for the object.', 'my-textdomain' ),
            'type'         => 'string',
        ),
    ),
);

    return $this->schema;
}

// Sets up the proper HTTP status code for authorization.
public function authorization_status_code() {

    $status = 401;

    if ( is_user_logged_in() ) {
        $status = 403;
    }

    return $status;
}
}

// Function to register our new routes from the controller.
function prefix_register_my_rest_routes() {
    $controller = new My_REST_Posts_Controller();
    $controller->register_routes();
}

add_action( 'rest_api_init', 'prefix_register_my_rest_routes' );
```

Benefits of Classes

This class contains all the same components you may have written using simple functions. The structure of a class gives us a convenient way to refer to related methods using the `$this->method_name()` syntax, but unlike a namespace the class also permits us to cache values and share logic.

In the `get_item_schema` method, note that we store the generated schema on the class as `$this->schema`. Class properties make it easy to cache these sorts of generated values. The introduction of schema caching in WordPress 5.3 increased the speed of some core REST API collection responses by up to 40%, so you should definitely consider following this pattern in your own controllers.

Class Inheritance & WP_REST_Controller

We've seen above how classes solve the global-function encapsulation issue, and how a class instance can be used to cache complex values to speed up response processing. The other major advantage of classes is the way in which class inheritance lets you share logic between multiple endpoints.

Our example class here did not extend any base class, but within WordPress core all endpoint controllers extend a single `abstract` controller class called `WP_REST_Controller`. Extending this class gives you access to a number of useful methods, including but not limited to:

- `prepare_response_for_collection()`: Prepare a response for insertion into a collection.
- `add_additional_fields_to_object()`: append any registered REST fields to your prepared response object.
- `get_fields_for_response()`: inspect the `_fields` query parameter to determine which response fields have been requested.
- `get_context_param()`: Retrieve the `context` parameter.
- `filter_response_by_context()`: Filters the response shape based on the provided context parameter.
- `get_collection_params()`: return a basic set of parameter definitions useful for collection endpoints.

Endpoint-specific methods like `get_item`, `register_routes`, and `update_item_permissions_check` are not fully implemented by the abstract class, and must be defined in your own class.

Visit the `WP_REST_Controller` class reference page for a complete list of this controller's methods.

It is important to note that `WP_REST_Controller` is implemented as an `abstract` class and only contains logic that is clearly needed in multiple classes. Inheritance couples your class to the base class it extends, and poorly-considered inheritance trees can make your endpoints much harder to maintain.

As an example, if you wrote a controller class for a posts endpoint (like the example above) and wanted to support custom post types as well, you should **NOT** extend your `My_REST_Posts_Controller` like this: `class My_CPT_REST_Controller extends My_REST_Posts_Controller`. Instead, you should either create an entirely separate base controller class for the shared logic, or make `My_REST_Posts_Controller` handle all available post types. Endpoint logic is subject to changing business requirements, and you don't want to have to change a number of unrelated controllers every time you update your base posts controller.

In most cases you will want to create a base controller class as either an `interface` or `abstract class` which each of your endpoint controllers can implement or extend, or to extend one of the core WordPress REST classes directly.

Internal WordPress REST API Classes

The WordPress REST API follows a deliberate design pattern for its internal classes, which may be categorized as either *infrastructure* or *endpoint* classes.

Endpoint classes encapsulate the functional logic necessary to perform CRUD operations on WordPress resources. WordPress exposes many REST API endpoints (such as `WP_REST_Posts_Controller`), but as discussed above all endpoints extend from a common base controller class:

- `WP_REST_Controller`: The base class for all WordPress core endpoints. This class is designed to represent a consistent pattern for manipulating WordPress resources. When interacting with an endpoint that implements `WP_REST_Controller`, a HTTP client can expect each endpoint to behave in a consistent way.

Infrastructure classes support the endpoint classes. They handle the logic for the WordPress REST API without performing any data transformation. The WordPress REST API implements three key infrastructure classes:

- `WP_REST_Server`: The main controller for the WordPress REST API. Routes are registered to the server within WordPress. When `WP_REST_Server` is called upon to serve a request, it determines which route is to be called, and passes the route callback a `WP_REST_Request` object. `WP_REST_Server` also handles authentication, and can perform request validation and permissions checks.
- `WP_REST_Request`: An object to represent the nature of the request. This object includes request details like request headers, parameters, and method, as well as the route. It can also perform request validation and sanitization.
- `WP_REST_Response`: An object to represent the nature of the response. This class extends `WP_HTTP_Response`, which includes headers, body, and status, and provides helper methods

like `add_link()` for adding linked media, and `query_navigation_headers()` for getting query navigtion headers.

Most types of API-driven application will not require you to extend or interact directly with the infrastructure layer, but if you are implementing your own REST API endpoints your application will likely benefit from one or more endpoint controller classes which extend `WP_REST_Controller`.

Modifying Responses

The default endpoints of the WordPress REST API are designed to return data by default that provides for the majority of sites and use cases, but there are often situations where you will need to access or expose additional data in the responses for various object types.

As with the rest of WordPress, the REST API is designed to be highly extensible to fit these needs. This guide details how to add additional data to the responses of default endpoints using the `register_rest_field` and `register_meta` functions. You can use these functions to add fields to any of the object types supported by the REST API. These custom fields can support both get and update operations.

Important Note about Changing Responses

[alert] Changing or removing data from core REST API endpoint responses can break plugins or WordPress core behavior, and should be avoided wherever possible. [/alert]

The API exposes many fields on API responses, including things you might not need, or which might not fit into how your site works. While it's tempting to modify or remove fields from REST API responses, this **will** cause problems with API clients that expect standard responses. This includes mobile clients, third party tools to help you manage your site, or wp-admin itself.

You may only need a small amount of data, but it's important to keep in mind that the API is about exposing an interface to all clients, not just the feature you're working on. Changing responses is dangerous.

Adding fields is not dangerous, so if you need to modify data, it's much better to duplicate the field instead with your modified data. Removing fields is never encouraged; if you need to get back a smaller subset of data, use the `_fields` parameter or work with contexts instead.

If you must remove fields from an existing context you should ensure that the behavior is **opt-in**, for example by providing a custom query parameter to trigger the field removal.

The API cannot prevent you from changing responses, but the code is structured to strongly discourage doing so. Internally, field registration is powered by filters, and these can be used if you absolutely have no other choice.

Using `register_rest_field` vs `register_meta`

There are two methods which can be used to add data to WordPress REST API responses, `register_rest_field` and `register_meta`.

`register_rest_field` may be used to add arbitrary fields to any REST API response, and can be used to both read and write data using the API. To register a new REST field you must provide your own callback functions to get or set the field's value, as well as manually specify your own schema definition for the field.

`register_meta` is used to whitelist an existing custom meta value for access through the REST API. By setting a meta field's `show_in_rest` parameter to `true`, that field's value will be exposed on a `.meta` key in the endpoint response, and WordPress will handle setting up the callbacks for reading and writing to that meta key. This is much simpler than `register_rest_field`, with one caveat:

[alert] Prior to WordPress 4.9.8, meta fields set to `show_in_rest` using `register_meta` are registered for all objects of a given type. If one custom post type shows a meta field, all custom post types will show that meta field. As of WordPress 4.9.8 it's possible to use `register_meta` with the `object_subtype` argument that allows one to reduce the usage of the meta key to a particular post type. [/alert]

Prior to WordPress 5.3, `register_meta` could only support scalar values (`string`, `integer`, `number` and `boolean`). WordPress 5.3 adds support for the `object` and `array` types.

Adding Custom Fields to API Responses

Using `register_rest_field`

The `register_rest_field` function is the most flexible way to add fields to REST API response objects. It accepts three parameters:

1. `$object_type`: The name of the object, as a string, or an array of the names of objects for which the field is being registered. This may be a core type like "post", "terms", "meta", "user" or "comment", but can also be the string name of a custom post type.
2. `$attribute`: The name of the field. This name will be used to define the key in the response object.

3. `$args`: An array with keys that define the callback functions used to retrieve the value of the field ('get_callback'), to update the value of the field ('update_callback'), and to define its schema ('schema').

Each of the keys of the $args array is optional, but if not used, that capability will not be added. This means that you may specify a callback function for reading the value and omit the update callback to make that field read-only if desired.

Fields should be registered at the `rest_api_init` action. Using this action rather than `init` will prevent the field registration from happening during requests to WordPress that do not use the REST API.

Examples

Read and write an extra field in comment responses

