WP engine

THE ULTIMATE GUIDE TO THE WORDPRESS REST API

By Josh Pollock

Table of Contents

Introduction:

This ebook provides an in-depth overview of website and application development using the WordPress REST API. It includes several practical examples, designed to get you working with the WordPress REST API to improve your WordPress site and integrate WordPress with non-WordPress frameworks and tools.

The WordPress REST API is revolutionary for WordPress for several reasons. One reason in particular is because of its ability to display and save content from other sites and applications, regardless of whether or not they use WordPress. This means that WordPress as a content management system can be used with any framework or any programming language.

Much of the discussion around the REST API has focused on integrating WordPress with other technologies, but that's not all it allows us to do. Many of the examples in this guide focus on things you can build with WordPress using the WordPress REST API.

Themes can use the REST API to load content dynamically. The WordPress REST API can also provide a more efficient and standardized way to process forms or handle front-end AJAX, as opposed to repurposing the existing admin-ajax endpoint in the front-end.

Historically, this type of integration has been limited in scope and only available through XML-RPC. The WordPress REST API, however, uses a RESTful API — a universal connector for data on the internet. What used to be incredibly complicated with XML-RPC is simplified by using the more common REST standard.

While XML-RPC APIs use structured markup (XML) for making requests and responses, RESTful APIs use JavaScript Standard Notation, or JSON.

Most programming languages can easily convert their standard data structures into JSON, and vice versa. For example, in PHP we use json_encode() and json_decode() to translate PHP arrays or objects into JSON.

This means that, on a practical level, WordPress can be the content management tool for an application written in any language and it can easily send and receive data using the JSON standard.

Now the famous “Five Minute Install” will not only deliver content management, a relational database, and the plugin theme system we expect from WordPress, but it will also include the kind of API expected in modern-day web development tools.

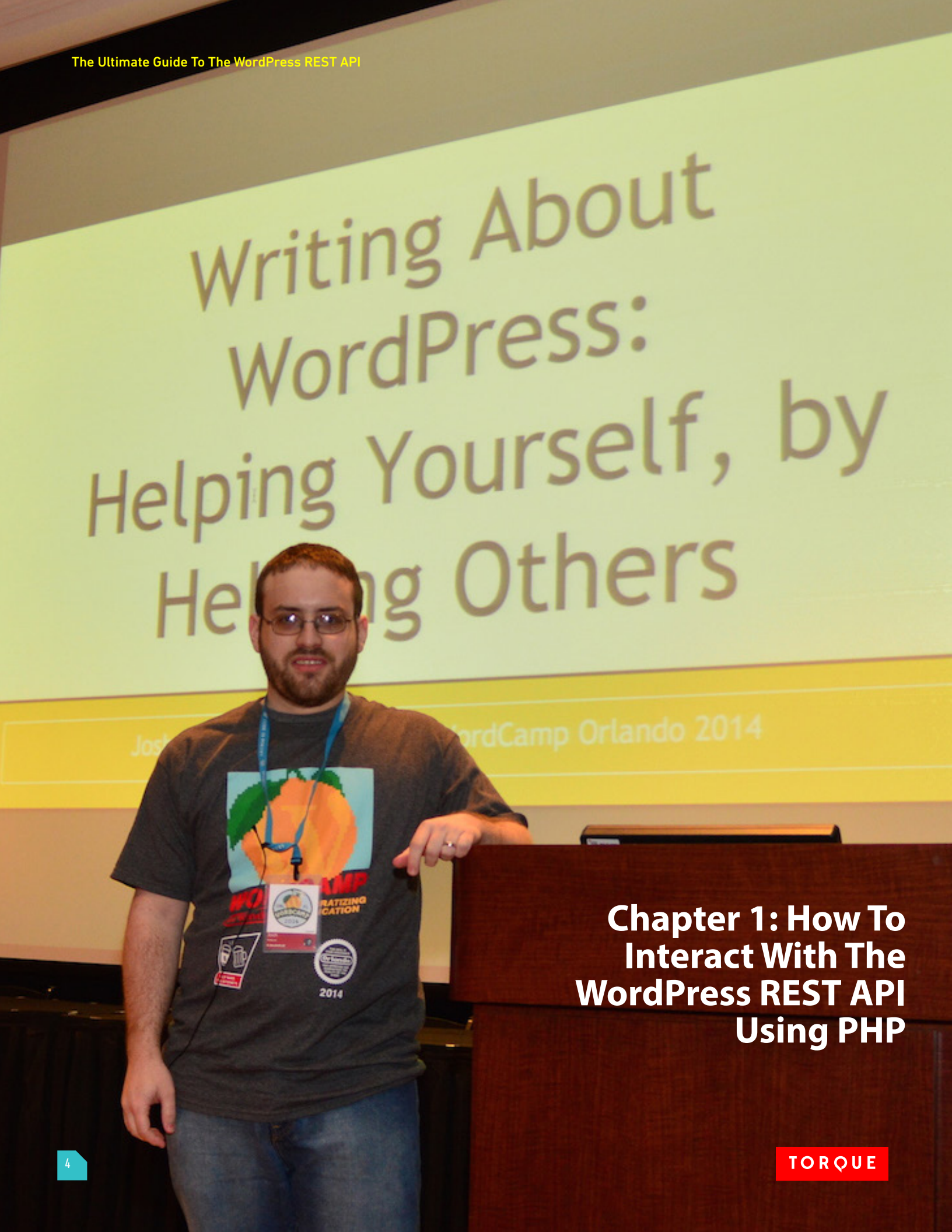
The WordPress JSON REST API provides a new way to interact with WordPress posts, users, comments, and taxonomies using the popular RESTful API standard, which allows you to create, read, update, and delete the main WordPress objects and provides infrastructure for creating your own custom APIs.

In this book, you'll learn the basics of working with and extending the default routes using the WordPress REST API. You'll also learn how to create your own custom routes and endpoints.

The WordPress REST API can be used to power single-page, client-side JavaScript applications — either as part of a WordPress theme or plugin, or in a separate, front-end client that's totally separated from the backend. Similarly, it also opens up new opportunities for making plugins and themes more extensible and dynamic, as well as more integrable with separate Software-as-a-Service (SaaS) businesses.

A RESTful API opens up endless opportunities for experienced WordPress developers and provides more flexibility when developing our own plugins or themes, and integrating them with other services.

The WordPress REST API opens up a lot of exciting possibilities for developers. But first we have to learn how to use it, and find ways it can make our lives easier. I hope this book helps you get started on that path.



Chapter 1: How To Interact With The WordPress REST API Using PHP

Fundamentals

Before you begin, you will need to set up two sites. The REST API is included in WordPress core, so it can be any two (WordPress) sites, as long as they are not live. Using a live site to experiment is risky, and should be avoided. Also, since we will be using the basic authentication plugin, development cannot be done on a live server because it transmits login and password as plain text.

