Using The Rest Api

- Authentication
- Backbone JavaScript Client
- Client Libraries
- Discovery
- Global Parameters
- Linking and Embedding
- Pagination

Authentication

Cookie Authentication

Cookie authentication is the standard authentication method included with WordPress. When you log in to your dashboard, this sets up the cookies correctly for you, so plugin and theme developers need only to have a logged-in user.

However, the REST API includes a technique called nonces to avoid CSRF issues. This prevents other sites from forcing you to perform actions without explicitly intending to do so. This requires slightly special handling for the API.

For developers using the built-in Javascript API, this is handled automatically for you. This is the recommended way to use the API for plugins and themes. Custom data models can extend `wp.api.models.Base` to ensure this is sent correctly for any custom requests.

For developers making manual Ajax requests, the nonce will need to be passed with each request. The API uses nonces with the action set to `wp_rest`. These can then be passed to the API via the `_wpnonce` data parameter (either POST data or in the query for GET requests), or via the `X-WP-Nonce` header. If no nonce is provided the API will set the current user to 0, turning the request into an **unauthenticated request**, even if you're logged into WordPress.

Note: Until recently, most software had spotty support for `DELETE` requests. For instance, PHP doesn't transform the request body of a `DELETE` request into a super global. As such, supplying the nonce as a header is the most reliable approach.

It is important to keep in mind that this authentication method relies on WordPress cookies. As a result this method is only applicable when the REST API is used inside of WordPress and the current user is logged in. In addition, the current user must have the appropriate capability to perform the action being performed.

As an example, this is how the built-in Javascript client creates the nonce:

```
<?php
wp_localize_script( 'wp-api', 'wpApiSettings', array(
	'root' => esc_url_raw( rest_url() ),
	'nonce' => wp_create_nonce( 'wp_rest' )
) );
```

This is then used in the base model:

```
options.beforeSend = function(xhr) {
	xhr.setRequestHeader('X-WP-Nonce', wpApiSettings.nonce);

	if (beforeSend) {
		return beforeSend.apply(this, arguments);
	}
};
```

Here is an example of editing the title of a post, using jQuery AJAX:

```
$.ajax( {
	url: wpApiSettings.root + 'wp/v2/posts/1',
	method: 'POST',
	beforeSend: function ( xhr ) {
		xhr.setRequestHeader( 'X-WP-Nonce', wpApiSettings.nonce );
	},
	data:{
		'title' : 'Hello Moon'
	}
} ).done( function ( response ) {
	console.log( response );
} );
```

Note that you do not need to verify that the nonce is valid inside your custom end point. This is automatically done for you in `rest_cookie_check_errors()`.

Basic Authentication with Application Passwords

As of 5.6, WordPress has shipped with Application Passwords, which can be generated from an Edit User page (wp-admin -> Users -> Edit User).

The credentials can be passed along to REST API requests served over https:// using Basic Auth / RFC 7617 — here's the documentation for how to use it with cURL.

For a simple command-line script example, just swap out USERNAME, PASSWORD, and HOSTNAME in this with their respective values:

```
curl --user "USERNAME:PASSWORD" https://HOSTNAME/wp-json/wp/v2/users?context=edit
```

Authentication Plugins

Plugins may be added to support alternative modes of authentication that will work from remote applications. Some example plugins are OAuth 1.0a Server and JSON Web Tokens.

[info]

There's also a Basic Authentication plugin.

Note that this plugin requires sending your username and password with every request, and should only be used for development and testing i.e. not in a production environment. Using Application Passwords (see above) is preferred.

[/info]

Backbone JavaScript Client

The REST API includes a JavaScript/Backbone client library.

The library provides an interface for the WP REST API by providing Backbone Models and Collections for all endpoints exposed through the API Schema.

Using

Activate the WP-API plugin. Enqueue the script directly:

```
wp_enqueue_script( 'wp-api' );
```

or as a dependency for your script:

```
wp_enqueue_script( 'my_script', 'path/to/my/script', array( 'wp-api' ) );
```

The library parses the root endpoint (the 'Schema') and creates matching Backbone models and collections. You will now have two root objects available to you: `wp.api.models` and `wp.api.collections`.