```
<?php
add_action( 'rest_api_init', function () {
    register_rest_field( 'comment', 'karma', array(
        'get_callback' => function( $comment_arr ) {
            $comment_obj = get_comment( $comment_arr['id'] );
            return (int) $comment_obj->comment_karma;
        },
        'update_callback' => function( $karma, $comment_obj ) {
            $ret = wp_update_comment( array(
                'comment_ID'    => $comment_obj->comment_ID,
                'comment_karma' => $karma
            ) );
            if ( false === $ret ) {
                return new WP_Error(
                  'rest_comment_karma_failed',
                  __( 'Failed to update comment karma.' ),
                  array( 'status' => 500 )
                );
            }
            return true;
        },
        'schema' => array(
            'description' => __( 'Comment karma.' ),
            'type'        => 'integer'
```

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

This example illustrates adding a field called `karma` to the response for posts. It works because the `comment_karma` field exists, but is unused by core. Note that an actual implementation of comment karma would need to use a separate endpoint.

This is a basic example; carefully consider what permissions checks or error handling may be required for your specific field.

How `register_rest_field` Works

The global variable `$wp_rest_additional_fields` is used internally by the REST API infrastructure to hold the response fields to be added to each object type. The REST API provides `register_rest_field` as a utility function for adding to this global variable. Adding to the global variable directly should be avoided to ensure forward-compatibility.

For each object type — posts, or users, terms, comments, etc. — `$wp_rest_additional_fields` contains an array of field definitions containing the callbacks used to retrieve or update the field's value.

Working with registered meta in the REST API

The `register_meta` function simplifies the process of defining a meta field for a particular object type. By setting `'show_in_rest' => true` when registering a new meta key, that key will be accessible through the REST API.

Read and write a post meta field in post responses

```
<?php
// The object type. For custom post types, this is 'post';
// for custom comment types, this is 'comment'. For user meta,
// this is 'user'.
$object_type = 'post';
```

```
$meta_args = array( // Validate and sanitize the meta value.
    // Note: currently (4.7) one of 'string', 'boolean', 'integer',
    // 'number' must be used as 'type'. The default is 'string'.
    'type'         => 'string',
    // Shown in the schema for the meta key.
    'description'  => 'A meta key associated with a string meta value.',
    // Return a single value of the type.
    'single'       => true,
    // Show in the WP REST API response. Default: false.
    'show_in_rest' => true,
);
register_meta( $object_type, 'my_meta_key', $meta_args );
```

This example shows how to allow reading and writing of a post meta field. This will allow that field to be updated via a POST request to `wp-json/wp/v2/posts/<post-id>` or created along with a post via a POST request to `wp-json/wp/v2/posts/`.

Note that for meta fields registered on custom post types, the post type must have `custom-fields` support. Otherwise the meta fields will not appear in the REST API.

Post Type Specific Meta

WordPress 4.9.8 adds support for registering meta for a specific post type or taxonomy by using the `register_post_meta` and `register_term_meta` functions. They follow the same rules as `register_meta` but accept a post type or taxonomy as their first parameter instead of an object type. The following code would register the `my_meta_key` example above, but only for the `page` custom post type.