GETing And POSTing

RESTful APIs use the four most common HTTP transport methods: GET, PUT, POST, and DELETE. For this chapter, you will use GET requests to obtain data via the API and POST requests to update items.

If you have not used a RESTful API, it may seem intimidating; but, when it comes down to it, it's as simple as creating the right URL strings.

It's time to start experimenting in your browser.

Once you have the WordPress REST API installed, navigate your browser to http://local.wordpress.dev/wp-json/wp/v2/posts/1

You'll notice a JSON object for post ID No. 1. Alternatively, you may receive a message telling you that you don't have a post of that ID. This illustrates a simple GET request. If you remove the ID from the end of the URL, you will see one page of posts.

Congratulations, you just made your first GET request to the WordPress REST API!

You will notice a minimal amount of data spent on structure or spacing in your browser. Although this is one of the big advantages of JSON, it also makes it difficult for humans to read.

Postman is an excellent performance tool that will apply spacing to your JSON and make it more readable. Similarly, it also has a lot of extra capabilities that make experimenting with APIs much easier.

One extremely useful capability of Postman is its ability to turn a request into a code snippet, which you can then copy and paste into your code. I also use it to export JavaScript to help me create a proof of concept for an API integration.

The WordPress HTTP API

Another useful API, and one of the relatively more uncommon ones, is the HTTP API, which, as you may have guessed, simplifies HTTP requests.

The HTTP API simplifies the process of sending additional requests to the same site or another site. With a few lines of code, you can use the HTTP API to turn a URL string into a JSON object.

The HTTP API gives us several useful functions to make HTTP requests via PHP.

Here's the most simple example:

```
$json = wp_remote_get( 'http://local.wordpress.dev/wp-json/wp/v2/posts/1' );
```

If you output $json now, you will see that the return isn't actually the JSON object that you want. It's an array that, among many other indexes, is an index called body, which has the JSON object. For the most part, that's actually all you need. Although you can get the body directly using wp_remote_retrieve_body(), this is a bad idea because, if something was wrong with your request or the server making the request, it could be a WP_Error instead of the actual response you need.

Now it's time to put together a simple PHP API client that will take a URL string and output a PHP array of post objects. Our mini-client is just one function:

```
function slug_get_json( $url ) {
	//GET the remote site
	$response = wp_remote_get( $url );

	//Check for error
	if ( is_wp_error( $response ) ) {
		return sprintf( 'The URL %1s could not be retrieved.', $url );
	}

	//get just the body
	$data = wp_remote_retrieve_body( $response );

	//return if not an error
	if ( ! is_wp_error( $data )  ) {

		//decode and return
		return json_decode( $data );

	}

}
```

This function starts from a URL string and makes a GET request to load it into $response. Next, you need to make sure the request works by testing if $response is not an object of the WP_Error class. If it is, it will return an error message. If not, you pass $response to wp_remote_retrieve_body() to get the JSON object into $data.

Again, make sure that you did not get an error. If no error appears, you can use json_decode() to return an array of post objects.

Go ahead and feed it a URL string like http://local.wordpress.dev/wp-json/wp/v2/posts/1 and run the results through print_r.

This gets really sophisticated if you pass it a URL string from another site. In the example set up in local.wordpress.dev, pass in the string http://local.wordpress-trunk.dev/wp-json/wp/v2/posts and you'll see the posts from the second site.

Making Queries

Getting a specific post or all posts via the API is only so useful. But what if you want specific posts based on various conditions, like you're use to with WP_Query?

Fortunately, there's a subset of WP_Query parameters available with the REST API's post endpoints that you can use to query other sites.

By adding filter parameters to the URL string, you can query for different post statuses other than publish. Alternatively, you can change how many posts you return or how they are ordered. You can even run a search by post title. For full documentation on these parameters, see the REST API documentation.

For example, if you want to retrieve eight posts in reverse alphabetical order, by post title, you can use this URL string:

http://local.wordpress.dev/wp-json/wp/v2/posts?per_page=8&order=DESC

Typing out this URL is tedious; fortunately WordPress can do it automatically by using the function add_query_arg(). If you pass it an array of arguments as the first parameter, and use our base URL as the second parameter, it will give you the right URL string. Here is how I created it:

```
$args = array(
	'filter[orderby]' => 'title',
	'filter[per_page]' => 8,
	'filter[order]' => 'DESC'
);
$url = add_query_arg( $args, rest_url( 'posts' ) );
```

The REST API provides one route per post type. In the example, you worked only with the default post types. You could substitute any registered post type, including the post type for “posts” in the example above.

With the default routes provided, you can only query by one post type at a time. If this is an issue for you, then you should consider creating a custom route, which I discuss later in this book.

Even with add_query_arg(), creating these strings can be tedious. Rather than manually create these strings each time, you can create a function to build these URL strings. Many PHP and JavaScript frameworks with an HTTP API, including jQuery's AJAX API, provide this functionality for you.

Custom Post Types

So far you've only learned about working with posts in the default post type. The REST API also provides nearly identical functionality for pages.

In addition, when you create custom post types, you can add support for the REST API to them. In the supports argument, you can use 'show_in_rest' => false to prevent the creation of routes and endpoints for that custom post type, or set it to true to expose them via the REST API. "If you, for example, have a post type called “planet,” then you will have all of the same routes starting with “wp-json/wp/v2/planet” that you have for posts.

The WordPress REST API respects the permissions and capabilities of each post type. As a result, there isn't much to discuss regarding custom post types, other than the fact that everything you learn about working with the posts route can be applied to custom posts types.

Outputting The Posts With PHP

Getting JSON objects is extremely useful, but you need to be able to output the posts. I'll demonstrate how to do so with PHP. In this book I switch between PHP and JavaScript, since I'm a WordPress developer. Just keep in mind that these fundamental concepts can be applied to any language. Also note that when I talk about posts, it is very similar to how you work with other objects.

Each object has various properties for all fields that make up a WordPress post.

This means that working with these objects is much like working with the post objects you work with when you access the global $post object or when you get an object with get_post().

Here's a simple loop that iterates through all posts retrieved by the REST API and outputs them with some basic markup:

```
$url =  add_query_arg( 'per_page', 10, rest_url() );
$posts = slug_get_json( $posts );
if ( ! empty( $posts ) ) {
       foreach( $posts as $post ) { ?>
              <article id="<?php echo esc_attr($post->ID ); ?>">
                     <h1><?php echo $post->title; ?></h1>
                     <div><?php wpautop( $post->content ); ?></div>
              </article>
       <?php }//foreach
}
```

This is a more efficient way of doing what you can already do with a WordPress theme. I'm showing it to you for two very important reasons: The first is because this will familiarize you with the data that the REST API returns, and in turn show just how easy it is to use. The second, and more important reason, is that with almost exactly the same code and a different URL, you can show posts from a completely different site.