The models and collections include:

Models:

- Category
- Comment
- Media
- Page
- PageMeta
- PageRevision
- Post
- PostMeta
- PostRevision
- Schema
- Status
- Tag
- Taxonomy
- Type
- User

Collections:

- Categories
- Comments
- Media
- PageMeta
- PageRevisions
- Pages
- Posts
- Statuses
- Tags
- Taxonomies
- Types
- Users

You can use these endpoints as-is to read, update, create and delete items using standard Backbone methods (fetch, sync, save & destroy for models, sync for collections). You can also extend these objects to make them your own, and build your views on top of them.

Default values

Each model and collection includes a reference to its default values, for example:

`wp.api.models.Post.prototype.args`

- author: null
- comment_status: null
- content: null
- date: null
- date_gmt: null
- excerpt: null
- featured_media: null
- format: null
- modified: null
- modified_gmt: null
- password: null
- ping_status: null
- slug: null
- status: null
- sticky: null
- title: null

Available methods

Each model and collection contains a list of methods the corresponding endpoint supports. For example, models created from `wp.api.models.Post` have a methods array of:

```
["GET", "POST", "PUT", "PATCH", "DELETE"]
```

Accepted options

Each model and collection contains a list of options the corresponding endpoint accepts (note that options are passed as the second parameter when creating models or collections), for example:

`wp.api.collections.Posts.prototype.options`

- author
- context
- filter
- order
- orderby
- page
- per_page
- search
- status

Localizing the API Schema

The client will accept and use a localized schema as part of the `wpApiSettings` object. The Schema is currently not passed by default; instead the client makes an ajax request to the API to load the Schema, then caches it in the browser's session storage (if available). Activating the client-js plugin with `SCRIPT_DEBUG` enabled uses a localized Schema. Check the client-js example or this branch which attempts to only localize the schema once per client.

Waiting for the client to load

Client startup is asynchronous. If the api schema is localized, the client can start immediately; if not the client makes an ajax request to load the schema. The client exposes a load promise for provide a reliable wait to wait for client to be ready:

```
wp.api.loadPromise.done( function() {
//... use the client here
} )
```

Model examples:

To create a post and edit its categories, make sure you are logged in, then:

```
// Create a new post
var post = new wp.api.models.Post( { title: 'This is a test post' } );
post.save();

// Load an existing post
var post = new wp.api.models.Post( { id: 1 } );
post.fetch();

// Get a collection of the post's categories (returns a promise)
// Uses _embedded data if available, in which case promise resolves immediately.
post.getCategories().done( function( postCategories ) {
  // ... do something with the categories.
  // The new post has an single Category: Uncategorized
  console.log( postCategories[0].name );
  // response -> "Uncategorized"
} );

// Get a posts author User model.
post.getAuthorUser().done( function( user ){
  // ... do something with user
  console.log( user.get( "name" ) );
} );

// Get a posts featured image Media model.
post.getFeaturedMedia().done( function( image ){
  // ... do something with image
  console.log( image );
} );

// Set the post categories.
post.setCategories( [ "apples", "oranges" ] );

// Get all the categories
var allCategories = new wp.api.collections.Categories()
allCategories.fetch();
```

```
var appleCategory = allCategories.findWhere( { slug: "apples" } );

// Add the category to the postCategories collection we previously fetched.
appleCategory.set( "parent_post", post.get( "id" ) );

// Use the POST method so Backbone will not PUT it even though it has an id.
postCategories.create( appleCategory.toJSON(), { type: "POST" } );

// Remove the Uncategorized category
postCategories.at( 0 ).destroy();

// Check the results - re-fetch
postCategories = post.getCategories();

postCategories.at( 0 ).get( "name" );
// response -> "apples"
```

Collection examples:

To get the last 10 posts:

```
var postsCollection = new wp.api.collections.Posts();
postsCollection.fetch();
```

To get the last 25 posts:

```
postsCollection.fetch( { data: { per_page: 25 } } );
```

Use filter to change the order & orderby options:

```
postsCollection.fetch( { data: { 'filter': { 'orderby': 'title', 'order': 'ASC' } } } );
```

All collections support pagination automatically, and you can get the next page of results using `more`:

```
postsCollection.more();
```

To get page 5 of a collection:

```
posts.fetch( { data: { page: 5 } } );
```

Check if the collection has any more posts:

```
posts.hasMore();
```

Working With Revisions

You can access post or page revisions using the PostRevisions or PageRevisions collections or through the Post or Page collection.