```
$meta_args = array(
    'type'         => 'string',
    'description'  => 'A meta key associated with a string meta value.',
    'single'       => true,
    'show_in_rest' => true,
);
register_post_meta( 'page', 'my_meta_key', $meta_args );
```

Object Meta Type

WordPress 5.3 adds support for using the `object` type when registering meta. Importantly `object` refers to a JSON object, this is equivalent to an associative array in PHP.

When registering `object` meta, setting the `type` to `object` is not sufficient, you also need to inform WordPress of what properties to allow. This is done by writing a JSON Schema when registering the metadata.

For instance, the following code sample registers a post meta field called “release” that accepts the given JSON data.

```
{
  "meta": {
    "release": {
      "version": "5.2",
      "artist": "Jaco"
    }
  }
}
```

```
register_post_meta(
	'post',
	'release',
	array(
		'single'       => true,
		'type'         => 'object',
		'show_in_rest' => array(
			'schema' => array(
				'type'       => 'object',
				'properties' => array(
					'version' => array(
						'type' => 'string',
					),
					'artist'  => array(
						'type' => 'string',
					),
				),
			),
		),
	)
);
```

Notice that `show_in_rest` becomes an array instead of `true` and a Json Schema is specified for the `schema` key. Each property is then listed in the `properties` array. At a minimum, each property must specify a `type`, but any JSON Schema keyword that `rest_validate_value_from_schema`

understands can be used as well.

Additional Properties

By default, the list of properties is a strict whitelist. If a property is sent in the request that is not listed, the REST API will return an error: `your_property is not a valid property of Object.`. If you don't know the property names ahead of time the `additionalProperties` keyword can be used. `additionalProperties` accepts a JSON Schema that is used to validate the unknown properties. For instance, to enforce that any additional properties are numbers, the following code can be used.

```
{
  "meta": {
    "release": {
      "version": "5.2",
      "artist": "Jaco",
      "unknown_field": 5.3
    }
  }
}
```

```
register_post_meta(
	'post',
	'version',
	array(
		'single'       => true,
		'type'         => 'object',
		'show_in_rest' => array(
			'schema' => array(
				'type'                 => 'object',
				'properties'           => array(
					'version' => array(
						'type' => 'string',
					),
					'artist'  => array(
						'type' => 'string',
					),
				),
				'additionalProperties' => array(
					'type' => 'number',
				),
			),
		),
```

```
    ),
  )
);
```

`additionalProperties` can be set to true to allow unknown properties of any format, but this is not recommended.

Array Meta Type

WordPress 5.3 also adds support for using the `array` type. Importantly `array` refers to a JSON array, this is equivalent to a numeric array in PHP.

When registering `array` meta, setting the `type` to `array` is not sufficient, you need to inform WordPress of what the expected format is of the items in the array. This is done by writing a JSON Schema entry when registering the metadata.

If you do not provide this value, `register_meta` will return false and issue the following warning: `When registering an "array" meta type to show in the REST API, you must specify the schema for each array item in "show_in_rest.schema.items".`

The following code sample registers a post meta field called “projects” that contains a list of project names. It accepts the given JSON data.

```
{
  "meta": {
    "projects": [
      "WordPress",
      "BuddyPress"
    ]
  }
}
```

```
register_post_meta(
  'post',
  'projects',
  array(
    'single'       => true,
    'type'         => 'array',
    'show_in_rest' => array(
      'schema' => array(
        'type'  => 'array',
```

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

Notice that again `show_in_rest` becomes an array instead of `true` and a JSON Schema is specified for the `schema` key. The `items` keyword is used to define the JSON schema to validate each array member against. It can be a scalar type like `string` or a complex type like `object`.

For instance, to accept the given JSON data, the following meta registration would be used.

```
{
  "meta": {
    "projects": [
      {
        "name": "WordPress",
        "website": "https://wordpress.org"
      },
      {
        "name": "BuddyPress",
        "website": "https://buddypress.org"
      }
    ]
  }
}
```

```
register_post_meta(
□'post',
□'projects',
□array(
□□'single'       => true,
□□'type'         => 'array',
□□'show_in_rest' => array(
□□□'schema' => array(
□□□□'items' => array(
□□□□□'type'       => 'object',
□□□□□'properties' => array(
□□□□□□'name'    => array(
```

```
□□□□□□□□'type' => 'string',
□□□□□□□),
□□□□□□'website' => array(
□□□□□□□□'type'   => 'string',
□□□□□□□□'format' => 'uri',
□□□□□□□),
□□□□□),
□□□□),
□□□),
□□),
□)
);
```

[alert]The `array` type enforces that the array keys are sequential integers starting from 0. The array will be reindexed using `array_values`.[/alert]

Non-Single Metadata

Non-single meta fields have an array of values per object, instead of one value per object. Each of those values is stored in a separate row in the meta table.

The `array` and `object` types can be used with non-single meta fields as well. For example, if the “release” meta key from earlier had `single` set to `false`, the following JSON data could be accepted.

```
{
  "meta": {
    "release": [
      {
        "version": "5.2",
        "artist": "Jaco"
      },
      {
        "version": "5.1",
        "artist": "Betty"
      }
    ]
  }
}
```

This would result in two meta database rows. The first containing `{ "version": "5.2", "artist": "Jaco" }` and the second containing `{ "version": "5.1", "artist": "Betty" }`.

Similarly, the following data would be accepted for the “projects” example if it had set `single` to `false`.

```
{
  "meta": {
    "projects": [
      [
        "WordPress",
        "BuddyPress"
      ],
      [
        "bbPress"
      ]
    ]
  }
}
```

This would result in two meta database rows. The first containing `[ "WordPress", "BuddyPress" ]` and the second containing `[ "bbPress" ]`.

Invalid Stored Values

If the existing value for a meta field does not validate against the registered type and schema, the value for that meta field will be returned as null. If that null value is passed back to the API when doing an update request, you'll receive a `rest_invalid_stored_value` error: `The %s property has an invalid stored value, and cannot be updated to null.`. To fix this, either update the meta key with a valid value or omit that property from your request.

Default Metadata Values

WordPress 5.5 adds official support for specifying a default value for metadata if no value is defined. For example, with this code snippet the `price` meta field will be set to `0.00` in the REST API response if a value is not yet.