Chapter 2: Copying Posts From One Site To Another

In this chapter you will build on what you learned in Chapter 1, and use the REST API to copy posts from one site to another. You will learn how to create a post on a site with data sourced from a remote site via the REST API. You will also learn how to take a post on a site and create a copy of it on a remote site. By the end of this chapter you should be able to display posts from a remote site and create posts on a remote site.

This is foundational work designed to make you comfortable passing JSON objects via HTTP requests. You can apply this knowledge to pass any type of data between WordPress and non-WordPress powered apps and sites. This opens up a world of possibilities. And the best part is, it's actually quite simple.

In Chapter 1 you learned how to create two functions: one for creating URL strings for generating GET requests via the REST API, and the other for making those requests. Make sure you are able to execute both of these functions in your test environment before you continue.

Copying Posts From A Remote Site

First, you need to know how to take a post from one site and create a copy of it on another.

You can start by getting the JSON post data from the REST API and converting it to a PHP object. You'll use the function slug_get_json as defined in the last chapter.

All you need to do is this:

```
$url = 'http://local.wordpress-trunk.dev/wp-json/wp/v2/posts/1';
$post = slug_get_json( $url );
```

Now, with a bit of work, you can turn $post into an array, which you can then pass to wp_insert_post() to create a copy of the post. The REST API returns an array that has slightly different key names than wp_insert_post() is expecting. For example, it has a key called title, however you need a key called post_title in the array passed to wp_insert_post().

You can use this function to do the conversion:

```
function slug_insert_post_from_json( $post ) {

  //check we have an array or object (but not a WP_Error object)
  //either make sure it's an array or throw an error
  if ( is_array( $post ) || ( is_object( $post )  && ! is_wp_error( $post ) ) ) {
     //ensure $post is an array, converting from object if need be
     $post = (array) $post;
  }
  else {
     return sprintf( 'The data inputted to %1s must be an object or an array', __FUNCTION__ );
  }

  //we're not dealing with author right now
  //you could use this id to find author details with request to:
  // wp-json/wp/v2/{$post[ 'author' ]}
  unset( $post[ 'author' ] );

  //set up an array to do most of the conversion in one loop
  //Note: We set ID as import_id to ATTEMPT to use the same ID
  //Leaving as ID would UPDATE an existing post of the same ID
  $convert_keys = array(
     'title' => 'post_title',
     'content' => 'post_content',
     'slug' => 'post_name',
     'status' => 'post_status',
     'parent' => 'post_parent',
     'excerpt' => 'post_excerpt',
     'date' => 'post_date',
     'type' => 'post_type',
     'ID' => 'import_id',
```

```
  );

  //copy FROM API response array TO how wp_insert_post() wants it and unset old key
  foreach ( $convert_keys as $from => $to ) {
     if ( isset( $post[ $from ] ) ) {
        $post[ $to ] = $post[ $from ];
        unset( $post[ $from ] );
     }

  }

  //remove all keys of $post that are not allowed
  //also convert keys that are object to strings
  $allowed = array_values( $convert_keys );
  foreach( $post as $key => $value ) {
     if( ! in_array( $key, $allowed ) ) {
        unset( $post[ $key ] );
     }else{
        if ( is_object( $value ) ) {
           $post[ $key ] = $value->rendered;
        }
     }

  }

  //create post and return its ID
  return wp_insert_post( $post );

}
```

For simplicity, this function can be broken down into several steps.

Step 1: Make sure that the data passed into the function is either an array or an object. If it's an object, typecast it as an array. This is important because the rest of the function requires an array. If you pass the result of json_decode() directly to this function it will likely be an object.

Step 2: Once you're sure that you have a valid array, set up a second array, which you can use to simplify converting the array keys. Each value in this array is a key value pair.

For example, 'excerpt' => 'post_excerpt' will be used to convert the key excerpt to post_excerpt. You can use that array to copy the value from the old key to a new key and then unset the old key.

One important thing to note is that you converted the key ID to import_id. If you left the ID as "ID," WordPress will attempt to update a post of that particular ID. The conversion suggests to WordPress that it uses the ID from the remote site as the ID of the new post.

In addition, you can use this same array to remove all of the keys in the post array that cannot be passed to wp_insert_post().

Step 3: The JSON response object returns the user ID. If you knew that the ID on the remote site would match the user ID on your site, you could keep that. For now, you can avoid setting the author. The code example demonstrates how you could turn that ID into a new request to get the author's information, but for now let's consider this a step you can revisit later.

Step 4: Pass the final $post array to wp_insert_post(). This should create the post and return the ID of a new post.

Go ahead and give it a try.

Your First POST Request

Now it's time to use a POST request to create a post on the remote site. This is actually much easier than the last function, thanks to the WordPress HTTP API.

Putting together a proper POST request in PHP is normally a bit challenging, however when using wp_remote_post() it's actually quite simple. This function requires two parameters: the URL to make the POST request to and an array of arguments to use in the request.

In the code below you'll see all of the arguments for the POST request. Most are the defaults, shown for completeness, but the two you need to be most concerned with are the body and the header. The body is used to get the JSON object to pass to the remote site, and the header is used to pass an authentication array.

Before looking at the complete code, you should understand how to prepare for the post request by populating two variables, $body and $headers, which will be used as the respective body and header of the POST request. You will use the slug_get_json() function from Chapter 1 to get a JSON array for a post.

You could also get the post with get_post() and then, after some manipulation, convert it to a JSON array. You can skip that step and use the REST API to get the data exactly the way the it likes it.

Here's the setup code:

```
$url = 'http://local.wordpress.dev/wp-json/wp/v2/posts/1';
$body = slug_get_json( $url );
if ( is_object( $post ) ) {
	$body = $post;
	$headers    = array (
		'Authorization' => 'Basic ' . base64_encode( 'admin' . ':' . 'password' ),
	);
	$remote_url = 'http://local.wordpress-trunk.dev/wp-json/wp/v2/posts';

}
```

The first two lines should look familiar from Chapter 1 — they give you an object from the REST API. Once you have that in $post, you need to make sure that it is, in fact, an object. If it is, move it into $body.

Next, you need to set up your headers. This will be an array containing an index called “Authorization,” and in it you will set a string with the username and password. In the example code, you'll notice we used the default VVV admin password.

Before you continue, put the URL for the remote site's post endpoint into $remote_url. Now you're ready to make the POST request. Inside the conditional, check if $post is an object you can send to wp_remote_post().

Pass $headers to “headers” and $body to “body.” Of course, you'll need to make sure $body is a JSON array by passing it through json_encode.