For example, to get a collection of all revisions of post ID 1:

```
var revisions = new wp.api.collections.PostRevisions({}, { parent: 1 });
```

Revision collections can also be accessed via their parent's collection. This example makes 2 HTTP requests instead of one, but now the original post and its revisions are available:

```
var post = new wp.api.models.Post( { id: 1 } );
post.fetch();
post.getRevisions().done( function( revisions ){
  console.log( revisions );
});
```

If you add custom endpoints to the API they will also become available as models/collections. For example, you will get new models and collections when you add REST API support to your custom post type. To access custom endpoints created, use the item's JSON endpoint name as CamelCase; for example if the item is found at the JSON path of /wp-json/wp/v2/my-custom-post, it can be accessed via the api at wp.api.collections.MyCustomPost. Note: Because the schema is stored in the user's session cache to avoid re-fetching, you may need to open a new tab to get a new read of the Schema.

```
// Extend wp.api.models.Post and wp.api.collections.Posts to load a custom post type
const CustomPost = wp.api.models.Post.extend( {
  urlRoot: wpApiSettings.root + 'wp/v2/custom_post_slug',
  defaults: {
    type: 'custom_post_slug',
  },
} );
const CustomPosts = wp.api.collections.Posts.extend( {
```

```
  url: wpApiSettings.root + 'wp/v2/custom_post_slug',
  model: BLProduct,
} );
const someCustomPosts = new CustomPosts();
someCustomPosts.fetch().then( ( posts ) => {
  // do something with the custom posts
} );
```

Client Libraries

The API can be used from any application by sending basic HTTP requests; however, client libraries simplify the process of querying or creating specific resources. These libraries make it easy to connect an external application to the WordPress REST API using a variety of programming languages.

[alert]Aside from the Backbone.js client, these libraries are not part of WordPress core, and are not necessarily maintained or endorsed by the WordPress REST API team.[/alert]

[info]To perform authenticated requests from outside of the WordPress admin, themes, or plugins, a separate authentication plugin is required.[/info]

PHP

Requests for PHP is a general purpose HTTP request library. While not specifically designed for the WordPress REST API, it is a great tool for interacting with any HTTP API.

WordPress-PHP-SDK is a PHP SDK for the WordPress REST API, which can be installed using composer.

WordPress-SDK is a Laravel package for the WordPress REST API, which can be installed using composer.

JavaScript

The Backbone.js client is built in to WordPress core and provides Backbone.js models & collections for working with REST API resources.

node-wpapi is an isomorphic JavaScript client library for querying or writing to the REST API using an intuitive chaining syntax. It can be used with Node.js or with client-side JavaScript applications.

ember-wordpress provides a connection between Ember Data and the REST API

Ruby

wp-api-client: a read-only REST API client written in Ruby.

C# / .NET

WordPressPCL: a full REST API client written in C#.

Dart / Flutter

wordpress_client: A library exposing a fluent api to interact with the WordPress REST API.

Discovery

When your client talks to an unknown site, you'll need to discover what the site is capable of and how the site is configured. There are a couple of steps for this, depending on what you need to discover.

Discovering the API

The first step of connecting to a site is finding out whether the site has the API enabled. Typically, you'll be working with URLs from user input, so the site you're accessing could be anything. The discovery step lets you verify the API is available, as well as indicating how to access it.

Link Header

The preferred way to handle discovery is to send a HEAD request to the supplied address. The REST API automatically adds a Link header to all front-end pages that looks like the following:

```
Link: <http://example.com/wp-json/>; rel="https://api.w.org/"
```

This URL points to the root route (`/`) of the API, which is then used for further discovery steps.