```
register_post_meta(
     'product',
     'price',
```

```
    array(
        'single'  => true,
        'type'    => 'string',
        'default' => '0.00',
    )
);
```

Adding Links to the API Response

WordPress generates a list of links associated with the queried resource to make it easier to navigate to related resources.


```
{
 "_links": {
    "self": [
      {
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/posts/28312"
      }
    ],
    "collection": [
      {
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/posts"
      }
    ],
    "author": [
      {
        "embeddable": true,
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/users/8670591"
      }
    ],
    "replies": [
      {
        "embeddable": true,
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/comments?post=28312"
      }
    ],
    "wp:term": [
      {
```

```
        "taxonomy": "category",
        "embeddable": true,
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/categories?post=28312"
      },
      {
        "taxonomy": "post_tag",
        "embeddable": true,
        "href": "https://make.wordpress.org/core/wp-json/wp/v2/tags?post=28312"
      }
    ]
  }
}
```

A sample of links from a Make.WordPress.org post

While these links will appear under the `_links` property in the JSON response object, it is not stored in `WP_REST_Response::$data` or accessible via `WP_REST_Response::get_data()`. Instead, the server will append the link data to the response right before echoing the response data.

Custom links can be added to the response by using the `WP_REST_Response::add_link()` method. This method accepts three parameters, the link relation, the URL and optionally a list of link attributes. For example, to add the `author` and `wp:term` link.

```
<?php
$response->add_link( 'author', rest_url( "/wp/v2/users/{$post->post_author}" ) );

$response->add_link( 'https://api.w.org/term', add_query_arg( 'post', $post->ID, rest_url(
"/wp/v2/{$tax_base}" ) ) );
```

The link relation MUST be either a registered link relation from the IANA or a URI that is under your control.

`author` is a registered link relation described as "the context's author", we use that to point to the WordPress user who wrote the post. No link relation exists that describes the terms associated with a post, so WordPress uses the `https://api.w.org/term` URL. This is transformed to `wp:term` when generating the response by using a CURIE.

The third parameter of `add_link()` is a list of custom attributes. The `embeddable` attribute can be used to include the linked resource appears in the `_embedded` section of the repsonse when using the `_embed` query parameter. If multiple links are added with the same relation, the embedded responses will be in the same order the links were added in.

```php
<?php
$response->add_link( 'author', rest_url( "/wp/v2/users/{$post->post_author}" ), array(
'embeddable' => true,
) );
$response->add_link( 'author', rest_url( "/wp/v2/users/{$additional_author}" ), array(
'embeddable' => true,
) );
```

A sample implementation of linking to multi-author posts.

```
{
  "_links": {
    "author": [
      {
        "embeddable": true,
        "href": "https://yourwebsite.com/wp-json/wp/v2/users/1"
      },
      {
        "embeddable": true,
        "href": "https://yourwebsite.com/wp-json/wp/v2/users/2"
      }
    ]
  },
  "_embedded": {
    "author": [
      {
        "id": 1,
        "name": "Primary Author"
      },
      {
        "id": 2,
        "name": "Secondary Author"
      }
    ]
  }
}
```

The order links are added is maintained.

Registering a CURIE

WordPress version 4.5 introduced support for Compact URIs, or CURIEs. This makes it possible to reference links by a much simpler identifier than the full URL which could easily be quite lengthy.

A CURIE is registered using the `rest_response_link_curies` filter. For example.

```
<?php
function my_plugin_prefix_register_curie( $curies ) {

	$curies[] = array(
		'name'      => 'my_plugin',
		'href'      => 'https://api.mypluginurl.com/{rel}',
		'templated' => true,
	);

	return $curies;
}
```

This will convert link URLs from `https://api.mypluginurl.com/my_link` to `my_plugin:my_link` in the API response. The full URL must still be used when adding links using `WP_REST_Response::add_link`.

Routes and Endpoints

Overview

The REST API provides us a way to match URIs to various resources in our WordPress install. By default, if you have pretty permalinks enabled, the WordPress REST API "lives" at `/wp-json/`. At our WordPress site `https://ourawesomesite.com`, we can access the REST API's index by making a `GET` request to `https://ourawesomesite.com/wp-json/`. The index provides information regarding what routes are available for that particular WordPress install, along with what HTTP methods are supported and what endpoints are registered.

Routes vs Endpoints

Endpoints are functions available through the API. This can be things like retrieving the API index, updating a post, or deleting a comment. Endpoints perform a specific function, taking some number of parameters and return data to the client.

A route is the "name" you use to access endpoints, used in the URL. A route can have multiple endpoints associated with it, and which is used depends on the HTTP verb.

For example, with the URL `http://example.com/wp-json/wp/v2/posts/123`:

- The "route" is `wp/v2/posts/123` - The route doesn't include `wp-json` because `wp-json` is the base path for the API itself.
- This route has 3 endpoints:
  - `GET` triggers a `get_item` method, returning the post data to the client.
  - `PUT` triggers an `update_item` method, taking the data to update, and returning the updated post data.
  - `DELETE` triggers a `delete_item` method, returning the now-deleted post data to the client.

[alert] On sites without pretty permalinks, the route is instead added to the URL as the `rest_route` parameter. For the above example, the full URL would then be `http://example.com/?rest_route=/wp/v2/posts/123` [/alert]

Creating Endpoints

If we wanted to create an endpoint that would return the phrase "Hello World, this is the WordPress REST API" when it receives a GET request, we would first need to register the route for that endpoint. To register routes you should use the `register_rest_route()` function. It needs to be called on the `rest_api_init` action hook. `register_rest_route()` handles all of the mapping for routes to endpoints. Let's try to create a "Hello World, this is the WordPress REST API" route.