Here's the complete code, which, as you can see, includes some error handling at the end:

```
$url = 'http://local.wordpress.dev/wp-json/wp/v2/posts/4426';
$post = slug_get_json( $url );
if ( is_object( $post ) ) {
	$remote_url = 'http://local.wordpress-trunk.dev/wp-json/wp/v2/posts';
	$headers    = array (
		'Authorization' => 'Basic ' . base64_encode( 'admin' . ':' . 'password' ),
	);

	$response = wp_remote_post( $remote_url, array (
		'method'      => 'POST',
		'timeout'     => 45,
		'redirection' => 5,
		'httpversion' => '1.0',
```

```
                    'blocking'    => true,
                    'headers'     => $headers,
                    'body'        => json_encode( $post ),
                    'cookies'     => array ()
                )
        );

        if ( is_wp_error( $response ) ) {
                $error_message = $response->get_error_message();
                echo sprintf(  '<p class="error">Something went wrong: %1s</p>', $error_message );
        }
        else {
                echo 'Response:<pre>';
                print_r( $response );
                echo '</pre>';
        }
}
else {
        $error_message = 'The input data was invalid.';
        echo sprintf(  '<p class="error">Something went wrong: %1s</p>', $error_message );
}
```

Run this in your console and you should get back an array from the remote site with the success code "200" in the header. Alternatively, you may get an error telling you what went wrong.

Chapter 3: Working With Post Meta Data

In this chapter you'll learn how to edit and create post meta fields using the WordPress REST API and how to retrieve or update metadata for a post. You'll also learn how to customize the default endpoints for a post to expose post meta or other data related to a post. While this chapter mainly discusses post meta, it can be largely applied to user meta as well.

Getting Meta Fields With The REST API

Assuming that you're familiar with getting and updating posts via the REST API, you probably know how to work with the main post data. You most likely know that if you query for a post your meta fields for the posts are not included.

There is an endpoint for post meta, but it requires authentication. Before we get into this, you need to learn how to expose meta fields in the main post route. This will allow a meta field to show up in a request for a post or a group of posts without any authentication or without using the meta endpoints that require authentication.

Imagine you have a post type called "jedi" and a meta field called "lightsaber_color" and you want all requests to "wp-json/wp/v2/jedi" to show the value of the "lightsaber_color" field. In this situation, you can use the function register_api_field() to add this field to the response.

This function has three arguments: The first is what object type — which in this case means post type — to add the field for. The second is the name of this field — in this case, the name of the meta field. And the third is an array of arguments. In the array you need to define the callback function for the names of the functions for getting and updating the value of the field.

The following example, and in a few examples after, will be receiving and updating post meta. To get started, you'll set up some generic callback functions:

```
function slug_get_post_meta_cb( $object, $field_name, $request ) {




    return get_post_meta( $object[ 'id' ], $field_name );
}

function slug_update_post_meta_cb( $value, $object, $field_name ) {
    return update_post_meta( $object[ 'id' ], $field_name, $value );
}
```

Keep in mind that these two functions will only work if the name of the field registered on the API is the same as the meta key. Also, be sure that the meta key is not protected. If the names don't match, a custom callback will be needed. I will address protected meta — keys beginning with an underscore — later in this chapter.

These two functions map arguments properly from the REST API to the familiar get_post_meta and update_update_meta functions. It's important to note that a “field” doesn't need to be treated as a meta field. You can declare a field of any name and provide a callback function to do anything you need with register_rest_field().

With these two functions you can register your API field:

```
add_action( 'rest_api_init', function() {
 register_api_field( 'jedi',
    'lightsaber_color',
    array(
       'get_callback'    => 'slug_get_post_meta_cb',
       'update_callback' => 'slug_update_post_meta_cb',
       'schema'          => null,
    )
 );

});
```

Now, whenever a request is made to wp-json/wp/v2/jedi/, a field lightsaber_color with the value of the corresponding meta field for the post(s) in the post type Jedi will be added to the data.

Creating And Editing Meta Fields With The REST API

If you make an authenticated request to the API, all posts will have additional endpoints available for meta.

For example, wp-json/wp/v2/posts/1/meta will show all non-protected meta fields for post ID 1, but only for authenticated requests.

If you want to make any meta_data or other related data publicly available, then register_api_field() is the only way to expose this data without a custom endpoint to non-authenticated users.

The meta endpoint for a post will show the meta IDs of each meta key. This is useful for updating an existing meta field.

For example, if you want to add a value for lightsaber color to a post in the jedi post type of ID 42, you need to make a POST request to wp-json/wp/v2/jedi/42/meta and, in the body of that request, include the meta key you were updating and the new value.

In order to update that key, you need to find the meta ID by making a request to wp-json/wp/v2/jedi/42/meta. Then, you can use that ID in a new POST request to that meta ID's endpoint — for example, if the meta ID was 100, it would be wp-json/wp/v2/jedi/42/meta/100.

Here's an example that uses the WordPress HTTP API to create the meta field:

```
//get URL for post 1's meta
$url = rest_url( 'wp/v2/posts/1/meta' );

//add basic auth headers (don't use in production)
$headers = array (
	'Authorization' => 'Basic ' . base64_encode( 'admin' . ':' . 'password' ),
);

//prepare body of request with meta key/ value
$body = array(
	'key' => 'lightsaber_color',
	'value' => 'blue'
);

//make POST request

$response = wp_remote_request( $url, array(
		'method' => 'POST',
		'headers' => $headers,
		'body' => $body
	)
);


//if response is not an error, echo color and get meta ID of new meta key
$body = wp_remote_retrieve_body( $response );
if ( ! is_wp_error( $body ) ) {
	$body = json_decode( $body );
	$meta_id = $body->id;
	echo "Color is " . $body->value;
	if ( $meta_id ) {

		//add meta ID to end of URL
		$url .= '/' . $meta_id;

		//this time just need to send value
		$body = array(

			'value' => 'green'

		);

		$response = wp_remote_request( $url, array(
				'method' => 'POST',
```

```
                        'headers' => $headers,
                        'body' => $body
                )
        );

        //if not an error echo new color
        $body = wp_remote_retrieve_body( $response );
        if ( ! is_wp_error( $body ) ) {
                $body = json_decode( $body );
                echo "Color  is  now " . $body->value;
        }

    }

}
```

You're probably not going to be editing a meta field immediately after creating it. To illustrate how you could make changes, however, I'll demonstrate how to edit the SEO title and SEO description fields added by Yoast's WordPress SEO plugin, while also making them available in the response for posts.

This will help bring the discussion full-circle.

The meta fields yoast_wpseo_title and yoast_wpseo_metadesc are considered protected by WordPress. To make an API request for them, you must first remove its protection. For this you can use the filter is_protected_meta.

Since this filter isn't part of the REST API, it affects any use of these fields. Make sure the protection is only removed during API calls — which can be determined based on the constant REST_REQUEST.

First, hook the filter is_protected_meta. If the REST_REQUEST constant is true, then, in the callback, set these two fields as unprotected.