For sites without "pretty permalinks" enabled, `/wp-json/` isn't automatically handled by WordPress. This means that normal/default WordPress permalinks will be used instead. These headers look more like this:

```
Link: <http://example.com/?rest_route=/>; rel="https://api.w.org/"
```

Clients should keep this variation in mind and ensure that both routes can be handled seamlessly.

This auto-discovery can be applied to any URL served by a WordPress installation, so no pre-processing on user input needs to be added. Since this is a HEAD request, the request should be safe to send blindly to servers without worrying about causing side-effects.

Element

For clients with a HTML parser, or running in the browser, the equivalent of the Link header is included in the `<head>` of front-end pages through a `<link>` element:

```
<link rel='https://api.w.org/' href='http://example.com/wp-json/' />
```

In-browser Javascript can access this via the DOM:

```
// jQuery method
var $link = jQuery( 'link[rel="https://api.w.org/"]' );
var api_root = $link.attr( 'href' );

// Native method
var links = document.getElementsByTagName( 'link' );
var link = Array.prototype.filter.call( links, function ( item ) {
return ( item.rel === 'https://api.w.org/' );
} );
var api_root = link[0].href;
```

For in-browser clients, or clients without access to HTTP headers, this may be a more usable way of discovering the API. This is similar to Atom/RSS feed discovery, so existing code for that purpose may also be automatically adapted instead.

RSD (Really Simple Discovery)

For clients with support for XML-RPC discovery, the RSD method may be more applicable. This is an XML-based discovery format typically used for XML-RPC. RSD has two steps. The first step is to find the RSD endpoint, supplied as a `<link>` element:

```
<link rel="EditURI" type="application/rsd+xml" title="RSD"
href="http://example.com/xmlrpc.php?rsd" />
```

The second step is to fetch the RSD document and parse the available endpoints. This involves using an XML parser on a document like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<rsd version="1.0" xmlns="http://archipelago.phrasewise.com/rsd">
  <service>
    <engineName>WordPress</engineName>
    <engineLink>http://wordpress.org/</engineLink>
    <homePageLink>http://example.com/</homePageLink>
    <apis>
      <api name="WordPress" blogID="1" preferred="true"
apiLink="http://example.com/xmlrpc.php" />
```

```
      <!-- ... -->
      <api name="WP-API" blogID="1" preferred="false" apiLink="http://example.com/wp-json/" />
    </apis>
  </service>
</rsd>
```

The REST API always has a `name` attribute with the value equal to `WP-API`.

RSD is the least-preferred method of autodiscovery for a couple of reasons. The first step of RSD-based discovery involves parsing the HTML to first find the RSD document itself, which is equivalent to the `<link>` Element autodiscovery. It then involves another step to parse the RSD document, which requires a full XML parser.

Where possible, we suggest avoiding RSD-based discovery due to the complexity, but existing XML-RPC clients may prefer to use this method if they already have an RSD parser enabled. For XML-RPC clients which wish to use the REST API as a progressive enhancement to the codebase, this avoids needing to support different forms of discovery.

Authentication Discovery

Discovery is also available for authentication methods available via the API. The API root's response is an object describing the API, which includes an `authentication` key:

```
{
    "name": "Example WordPress Site",
    "description": "YOLO",
    "routes": { ... },
    "authentication": {
        "oauth1": {
            "request": "http://example.com/oauth/request",
            "authorize": "http://example.com/oauth/authorize",
            "access": "http://example.com/oauth/access",
            "version": "0.1"
        }
    }
}
```

The `authentication` value is a map (associative array) of authentication method ID to authentication options. The options available here are specific to the authentication method itself.

See the authentication documentation for the options for specific authentication methods.

Extension Discovery

Once you've discovered the API, the next step is check what the API supports. The index of the API exposes the `namespaces` item, which contains the extensions to the API that are supported.

For WordPress sites using versions 4.4 through 4.6, only the base API infrastructure is available, not the full API with endpoints. This also includes the oEmbed endpoints:

```
{
"name": "Example WordPress Site",
"namespaces": [
"oembed/1.0/"
]
}
```

Sites with the full API available (i.e. with WordPress 4.7+ or the REST API plugin installed) will have the `wp/v2` item in `namespaces` as well:

```
{
"name": "Example WordPress Site",
"namespaces": [
"wp/v2",
"oembed/1.0/"
]
}
```

Before attempting to use any of the core endpoints, you should be sure to check that the API is supported by checking for `wp/v2` support. WordPress 4.4 enabled the API infrastructure for all sites, but did **not** include the core endpoints under `wp/v2`. Core endpoints were added in WordPress 4.7.

This same mechanism can be used for detecting support for any plugins that support the REST API. For example, take a plugin which registers the following route:

```
<?php
register_rest_route( 'testplugin/v1', '/testroute', array( /* ... */ ) );
```

This would add the `testplugin/v1` namespace to the index:

```
{
"name": "Example WordPress Site",
```

```
□"namespaces": [
□□"wp/v2",
□□"oembed/1.0/",
□□"testplugin/v1"
□]
}
```


Resource Discovery

As of WordPress 5.5 the REST API also provides a discovery mechanism for identifying the REST API route equivalent of the current document. A link is added with a `rel` of `alternate` and a `type` of `application/json` that points to a REST API url. The link is added both as a `Link` header and a `<link>` element.

For instance, in the `<head>` section of this page, the following `<link>` appears.

```
<link rel="alternate" type="application/json" href="https://developer.wordpress.org/wp-json/wp/v2/rest-api-handbook/23085">
```

Links are added for post, pages, and other custom post types, as well as terms and author pages. Links are not currently output for post archives or search results.

Global Parameters

The API includes a number of global parameters (also called "meta-parameters") which control how the API handles the request/response handling. These operate at a layer above the actual resources themselves, and are available on all resources.

`_fields`

A REST resource like a Post contains a large quantity of data: basic information such as content, title, and author ID, but also registered metadata and fields, media information, and links to other resources. Your application may not need all of this information on every request.

To instruct WordPress to return only a subset of the fields in a response, you may use the `_fields` query parameter. If for example you are building an archive view and only need the ID, title, permalink, author and excerpt for a collection of posts, you can restrict the response to only those properties with this fields query:

```
/wp/v2/posts?_fields=author,id,excerpt,title,link
```

You may alternatively provide that same list using query parameter array syntax instead of a comma-separated list:

```
/wp/v2/posts?_fields[]=author&_fields[]=id&_fields[]=excerpt&_fields[]=title&_fields[]=link
```

When `_fields` is provided then WordPress will skip unneeded fields when generating your response object, avoiding potentially expensive internal computation or queries for data you don't need. This also means the JSON object returned from the REST API will be smaller, requiring less time to download and less time to parse on your client device.

Carefully design your queries to pull in only the needed properties from each resource to make your application faster to use and more efficient to run.

As of WordPress 5.3 the `_fields` parameter supports nested properties. This can be useful if you have registered many meta keys, permitting you to request the value for only one of the registered meta properties:

```
?_fields=meta.one-of-many-keys
```

Only the meta value with the key `one-of-many-keys` will be returned, and others will be excluded.

You can also request specific deeply-nested properties within a complex meta object:

```
?_fields=meta.key_name.nested_prop.deeply_nested_prop,meta.key_name.other_nested_prop
```

`_embed`

Most resources include links to related resources. For example, a post can link to the parent post, or to comments on the post. To reduce the number of HTTP requests required, clients may wish to fetch a resource as well as the linked resources. The `_embed` parameter indicates to the server that the response should include these embedded resources.

Embed mode is enabled if the `_embed` parameter is passed in the query string (GET parameter). This parameter does not require a value (i.e. `?_embed` is valid), however can be passed "1" as a value if required by a client library.

As of WordPress 5.4, the resources to embed can be limited by passing a list of link relation names to the `_embed` parameter. For example, `/wp/v2/posts?_embed=author,wp:term` will only embed the post's author and the lists of terms associated with the post.

Resources in embed mode will contain an additional `_embedded` key next to the `_links` key containing the linked resources. Only links with the `embeddable` parameter set to `true` will be embedded.

In order to use `_embed` together with `_fields`, add `_embedded` as well as `_links` to the fields, for instance, `/wp-json/wp/v2/posts/_embed=author,wp:term&_fields=title,author,_links,_embedded`.

For more about linking and embedding, see the Linking and Embedding page.

`_method` (or `X-HTTP-Method-Override` header)

Some servers and clients cannot correctly process some HTTP methods that the API makes use of. For example, all deletion requests on resources use the `DELETE` method, but some clients do not provide the ability to send this method.

To ensure compatibility with these servers and clients, the API supports a method override. This can be passed either via a `_method` parameter or the `X-HTTP-Method-Override` header, with the value set to the HTTP method to use.

[alert] Clients should only ever send a method override parameter or header with POST requests. Using the method override with GET requests may cause the request to be incorrectly cached. [/alert]

A `POST` to `/wp-json/wp/v2/posts/42?_method=DELETE` would be translated to a `DELETE` to the `wp/v2/posts/42` route.

Similarly, the following POST request would become a DELETE:

```
POST /wp-json/wp/v2/posts/42 HTTP/1.1
Host: example.com
X-HTTP-Method-Override: DELETE
```

`_envelope`

Similarly to `_method`, some servers, clients, and proxies do not support accessing the full response data. The API supports passing an `_envelope` parameter, which sends all response data in the body, including headers and status code.

Envelope mode is enabled if the `_envelope` parameter is passed in the query string (GET parameter). This parameter does not require a value (i.e. `?_envelope` is valid), but can be passed "1" as a value if required by a client library.

[info] For future compatibility, other values should not be passed. [/info]

Enveloped responses include a "fake" HTTP 200 response code with no additional headers (apart from Content-Type) that should ensure the response correctly passes through intermediaries.

For example, given the following response to a `GET` to `wp/v2/users/me`:

```
HTTP/1.1 302 Found
Location: http://example.com/wp-json/wp/v2/users/42

{
  "id": 42,
  ...
}
```

The equivalent enveloped response (with a `GET` to `wp/v2/users/me?_envelope`) would be:

```
HTTP/1.1 200 OK

{
  "status": 302,
  "headers": {
    "Location": "http://example.com/wp-json/wp/v2/users/42"
  },
  "body": {
    "id": 42
  }
}
```

`_jsonp`

The API natively supports JSONP responses to allow cross-domain requests for legacy browsers and clients. This parameter takes a JavaScript callback function which will be prepended to the data. This URL can then be loaded via a `<script>` tag.

The callback function can contain any alphanumeric, `_` (underscore), or `.` (period) character. Callbacks which contain invalid characters will receive a HTTP 400 error response, and the callback will not be called.

[info] Modern browsers can use Cross-Origin Resource Sharing (CORS) preflight requests for cross-domain requests, but JSONP can be used to ensure support with all browsers.

- Browser Support
- MDN Article on CORS

[/info]
For example:

```
<script>
function receiveData( data ) {
  // Do something with the data here.
  // For demonstration purposes, we'll simply log it.
  console.log( data );
}
</script>
<script src="https://example.com/wp-json/?_jsonp=receiveData"></script>
```

Linking and Embedding

The WP REST API incorporates hyperlinking throughout the API to allow discoverability and browsability, as well as embedding related resources together in one response. While the REST API does not completely conform to the entire HAL standard, it implements the `._links` and `._embedded` properties from that standard as described below.

Links

The `_links` property of the response object contains a map of links to other API resources, grouped by "relation." The relation specifies how the linked resource relates to the primary resource.

Examples include:

- `author` - describing a relationship between a resource and its author
- `wp:term` - describing the relationship between a post and its tags or categories

The relation is either a

- standardized relation
- a `URI relation` - like `https://api.w.org/term`
- or a `Compact URI relation` - like `wp:term`

Compact URI relations can be normalized to full URI relations to ensure full compatibility if required. This is similar to HTML `<link>` tags, or `<a rel="">` links.

The links are an object containing a `href` property with an absolute URL to the resource, as well as other optional properties. These include content types, disambiguation information, and data on actions that can be taken with the link.

For collection responses (those that return a list of objects rather than a top-level object), each item contains links, and the top-level response includes links via the `Link` header instead.

[info] If your client library does not allow accessing headers, you can use the `_envelope` parameter to include the headers as body data instead. [/info]

Example Response

A typical single post request (`/wp/v2/posts/42`):