```
/**
 * This is our callback function that embeds our phrase in a WP_REST_Response
 */
function prefix_get_endpoint_phrase() {
    // rest_ensure_response() wraps the data we want to return into a WP_REST_Response, and
ensures it will be properly returned.
    return rest_ensure_response( 'Hello World, this is the WordPress REST API' );
}

/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_example_routes() {
    // register_rest_route() handles more arguments but we are going to stick to the basics
for now.
    register_rest_route( 'hello-world/v1', '/phrase', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable
endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched
by the WP_REST_Server class.
        'callback' => 'prefix_get_endpoint_phrase',
    ) );
}

add_action( 'rest_api_init', 'prefix_register_example_routes' );
```

The first argument passed into `register_rest_route()` is the namespace, which provides us a way to group our routes. The second argument passed in is the resource path, or resource base. For our example, the resource we are retrieving is the "Hello World, this is the WordPress REST API" phrase. The third argument is an array of options. We specify what methods the endpoint can use and what callback should happen when the endpoint is matched (more things can be done but these are the fundamentals).

The third argument also allows us to provide a permissions callback, which can restrict access for the endpoint to only certain users. The third argument also offers a way to register arguments for

the endpoint so that requests can modify the response of our endpoint. We will get into those concepts in the endpoints section of this guide.

When we go to `https://ourawesomesite.com/wp-json/hello-world/v1/phrase` we can now see our REST API greeting us kindly. Let's take a look at routes a bit more in depth.

Routes

Routes in the REST API are represented by URIs. The route itself is what is tacked onto the end of `https://ourawesomesite.com/wp-json`. The index route for the API is `/` which is why `https://ourawesomesite.com/wp-json/` returns all of the available information for the API. All routes should be built onto this route, the `wp-json` portion can be changed, but in general, it is advised to keep it the same.

We want to make sure that our routes are unique. For instance we could have a route for books like this: `/books`. Our books route would now live at `https://ourawesomesite.com/wp-json/books`. However, this is not a good practice as we would end up polluting potential routes for the API. What if another plugin we wanted to register a books route as well? We would be in big trouble in that case, as the two routes would conflict with each other and only one could be used. The fourth parameter to `register_rest_route()` is a boolean for whether the route should override an existing route.

The override parameter does not really solve our problem either, as both routes could override or we would want to use both routes for different things. This is where using namespaces for our routes comes in.

Namespaces

It is extremely important to add namespaces to your routes. The "core" endpoints use the `wp/v2` namespace.

[warning]**Do not place anything into the `wp` namespace unless you are making endpoints with the intention of merging them into core.**[/warning]

There are some key things to take notice of in the core endpoint namespace. The first part of the namespace is `wp`, which represents the vendor name; WordPress. For our plugins we will want to come up with unique names for what we call the vendor portion of the namespace. In the example above we used `hello-world`.

Following the vendor portion is the version portion of the namespace. The "core" endpoints utilize `v2` to represent version 2 of the WordPress REST API. If you are writing a plugin, you can maintain backwards compatibility of your REST API endpoints by simply creating new endpoints and bumping up the version number you provide. This way both the original `v1` and `v2` endpoints can

be accessed.

The part of the route that follows the namespace is the resource path.

Resource Paths

The resource path should signify what resource the endpoint is associated with. In the example we used above, we used the word `phrase` to signify that the resource we are interacting with is a phrase. To avoid any collisions, each resource path we register should also be unique within a namespace. Resource paths should be used to define different resource routes within a given namespace.

Let's say we have a plugin that handles some basic eCommerce functionality. We will have two main resource types orders, and products. Orders are a request for product(s) but they are not the product themselves. The same concept applies to products. Although these resources are related they are not the same thing and each should live in a separate resource paths. Our routes will end up looking something like this for our eCommerce plugin: `/my-shop/v1/orders` and `/my-shop/v1/products`.

Using routes like this, we would want each to return a collection of orders or products. What if we wanted to grab a specific product by ID, we would need to use path variables in our routes.

Path Variables

Path variables enable us to add dynamic routes. To expand on our eCommerce routes, we could register a route to grab individual products.

```
/**
 * This is our callback function to return our products.
 *
 * @param WP_REST_Request $request This function accepts a rest request to process data.
 */
function prefix_get_products( $request ) {
    // In practice this function would fetch the desired data. Here we are just making stuff
up.
    $products = array(
        '1' => 'I am product 1',
        '2' => 'I am product 2',
        '3' => 'I am product 3',
    );
```

```
    return rest_ensure_response( $products );
}

/**
 * This is our callback function to return a single product.
 *
 * @param WP_REST_Request $request This function accepts a rest request to process data.
 */
function prefix_get_product( $request ) {
    // In practice this function would fetch the desired data. Here we are just making stuff up.
    $products = array(
        '1' => 'I am product 1',
        '2' => 'I am product 2',
        '3' => 'I am product 3',
    );

    // Here we are grabbing the 'id' path variable from the $request object. WP_REST_Request implements ArrayAccess, which allows us to grab properties as though it is an array.
    $id = (string) $request['id'];

    if ( isset( $products[ $id ] ) ) {
        // Grab the product.
        $product = $products[ $id ];

        // Return the product as a response.
        return rest_ensure_response( $product );
    } else {
        // Return a WP_Error because the request product was not found. In this case we return a 404 because the main resource was not found.
        return new WP_Error( 'rest_product_invalid', esc_html__( 'The product does not exist.', 'my-text-domain' ), array( 'status' => 404 ) );
    }

    // If the code somehow executes to here something bad happened return a 500.
    return new WP_Error( 'rest_api_sad', esc_html__( 'Something went horribly wrong.', 'my-text-domain' ), array( 'status' => 500 ) );
}