Here's what it looks like:

```
add_filter( 'is_protected_meta', function( $protected, $meta_key ) {
   if ( '_yoast_wpseo_title' == $meta_key || '_yoast_wpseo_metadesc' == $meta_key && defined( 'REST_
REQUEST' ) && REST_REQUEST ) {
      $protected = false;

   }

   return $protected;

}, 10, 2 );
```

Now, you can register both fields in the API for posts:

```
register_api_field( 'post',
   '_yoast_wpseo_metadesc',
   array(
      'get_callback'    => 'slug_get_post_meta_cb',
      'update_callback' => 'slug_update_post_meta_cb',
      'schema'          => null,
   )

);
```

With all of that in place, you can now search for their IDs with a GET request to the meta endpoint for a post. This will allow you to create a URL to send a POST request to update the values. Of course, if there is no value, you will need to create one.

Here's a function that relies on the fields being registered and unprotected, similar to the example code above, which can update or create values in the two fields for a given post ID:

```
function slug_update_wp_seo_via_api( $post_id, $meta_title, $meta_desc, $auth_headers )  {

   //get URL for post's meta
   $meta_url = rest_url( 'wp/v2/posts/' . $post_id . '/meta' );

   //add basic auth headers (don't use in production)
   $headers[ 'Authorization' ] = $auth_headers;

   //make GET request for all meta
   $response = wp_remote_request( $meta_url, array(
         'method' => 'GET',
         'headers' => $headers,
      )
   );

   //if response is not an error continue
   $body = wp_remote_retrieve_body( $response );
   if ( ! is_wp_error( $body ) ) {

      //get meta data into an array
      $meta_data = json_decode( $body );

      //set variables for IDs of both fields to false
      $meta_title_id = $meta_desc_id = false;

      //loop until we found each
      foreach( $meta_data as $meta ) {

         //if current meta key is title set variable for it
         if ( '_yoast_wpseo_title' == $meta->key ) {
            $meta_title_id = $meta->id;
         }

         //if current meta key is description set variable for it
         if ( '_yoast_wpseo_metadesc' == $meta->key ){
            $meta_desc_id = $meta->id;
         }

         //break loop if we have both
         if ( false != $meta_desc_id && false != $meta_title_id ) {
            break;

         }

      }

      //see if we have a value for meta title ID
      //if so update, if not create
      if ( $meta_title_id ) {
         //add meta ID to end of URL
         $url = $meta_url . '/' . $meta_title_id;

         //put value in body
         $body = array(
            'value' => $meta_title
         );

         //update via POST request
         $response = wp_remote_request( $url, array(
               'method' => 'POST',
               'headers' => $headers,
               'body' => $body
            )
         );
```

```
        //fire an action to expose response when updating SEO title
        do_action( 'slug_wp_seo_title_updated_via_api', $response, $post_id );

    }else {

        //put value and key in body
        $body = array(
            'key' => '_yoast_wpseo_title',
            'value' => $meta_title
        );


        //Create via POST request
        $response = wp_remote_request( $meta_url, array(
                'method' => 'POST',
                'headers' => $headers,
                'body' => $body
            )
        );


        //fire an action to expose response when creating SEO title
        do_action( 'slug_wp_seo_title_created_via_api', $response, $post_id );

    }

    //see if we have a value for meta description ID
    //if so update, if not create
    if ( $meta_desc_id ) {

        //add meta ID to end of URL
        $url = $meta_url . '/' . $meta_desc_id;

        //put value in body
        $body = array(
            'value' => $meta_desc
        );


        //update via POST request
        $response = wp_remote_request( $url, array(
                'method' => 'POST',
                'headers' => $headers,
                'body' => $body
            )
        );



        //fire an action to expose response when updating SEO description
        do_action( 'slug_wp_seo_desc_updated_via_api', $response, $post_id );

    }else {

        //put value and key in body
        $body = array(
            'key' => '_yoast_wpseo_metadesc',
            'value' => $meta_desc
        );

        //Create via POST request
        $response = wp_remote_request( $meta_url, array(
                'method' => 'POST',
```

```
                'headers' => $headers,
                'body' => $body
            )
        );


        //fire an action to expose response when creating SEO description
        do_action( 'slug_wp_seo_desc_created_via_api', $response, $post_id );

    }

}

}
```

You Learned A Lot More Than How To Work With Post Meta

You should now successfully be able start working with post meta for post types using the WordPress REST API. Keep in mind that register_api_field can be used for a lot more than meta fields: It is a gateway to viewing and updating any type of data you want.

What you've learned in this chapter should equip you to use register_api_field() with any function you need to show or save data. This handy little function is one of the most useful parts of the API. Use its power wisely.

Chapter 4: Processing Forms With AJAX

In this chapter, you'll learn how to write a simple plugin to create and edit posts using the WordPress REST API. You'll also learn how to work with data in order to update your page dynamically based on the results.

Although this could be the basis of a front-end editing plugin, or a redo of the WordPress post editor, the objective isn't to reverse the engineering of the WordPress post editor. The point is to introduce you to processing form data using the REST API and AJAX.

After you understand how it's done, and just how simple it actually is, you can apply what you've learned to make your WordPress site more dynamic.

Fundamentals

Before you begin, be sure to review Chapters 1 through 3. Pay close attention to Chapter 2 on how to create posts by making POST requests to the posts endpoint. In this chapter, you'll do something similar, but instead of using the WordPress HTTP API and PHP, you'll use jQuery's AJAX methods. All of the code for this project should go in its own plugin file.

Be sure to install the JavaScript client for the WordPress REST API. You'll use the JavaScript client to make it possible to authorize via the current user's cookies. You can substitute another authorization method, such as oAuth, if you'd like.

Setting Up The Plugin

For this example, the plugin you're creating only requires one PHP file and one JavaScript file.

You can start by writing a PHP file to do three key things: enqueue the JavaScript file, localize a dynamically created JavaScript object into the DOM when you use that file, and create the HTML markup for your form.

To do this, you need two functions and two hooks. Create a new folder in your plugin directory, with one PHP file inside of it. You can call the file jp-rest-post-editor.php. Here's the php file with empty functions:

```
<?php
/*
Plugin Name: JP REST API Post Editor
*/

add_shortcode( 'JP-POST-EDITOR', 'jp_rest_post_editor_form');
function jp_rest_post_editor_form( ) {

}

add_action( 'wp_enqueue_scripts', 'jp_rest_api_scripts' );
function jp_rest_api_scripts() {

}
```

For this demonstration, notice that you're working only with the post title and post content. This means that in the editor form function you only need the HTML for a simple form for those two fields, like this:

```
function jp_rest_post_editor_form( ) {
 $form = '
       <form id="editor">
        <input type="text" name="title" id="title" value="Hello there">
        <textarea id="content" ></textarea>
         <input type="submit" value="Submit" id="submit">
     </form>
     <div id="results">

     </div>
   ';

return $form;

}
```

Of course, you only want to show this to users who are logged in and can edit posts. You can wrap the variable containing the form in some conditional checks. That way if the current user isn't logged in, you can still give them a link to the WordPress login.