```
{
  "id": 42,
  "_links": {
    "collection": [
      {
        "href": "https://example.com/wp-json/wp/v2/posts"
      }
    ],
    "author": [
      {
        "href": "https://example.com/wp-json/wp/v2/users/1",
        "embeddable": true
      }
    ]
  }
}
```


Embedding

Optionally, some linked resources may be included in the response to reduce the number of HTTP requests required. These resources are "embedded" into the main response.

Embedding is triggered by setting the `_embed` query parameter on the request. This will then include embedded resources under the `_embedded` key adjacent to the `_links` key. The layout of this object mirrors the `_links` object, but includes the embedded resource in place of the link properties.

Only links with the `embeddable` flag set to `true` can be embedded, and `_embed` will cause all embeddable links to be embedded. Only relations containing embedded responses are included in `_embedded`, however relations with mixed embeddable and unembeddable links will contain dummy responses for the unembeddable links to ensure numeric indexes match those in `_links`.

When embedding a collection response, for instance `/wp/v2/posts?author=1`, the embedded collection will have the default pagination limits applied.

Example Response


```
{
  "id": 42,
```

```
  "_links": {
    "collection": [
      {
        "href": "https://example.com/wp-json/wp/v2/posts"
      }
    ],
    "author": [
      {
        "href": "https://example.com/wp-json/wp/v2/users/1",
        "embeddable": true
      }
    ]
  },
  "_embedded": {
    "author": {
      "id": 1,
      "name": "admin",
      "description": "Site administrator"
    }
  }
}
```

Pagination

WordPress sites can have a lot of content—far more than you'd want to pull down in a single request. The API endpoints default to providing a limited number of items per request, the same way that a WordPress site will default to 10 posts per page in archive views.

Pagination Parameters

Any API response which contains multiple resources supports several common query parameters to handle paging through the response data:

- `?page=`: specify the page of results to return.
  - For example, `/wp/v2/posts?page=2` is the second page of posts results
  - By retrieving `/wp/v2/posts`, then `/wp/v2/posts?page=2`, and so on, you may access every available post through the API, one page at a time.
- `?per_page=`: specify the number of records to return in one request, specified as an integer from 1 to 100.
  - For example, `/wp/v2/posts?per_page=1` will return only the first post in the collection
- `?offset=`: specify an arbitrary offset at which to start retrieving posts
  - For example, `/wp/v2/posts?offset=6` will use the default number of posts per page, but start at the 6th post in the collection
  - `?per_page=5&page=4` is equivalent to `?per_page=5&offset=15`

[tip] Large queries can hurt site performance, so `per_page` is **capped at 100 records**. If you wish to retrieve more than 100 records, for example to build a client-side list of all available categories, you may make multiple API requests and combine the results within your application. [/tip]

To determine how many pages of data are available, the API returns two header fields with every paginated response:

- `X-WP-Total`: the total number of records in the collection
- `X-WP-TotalPages`: the total number of pages encompassing all available records

By inspecting these header fields you can determine how much more data is available within the API.

Ordering Results

In addition to the pagination query parameters detailed above, several other parameters control the order of the returned results:

- `?order=`: control whether results are returned in ascending or descending order
  - Valid values are `?order=asc` (for ascending order) and `?order=desc` (for descending order).
  - All native collections are returned in descending order by default.
- `?orderby=`: control the field by which the collection is sorted
  - The valid values for `orderby` will vary depending on the queried resource; for the `/wp/v2/posts` collection, the valid values are "date," "relevance," "id," "include," "title," and "slug"
  - See the REST API reference for the values supported by other collections
  - All collections with dated resources default to `orderby=date`