/**
```

```
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_product_routes() {
    // Here we are registering our route for a collection of products.
    register_rest_route( 'my-shop/v1', '/products', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched by the WP_REST_Server class.
        'callback' => 'prefix_get_products',
    ) );
    // Here we are registering our route for single products. The (?P<id>[\d]+) is our path variable for the ID, which, in this example, can only be some form of positive number.
    register_rest_route( 'my-shop/v1', '/products/(?P<id>[\d]+)', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched by the WP_REST_Server class.
        'callback' => 'prefix_get_product',
    ) );
}

add_action( 'rest_api_init', 'prefix_register_product_routes' );
```

The above example covers a lot. The important part to note is that in the second route we register, we add on a path variable `/(?P<id>[\d]+)` to our resource path `/products`. The path variable is a regular expression. In this case it uses `[\d]+` to signify that should be any numerical character at least once. If you are using numeric IDs for your resources, then this is a great example of how to use a path variable. When using path variables, we now have to be careful around what can be matched as it is user input.

The regex luckily will filter out anything that is not numerical. However, what if the product for the requested ID doesn't exist. We need to do error handling. You can see the basic way we are handling errors in the code example above. When you return a `WP_Error` in your endpoint callbacks the API server will automatically handle serving the error to the client.

Although this section is about routes, we have covered quite a bit about endpoints. Endpoints and routes are interrelated, but they definitely have distinctions.

Endpoints

Endpoints are the destination that a route needs to map to. For any given route, you could have a number of different endpoints registered to it. We will expand on our fictitious eCommerce plugin, to better show the distinction between routes and endpoints. We are going to create two endpoints that exist at the `/wp-json/my-shop/v1/products/` route. One endpoint uses the HTTP verb `GET` to get products, and the other endpoint uses the HTTP verb `POST` to create a new product.

```
/**
 * This is our callback function to return our products.
 *
 * @param WP_REST_Request $request This function accepts a rest request to process data.
 */
function prefix_get_products( $request ) {
    // In practice this function would fetch the desired data. Here we are just making stuff
up.
    $products = array(
        '1' => 'I am product 1',
        '2' => 'I am product 2',
        '3' => 'I am product 3',
    );

    return rest_ensure_response( $products );
}

/**
 * This is our callback function to return a single product.
 *
 * @param WP_REST_Request $request This function accepts a rest request to process data.
 */
function prefix_create_product( $request ) {
    // In practice this function would create a product. Here we are just making stuff up.
   return rest_ensure_response( 'Product has been created' );
}

/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_product_routes() {
```

```
    // Here we are registering our route for a collection of products and creation of
products.
    register_rest_route( 'my-shop/v1', '/products', array(
        array(
            // By using this constant we ensure that when the WP_REST_Server changes, our
readable endpoints will work as intended.
            'methods'  => WP_REST_Server::READABLE,
            // Here we register our callback. The callback is fired when this endpoint is
matched by the WP_REST_Server class.
            'callback' => 'prefix_get_products',
        ),
        array(
            // By using this constant we ensure that when the WP_REST_Server changes, our
create endpoints will work as intended.
            'methods'  => WP_REST_Server::CREATABLE,
            // Here we register our callback. The callback is fired when this endpoint is
matched by the WP_REST_Server class.
            'callback' => 'prefix_create_product',
        ),
    ) );
}

add_action( 'rest_api_init', 'prefix_register_product_routes' );
```

Depending on what HTTP Method we use for the route `/wp-json/my-shop/v1/products`, we are matched to a different endpoint and a different callback is fired. When we use `POST` we trigger the `prefix_create_product()` callback, and when we use `GET` we trigger the `prefix_get_products()` callback.

There are a number of different HTTP methods and the REST API can make use of any of them.

HTTP Methods

HTTP methods are sometimes referred to as HTTP verbs. They are simply just different ways to communicate via HTTP. The main ones used by the WordPress REST API are:

- `GET` should be used for retrieving data from the API.
- `POST` should be used for creating new resources (i.e users, posts, taxonomies).
- `PUT` should be used for updating resources.
- `DELETE` should be used for deleting resources.
- `OPTIONS` should be used to provide context about our resources.

It is important to note that these methods are not supported by every client, as they were introduced in HTTP 1.1. Luckily, the API provides a workaround for these unfortunate cases. If you want to delete a resource but can't send a `DELETE` request, then you can use the `_method` parameter or the `X-HTTP-Method-Override` header in your request. How this works is you will send a `POST` request to `https://ourawesomesite.com/wp-json/my-shop/v1/products/1?_method=DELETE`. Now you will have deleted product number 1, even though your client could not send the proper HTTP method in the request, or maybe there was a firewall in place that blocks out DELETE requests.

The HTTP method, in combination with the route and callbacks, are what make up the core of an endpoint.

Callbacks

There are currently only two types of callbacks for endpoints supported by the REST API; `callback` and `permission_callback`. The main callback should handle the interaction with the resource. The permissions callback should handle what users have access to the endpoint. You can add additional callbacks by adding additional information when registering an endpoint. You can then hook into `rest_pre_dispatch`, `rest_dispatch_request`, or `rest_post_dispatch` hooks to fire your new custom callbacks.

Endpoint Callback

The main callback for a delete endpoint should only delete the resource and return a copy of it in the response. The main callback for a creation endpoint should only create the resource and return a response matching the newly created data. An update callback should only modify resources that actually exist. A reading callback should only retrieve data that already exists. It is important to take into account the concept of idempotence.

Idempotence, in the context of a REST API, means that if you make the same request to an endpoint the server will process the request the same way. Imagine if our read endpoint was not idempotent. Whenever we made a request to it the state of our server would be modified by the request, even though we were only trying to get data. This could be catastrophic. Any time someone fetched data from your server something would change internally. It is important to make sure that read, update, and delete endpoints do not have nasty side effects and just stick to what they are intended to do.

In a REST API, the concept of idempotence is tied to HTTP methods instead of endpoint callbacks. Any callback using `GET`, `HEAD`, `TRACE`, `OPTIONS`, `PUT`, or `DELETE`, should not produce any side effects. `POST` requests are not idempotent, and are typically used for creating resources. If you created an idempotent creation method then you would only ever create one resource because when you make the same request there would be no more side effects to the server. For creating, if you make the same request over and over the server should generate new resources each time.

To restrict usage of endpoints we need to register a permissions callback.

Permissions Callback

Permissions callbacks are extremely important for security with the WordPress REST API. If you have any private data that should not be displayed publicly, then you need to have permissions callbacks registered for your endpoints. Below is an example of how to register permissions callbacks.