Here's the complete function:

```
function jp_rest_post_editor_form( ) {

$form = '
       <form id="editor">
        <input type="text" name="title" id="title" value="Hello there">
        <textarea id="content" ></textarea>
         <input type="submit" value="Submit" id="submit">
     </form>
     <div id="results">

     </div>
   ';

if (  is_user_logged_in() ) {
     if ( user_can( get_current_user_id(), 'edit_posts' ) ) {
        return $form;
     }
     else {
        return __( 'You do not have permissions to edit posts.', 'jp-rest-post-editor' );
     }
  }
  else {
     return sprintf( '<a href="%1s" title="Login">%2s</a>', wp_login_url( get_permalink( get_
queried_object_id() ) ), __( 'You must be logged in to edit posts, please click here to log in.',
'jp-rest-post-editor') );
  }

}
```

One important thing to notice in this form is that it does not have a method or an action set. This is so it will not be processed automatically or cause a page reload upon submit.

The other thing you need to do is enqueue your JavaScript file. Once that's done, you'll want to localize an array of data into it, which you'll need to include in the JavaScript that needs to be generated dynamically. This includes the base URL for the REST API, since that can change with a filter and a nonce, for security purposes.

You should provide a success and failure message in that array so the strings are translation friendly. Later on, you'll need to know the current user's ID, so you should include that now, as well.

This is all accomplished via wp_enqueue_script() and wp_localize_script(). If you want to add custom styles to the editor, you can use the wp_enqueue_style() to do so.

Here's that function:

```
function jp_rest_api_scripts() {
       wp_enqueue_script( 'jp-api-post-editor', plugins_url( 'jp-api-post-editor.js', __FILE__ ),
array( 'jquery' ), false, true );
       wp_localize_script( 'jp-api-post-editor', 'JP_POST_EDITOR', array(
              'root' => esc_url_raw( rest_url() ),
              'nonce' => wp_create_nonce( 'wp_json' ),
              'successMessage' => __( 'Post Created Successfully.', 'jp-rest-post-editor' ),
              'failureMessage' => __( 'An error occurred.', 'jp-rest-post-editor' ),
              'userID'    => get_current_user_id(),
       ) );

}
```

That's all the PHP you need. Everything else is handled via JavaScript.

Create a new page with the editor shortcode (JP-POST-EDITOR), and go to that page. You should see the post editor form. It's not going to work yet, of course — you'll need to write a little JavaScript first.

Creating Posts

Creating posts from your form is going to require a POST request, which you can make using jQuery's AJAX method. For the most part, this is very simple, and should be familiar JavaScript. The two things that may be new are creating the JSON array and adding the authorization header. Let's walk through each of them.

To create the JSON object for your AJAX request, you must first create a JavaScript array from the form input and then use JSON.stringify() to convert it into JSON. This is the beginning of the JavaScript file that shows how to build the JSON array:

```
(function($){

   $( '#editor' ).on( 'submit', function(e) {
       e.preventDefault();

       var title =  $( '#title' ).val();
       var content =  $( '#content' ).val();
       var JSONObj = {
           "title"                :title,
           "content_raw"  :content,
           "status"            :'publish'
       };

       var data = JSON.stringify(JSONObj);
})(jQuery);
```

Before you can pass the variable data to the AJAX request, you must first set the URL for the request. This step is as simple as appending wp.v2/posts to the root URL for the API, which is accessible via JP_POST_EDITOR.root:

```
var url = JP_POST_EDITOR.root;
url += 'wp/v2/posts';
```

The AJAX request itself looks much like any other AJAX request you would make, with the exception of the authorization headers. Thanks to the REST API's JavaScript client, you only need to add a header to the request containing the nonce set in the JP_POST_EDITOR object. Alternatively, you could substitute some other authorization method here, such as oAuth.

Using the nonce to verify cookie authentication involves setting a request header with the name X-WP-Nonce, which contains the nonce value. You can use the beforeSend function of the request to send the nonce.

Here's what that looks like in the AJAX request:

```
$.ajax({
           type:"POST",
           url: url,
           dataType : 'json',
           data: data,
           beforeSend : function( xhr ) {
               xhr.setRequestHeader( 'X-WP-Nonce', JP_POST_EDITOR.nonce );
           },
 });
```

The only thing missing is the success and failure functions. You can create alerts using the messages that you localized into the script earlier. For now, output the result of the request as a JSON array so you can see what it looks like.

Here's the complete code for the JavaScript to create a post editor that can create new posts:

```
(function($){

   $( '#editor' ).on( 'submit', function(e) {
       e.preventDefault();

       var title =  $( '#title' ).val();
       var content =  $( '#content' ).val();
       var JSONObj = {
           "title"                :title,
           "content_raw"  :content,
           "status"            :'publish'
       };

       var data = JSON.stringify(JSONObj);
       var url = JP_POST_EDITOR.root;
       url += 'wp/v2/posts';

       $.ajax({
           type:"POST",
           url: url,
           dataType : 'json',
           data: data,
           beforeSend : function( xhr ) {
               xhr.setRequestHeader( 'X-WP-Nonce', JP_POST_EDITOR.nonce );
           },

           success: function(response) {
               alert( JP_POST_EDITOR.successMessage );
               $( "#results").append( JSON.stringify( response ) );
           },
           failure: function( response ) {
               alert( JP_POST_EDITOR.failureMessage );
           }
       });

   });

})(jQuery);
```

That's all you need for a basic editor. Go ahead and give it a try.

If you're logged in and the API is active, you should create a new post, and then create an alert telling you that the post is created. Next, place the returned JSON object into the #results container.

Editing Existing Posts

Before you can edit posts, you need to have two things: a list of posts by author, with all of the post titles and post content, and a new form field to hold the ID of the post you're editing. You'll also need to know the ID of the post.

Before adding that hidden field to your form, add this bit of HTML:

```
<input type="hidden" name="post-id" id="post-id" value="">
```

Next, in the JavaScript function for creating posts, you can add a few lines of code to get the value of that field. This code will automatically change the URL so you can edit the post of that ID, rather than create a new one:

```
var postID = $( '#post-id').val();
if ( undefined !== postID ) {
   url += '/';
   url += postID;
}
```

Be sure to add this code before the AJAX section of the editor form processor.

If the field has a value, then the variable URL in the AJAX function will have the ID of the post you are editing. If not, it will create a new post, just like before.

To populate that field, as well as the post title and post content field, you'll need to add a second form. This one will make a GET request to retrieve all posts by the current user.

Now, based on the selection in that form, you can set the editor form to update.