```
/**
 * This is our callback function that embeds our resource in a WP_REST_Response
 */
function prefix_get_private_data() {
    // rest_ensure_response() wraps the data we want to return into a WP_REST_Response, and
ensures it will be properly returned.
    return rest_ensure_response( 'This is private data.' );
}

/**
 * This is our callback function that embeds our resource in a WP_REST_Response
 */
function prefix_get_private_data_permissions_check() {
    // Restrict endpoint to only users who have the edit_posts capability.
    if ( ! current_user_can( 'edit_posts' ) ) {
        return new WP_Error( 'rest_forbidden', esc_html__( 'OMG you can not view private
data.', 'my-text-domain' ), array( 'status' => 401 ) );
    }

    // This is a black-listing approach. You could alternatively do this via white-listing, by
returning false here and changing the permissions check.
    return true;
}

/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_example_routes() {
    // register_rest_route() handles more arguments but we are going to stick to the basics
for now.
    register_rest_route( 'my-plugin/v1', '/private-data', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable
endpoints will work as intended.
```

```
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched
by the WP_REST_Server class.
        'callback' => 'prefix_get_private_data',
        // Here we register our permissions callback. The callback is fired before the main
callback to check if the current user can access the endpoint.
        'permission_callback' => 'prefix_get_private_data_permissions_check',
    ) );
}

add_action( 'rest_api_init', 'prefix_register_example_routes' );
```

If you try out this endpoint without any Authentication enabled then you will also be returned the error response, preventing you from seeing the data. Authentication is a huge topic and eventually a portion of this chapter will be created to show you how to create your own authentication processes.

Arguments

When making requests to an endpoint you might need to specify extra parameters to change the response. These extra parameters can be added while registering endpoints. Let's look at an example of how to use arguments with an endpoint.

```
/**
 * This is our callback function that embeds our resource in a WP_REST_Response
 */
function prefix_get_colors( $request ) {
    // In practice this function would fetch the desired data. Here we are just making stuff
up.
    $colors = array(
        'blue',
        'blue',
        'red',
        'red',
        'green',
        'green',
    );

    if ( isset( $request['filter'] ) ) {
       $filtered_colors = array();
```

```
        foreach ( $colors as $color ) {
            if ( $request['filter'] === $color ) {
                $filtered_colors[] = $color;
            }
        }
        return rest_ensure_response( $filtered_colors );
    }
    return rest_ensure_response( $colors );
}

/**
 * We can use this function to contain our arguments for the example product endpoint.
 */
function prefix_get_color_arguments() {
    $args = array();
    // Here we are registering the schema for the filter argument.
    $args['filter'] = array(
        // description should be a human readable description of the argument.
        'description' => esc_html__( 'The filter parameter is used to filter the collection of colors', 'my-text-domain' ),
        // type specifies the type of data that the argument should be.
        'type'        => 'string',
        // enum specified what values filter can take on.
        'enum'        => array( 'red', 'green', 'blue' ),
    );
    return $args;
}

/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_example_routes() {
    // register_rest_route() handles more arguments but we are going to stick to the basics for now.
    register_rest_route( 'my-colors/v1', '/colors', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched by the WP_REST_Server class.
```

```
        'callback' => 'prefix_get_colors',
        // Here we register our permissions callback. The callback is fired before the main
callback to check if the current user can access the endpoint.
        'args' => prefix_get_color_arguments(),
    ) );
}

add_action( 'rest_api_init', 'prefix_register_example_routes' );
```

We have now specified a `filter` argument for this example. We can specify the argument as a query parameter when we request the endpoint. If we make a `GET` request to `https://ourawesomesitem.com/my-colors/v1/colors?filter=blue`, we will be returned only the blue colors in our collection. You could also pass these as body parameters in the request body, instead of in the query string. To understand the distinction between query parameters and body parameters you should read about the HTTP spec. Query parameters live in the query string tacked onto the URL and body parameters are directly embedded in the body of an HTTP request.

We have created an argument for our endpoint, but how do we verify that the argument is a string and tell whether it matches the value red, green, or blue. To do this we need to specify a validation callback for our argument.

Validation

Validation and sanitization are extremely important for security in the API. The validate callback (in WP 4.6+), fires before the sanitize callback. You should use the `validate_callback` for your arguments to verify whether the input you are receiving is valid. The `sanitize_callback` should be used to transform the argument input or clean out unwanted parts out of the argument, before the argument is processed by the main callback.

In the example above, we need to verify that the `filter` parameter is a string, and it matches the value red, green, or blue. Let's look at what the code looks like after adding in a `validate_callback`.

```
/**
 * This is our callback function that embeds our resource in a WP_REST_Response
 */
function prefix_get_colors( $request ) {
    // In practice this function would fetch more practical data. Here we are just making
stuff up.
    $colors = array(
        'blue',
        'blue',
        'red',
        'red',
```

```
        'green',
        'green',
    );

    if ( isset( $request['filter'] ) ) {
       $filtered_colors = array();
       foreach ( $colors as $color ) {
           if ( $request['filter'] === $color ) {
               $filtered_colors[] = $color;
           }
       }
       return rest_ensure_response( $filtered_colors );
    }
    return rest_ensure_response( $colors );
}
/**
 * Validate a request argument based on details registered to the route.
 *
 * @param  mixed            $value   Value of the 'filter' argument.
 * @param  WP_REST_Request  $request The current request object.
 * @param  string           $param   Key of the parameter. In this case it is 'filter'.
 * @return WP_Error|boolean
 */
function prefix_filter_arg_validate_callback( $value, $request, $param ) {
    // If the 'filter' argument is not a string return an error.
    if ( ! is_string( $value ) ) {
        return new WP_Error( 'rest_invalid_param', esc_html__( 'The filter argument must be a
string.', 'my-text-domain' ), array( 'status' => 400 ) );
    }

    // Get the registered attributes for this endpoint request.
    $attributes = $request->get_attributes();

    // Grab the filter param schema.
    $args = $attributes['args'][ $param ];

    // If the filter param is not a value in our enum then we should return an error as well.
    if ( ! in_array( $value, $args['enum'], true ) ) {
        return new WP_Error( 'rest_invalid_param', sprintf( __( '%s is not one of %s' ),
$param, implode( ', ', $args['enum'] ) ), array( 'status' => 400 ) );
```

```
    }
}

/**
 * We can use this function to contain our arguments for the example product endpoint.
 */
function prefix_get_color_arguments() {
    $args = array();
    // Here we are registering the schema for the filter argument.
    $args['filter'] = array(
        // description should be a human readable description of the argument.
        'description' => esc_html__( 'The filter parameter is used to filter the collection of colors', 'my-text-domain' ),
        // type specifies the type of data that the argument should be.
        'type'        => 'string',
        // enum specified what values filter can take on.
        'enum'        => array( 'red', 'green', 'blue' ),
        // Here we register the validation callback for the filter argument.
        'validate_callback' => 'prefix_filter_arg_validate_callback',
    );
    return $args;
}

/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_example_routes() {
    // register_rest_route() handles more arguments but we are going to stick to the basics for now.
    register_rest_route( 'my-colors/v1', '/colors', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched by the WP_REST_Server class.
        'callback' => 'prefix_get_colors',
        // Here we register our permissions callback. The callback is fired before the main callback to check if the current user can access the endpoint.
        'args' => prefix_get_color_arguments(),
    ) );
```

```
}


add_action( 'rest_api_init', 'prefix_register_example_routes' );
```

Sanitizing

In the above example, we do not need to use a `sanitize_callback`, because we are restricting input to only values in our enum. If we did not have strict validation and accepted any string as a parameter, we would definitely need to register a `sanitize_callback`. What if we wanted to update a content field and the user entered something like `alert('ZOMG Hacking you');`. The field value could potentially be an executable script. To strip out unwanted data or to transform data into a desired format we need to register a `sanitize_callback` for our arguments. Here is an example of how to use WordPress's `sanitize_text_field()` for a sanitize callback:

```
/**
 * This is our callback function that embeds our resource in a WP_REST_Response.
 *
 * The parameter is already sanitized by this point so we can use it without any worries.
 */
function prefix_get_item( $request ) {
    if ( isset( $request['data'] ) ) {
        return rest_ensure_response( $request['data'] );
    }

    return new WP_Error( 'rest_invalid', esc_html__( 'The data parameter is required.', 'my-
text-domain' ), array( 'status' => 400 ) );
}

/**
 * Validate a request argument based on details registered to the route.
 *
 * @param  mixed            $value   Value of the 'filter' argument.
 * @param  WP_REST_Request  $request The current request object.
 * @param  string           $param   Key of the parameter. In this case it is 'filter'.
 * @return WP_Error|boolean
 */
function prefix_data_arg_validate_callback( $value, $request, $param ) {
    // If the 'data' argument is not a string return an error.
    if ( ! is_string( $value ) ) {
        return new WP_Error( 'rest_invalid_param', esc_html__( 'The filter argument must be a
```

```
string.', 'my-text-domain' ), array( 'status' => 400 ) );
    }
}

/**
 * Sanitize a request argument based on details registered to the route.
 *
 * @param  mixed            $value   Value of the 'filter' argument.
 * @param  WP_REST_Request  $request The current request object.
 * @param  string           $param   Key of the parameter. In this case it is 'filter'.
 * @return WP_Error|boolean
 */
function prefix_data_arg_sanitize_callback( $value, $request, $param ) {
    // It is as simple as returning the sanitized value.
    return sanitize_text_field( $value );
}

/**
 * We can use this function to contain our arguments for the example product endpoint.
 */
function prefix_get_data_arguments() {
    $args = array();
    // Here we are registering the schema for the filter argument.
    $args['data'] = array(
        // description should be a human readable description of the argument.
        'description' => esc_html__( 'The data parameter is used to be sanitized and returned
in the response.', 'my-text-domain' ),
        // type specifies the type of data that the argument should be.
        'type'        => 'string',
        // Set the argument to be required for the endpoint.
        'required'    => true,
        // We are registering a basic validation callback for the data argument.
        'validate_callback' => 'prefix_data_arg_validate_callback',
        // Here we register the validation callback for the filter argument.
        'sanitize_callback' => 'prefix_data_arg_sanitize_callback',
    );
    return $args;
}
```

```
/**
 * This function is where we register our routes for our example endpoint.
 */
function prefix_register_example_routes() {
    // register_rest_route() handles more arguments but we are going to stick to the basics for now.
    register_rest_route( 'my-plugin/v1', '/sanitized-data', array(
        // By using this constant we ensure that when the WP_REST_Server changes our readable endpoints will work as intended.
        'methods'  => WP_REST_Server::READABLE,
        // Here we register our callback. The callback is fired when this endpoint is matched by the WP_REST_Server class.
        'callback' => 'prefix_get_item',
        // Here we register our permissions callback. The callback is fired before the main callback to check if the current user can access the endpoint.
        'args' => prefix_get_data_arguments(),
    ) );
}

add_action( 'rest_api_init', 'prefix_register_example_routes' );
```

Summary

We have covered the basics of registering endpoints for the WordPress REST API. Routes are the URIs that our endpoints live at. Endpoints are a collection of callbacks, methods, args, and other options. Each endpoint is mapped to a route when using `register_rest_route()`. An endpoint by default can support various HTTP methods, a main callback, a permissions callback, and registered arguments. We can register endpoints to cover any of our use cases for interacting with WordPress. The endpoints serve as the core interaction point with the REST API, but there are many other topics to explore and understand, to fully utilize this powerful API.

Schema

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.

[info]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.[/info]

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"`.

[info]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.[/info]

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