In the PHP, you can add the second form to the HTML like this:

```
<form id="select-post">
  <select id="posts" name="posts">

  </select>
  <input type="submit" value="Choose Post To Edit" id="choose-post">
</form>
```

You can use the REST API to populate the options in the #posts select. To do that you'll need to make a request for posts by the current user. You can use the results to accomplish this goal.

At this time you may want to go back to Chapter 1 and review how to make GET requests to the REST API using PHP.

When you set up your script, set the current user ID as part of the JP_POST_EDITOR object. You can use that to form the URL for requesting posts by the current user.

The first thing you need to do is create a function to get posts by the current author and populate that select field. The AJAX request in this function is even more simple than the one you used to update posts, as it does not require authentication. The success function loops through the results and adds them to the post selector form as options for its one field:

```
function getPostsByUser( defaultID ) {
   url += '?filter[author]=';
   url += JP_POST_EDITOR.userID;
   url += '&filter[per_page]=20';

   $.ajax({
       type:"GET",
       url: url,
       dataType : 'json',
       success: function(response) {
           var posts = {};

           $.each(response, function(i, val) {
               $( "#posts" ).append(new Option( val.title, val.ID ) );
           });

           if ( undefined != defaultID ) {
               $('[name=posts]').val( defaultID )
           }
       }
   });
}
```

You will notice that this function has a parameter for defaultID, which, if defined, will be used to set the default value of the select field. For now, you can ignore it. Use this function without the default value and set it to run on document ready, like this:

```
$( document ).ready( function()  {
   getPostsByUser();
});
```

Just having a list of posts by the current user isn't enough. You need to get the content and title for the selected post and push it into the form for editing.

For that you'll need a second GET request to run on the submission of the post selector form, like this:

```
$( '#select-post' ).on( 'submit', function(e) {

   e.preventDefault();
   var ID = $( '#posts' ).val();
   var postURL = JP_POST_EDITOR.root;
   postURL += 'wp/v2/posts/';
   postURL += ID;

   $.ajax({
       type:"GET",
       url: postURL,
       dataType : 'json',
       success: function(post) {
           var title = post.title;
           var content = post.content;
           var postID = postID;

           $( '#editor #title').val( title );
           $( '#editor #content').val( content );
           $( '#select-post #posts').val( postID );
       }
   });

});
```

Here you build a new URL to get the post data for the selected post. This will be in the form of <json-url>wp/v2/posts/<post-id>.

After that, go ahead and make the actual request — take the returned data and set it as the value of the three fields in the editor form.

When you refresh the page, you should see all posts by the current user in that selector. When you click the submit button, these two things should occur: First, the title and content of the post you selected should be visible in the editor, and, second, the hidden field for the post ID you added should be set.

The function for the editor form is not set for editing existing posts. For that you'll need to make a slight modification to the function. Similarly, this will improve the success function for that request so it nicely displays the title and content of the post in the #results container, instead of the raw JSON data.

To do this you'll need one function to update the #results container. It should look like this:

```
function results( val ) {
       $( "#results").empty();
       $( "#results" ).append( '<div class="post-title">' + val.title + '</div>'  );
       $( "#results" ).append( '<div class="post-content">' + val.content + '</div>'  );
}
```

This is simple jQuery, but it is nonetheless a good introduction to updating page content using data from the REST API. You can get more creative with the markup or add additional fields.

Now that that's in place, you can use it in your modified form processing function:

```
$( '#editor' ).on( 'submit', function(e) {
   e.preventDefault();

   var title =  $( '#title' ).val();
   var content =  $( '#content' ).val();
   console.log( content );
```

```
    var JSONObj = {
        "title"         :title,
        "content_raw"   :content,
        "status"        :'publish'
    };

    var data = JSON.stringify(JSONObj);

    var postID = $( '#post-id').val();
    if ( undefined !== postID ) {
        url += '/';
        url += postID;
    }

    $.ajax({
        type:"POST",
        url: url,
        dataType : 'json',
        data: data,
        beforeSend : function( xhr ) {
            xhr.setRequestHeader( 'X-WP-Nonce', JP_POST_EDITOR.nonce );
        },

        success: function(response) {
            alert( JP_POST_EDITOR.successMessage );
            getPostsByUser( response.ID );
            results( response );
        },
        failure: function( response ) {
            alert( JP_POST_EDITOR.failureMessage );
        }
    });

});
```

The first thing that's changed is that you've conditionally added the post ID that's being edited. This means that the form can still serve to create new posts by POSTing to the endpoint or update posts via posts/<post-id>.

The other change is in the success function: The new results() function is used to output the post title and content. You also reran the getPostsByUser() function, but set the new or edited post as the default. Now the post editor will automatically be set to edit the post you just created.

You can see the complete JavaScript and files here. It's not full featured, but, with more than 100 lines of JavaScript, it's a pretty good start.

Chapter 5: Working With Taxonomies

This chapter discusses taxonomies and how to work with them using the WordPress REST API. You'll learn how to get a list of taxonomies, a list of terms in a taxonomy, and the list of posts associated with a term — all using PHP.

You'll also learn how to query for posts with a given taxonomy term. And, to put it all together in a practical example (and bring a little jQuery into the discussion), you'll learn how to use the REST API to show a list of posts in a specific tag when a user clicks on a tag link.

Fundamentals

You should already be familiar with the three built-in public taxonomies: category, tag, and post_format. WordPress also has a few internal private taxonomies, such as nav_menus, which is used to store menu settings. Since the REST API doesn't currently support private taxonomies, you can ignore them for now.

This chapter assumes that you already have a basic understanding of working with posts via the WordPress REST API. If you don't, please review Chapter 3 at this time.

All of the code in this chapter is designed to work with tags and categories, since those taxonomies are used on every site. This information can be easily abstracted to work with any custom taxonomy. If you're unfamiliar with custom taxonomies, but want to learn more, I recommend you read my article in Smashing Magazine on WordPress custom taxonomies.

Working With Taxonomies Directly

In WordPress, there are two common ways to work with taxonomies: The first is by listing terms in a taxonomy — for example, the category archives on our site lists all posts in a specific category. The second way is by listing taxonomy terms associated with a post — for example, listing all of the tags used on the current post.

The latter is often done using widgets — the tag cloud widget or category list widget, for example.

The WordPress REST API can provide a similar function.

The REST API gives us a taxonomies route, which includes many helpful endpoints for listing taxonomies and information about a taxonomy. For each taxonomy, a terms route lists all terms in a taxonomy. The posts route handles listing posts with a specific term, which I will discuss in the next section.

The basic endpoints for the taxonomies and terms are as follows:

- **wp-json/wp/v2/taxonomies/:** This is used to make a GET request to list all public taxonomies, which is useful for checking if a taxonomy is registered.
- **/wp-json/wp/v2/taxonomies/<taxonomy-slug>:** This is used to make a GET request to find information about a taxonomy, which is useful for finding information about a taxonomy, such as its labels.

For example, /wp-json/wp/v2/taxonomies/category provides basic information about the category taxonomy:

- **/wp-json/wp/v2/terms/<taxonomy-slug>:** Make a GET request to get all terms in the taxonomy. For example, GET /wp-json/wp/v2/terms/category for all categories, which is useful for finding the ID of a term in the taxonomy and for updating via the next endpoint. To create a term in the taxonomy, make a POST request to this endpoint.
- **/wp-json/wp/v2/terms/<taxonomy-slug>:** Make a GET request for information about a specific term. Make a POST request to update that term.

In the above examples, you would use "category" in place of <taxonomy-slug>, or post_tag for tags. For a custom taxonomy, use the name you used to register it.

You can make a list of terms in a taxonomy, in this case categories, like this:

```
$response = wp_remote_get( rest_url( 'wp/v2/terms/categories' ) );
if ( ! is_wp_error( $response ) ) {
	$terms = json_decode( wp_remote_retrieve_body( $response ) );
	$term_list = array();
	foreach( $terms as $term ) {
		$term_list[] = sprintf( '<li><a href="%1s">%2s</a></li>', esc_url( $term->link
),$term->name );
	}
	if ( ! empty( $term_list ) ) {
		echo sprintf( '<ul>%1s</ul>', implode( $term_list ) );
	}
}
```

If you've read Chapters 1 through 4, this should be fairly self explanatory: Use wp_remote_get() to make the request. Check if it's an error. And, if there are no errors, convert the body of the response to a PHP object and then loop through it.

Of course, you can do the same thing more efficiently by using get_terms() or a similar function. The preceding code is useful in an AJAX callback or when getting terms from a taxonomy on another site in a WordPress multisite installation.

To get terms from another site in a multisite installation, you would replace the function rest_url() with the get_rest_url(). These two functions work the same way, except get_rest_url() takes the site ID as its first argument.

Here's the modified code that will list the categories of every site in a multisite installation:

```
$sites = wp_get_sites();
$the_list = array();
foreach( $sites as $site ) {
	$response = wp_remote_get( get_rest_url( $site->site_id, 'wp/v2/terms/categories' ) );

	if ( ! is_wp_error( $response ) ) {
		$terms = json_decode( wp_remote_retrieve_body( $response ) );
		$term_list = array();
		foreach( $terms as $term ) {
			$term_list[] = sprintf( '<li><a href="%1s">%2s</a></li>', esc_url( $term->link
),$term->name );
		}

		if ( ! empty( $term_list ) ) {
			$site_info = get_blog_details( $site->site_id );
			$term_list = sprintf( '<ul>%1s</ul>', implode( $term_list ) );
			$the_list[] = sprintf( '<li><a href="%1s">%2s</a><ul>%3s</ul>', $site_info-
>siteurl, $site_info->blogname, $term_list );

		}

	}
}

if ( ! empty( $the_list ) ) {
	echo sprintf( '<ul>%1s</ul>', implode( $the_list ) );
}
```

This code is much more simple than working with switch_to_blog to run multiple WP_Query objects.

Keep in mind, however, that this will only work in multisite. Also, while it may appear quite simple, if you have a lot of different sites, it will involve several HTTP requests and numerous database queries. I strongly advise caching the results as much as possible.

Getting Posts By Taxonomy

The routes and endpoints described above only show you information about the taxonomy and its terms, not the posts associated with a given post. That information is retrieved via the post endpoints. You can either query for a post by a taxonomy or you can list terms in a taxonomy for a specific post.

For example, to get posts with the tag "stargate," use the tag parameter in your URL string, like this:

```
rest_url( 'wp/v2/posts?filter[tag]=stargate' );
```

Similarly, to get posts in the category "sci-fi," use the parameter category_name, like this:

```
rest_url( 'wp/v2/posts?filter[category_name]=stargate' );
```

You can combine queries for posts in multiple taxonomies to the URL string for posts with the tag "sith," and category "force_users," like this:

```
rest_url( 'wp/v2/posts?[category_name]=force_users&filter[tag]=sith' );
```

With these URL strings, you can use the same basic pattern as above to get the posts via the WordPress HTTP API.

Editing The Terms For A Post

All posts have separate endpoints for the terms of each taxonomy. For example, to see all categories of the post with ID 1, use wp-json/wp/v2/posts/1/terms/category/.

Updating and creating terms works similarly to how meta fields are created and updated, as demonstrated in Chapter 3, but it is not identical.

The key difference is, since taxonomy terms exist independently of posts, if a term doesn't exist, you need to create a term using the term route, then append it to the post. If it does exist, you can add it or remove it with a POST request — but you have to know the ID of the term.

This is a multi-step process. The first step is to GET the list of terms in a specific taxonomy. In this example, there's a post with an ID of 177, and it has two categories. You can find that info like this:

```
//make request to get terms
$url      = rest_url( 'wp/v2/posts/177/terms/category' );
$response = wp_remote_request( $url, array(
     'method' => 'GET'
  )
);

$body = wp_remote_retrieve_body( $response );
if ( ! is_wp_error( $body ) ) {

  //decode body from json
  $terms = json_decode( $body );

  //simplify array into term slug => term id
  $terms = array_combine( wp_list_pluck( $terms, 'slug' ), wp_list_pluck( $terms, id ) );

}
```

At the end of this, you'll notice an array of associated terms, keyed by term slug with the term ID as its value. This means you could remove the term with the slug star-wars by using the key of the array star-wars to find the term ID and make the DELETE request.

Continuing on the preceding example, here's how you would do that:

```
//first make sure we have the term
if ( isset( $terms[ 'star-wars' ] ) ) {

       //get term ID
       $term_id = $terms['star-wars'];

       //update url with term ID
       //$url was /wp-json/wp/v2/posts/177/terms/category
```

```
	$term_url = $url . '/' . $term_id;

	//make DELETE request with basic auth
	$headers =  array(
		'headers' => array(
			'Authorization' => 'Basic ' . base64_encode( 'admin : password' ),
		)
	);

	$response = wp_remote_request( $term_url,
		array(
			'method' => 'DELETE',
			'headers' => $headers
		)

	);

}
```

So far you've learned how to list terms for a post and remove terms from a post, but what if you wanted to add one? This will require finding the term ID. Similar to updating meta fields, this will require multiple requests, so I wrote a function to handle all of them.

This function checks if the term with the slug passed to it exists by querying the term route for the taxonomy. If it does, then it uses the ID of that term to add it to the specified post with a POST request. If not, it creates the term and then adds it to the post.

```
<?php

function slug_add_term_to_post( $post_id, $taxonomy, $term_slug, $term_name, $auth_header ) {

	//find post type of this post
	//return false if not found/ IE this post doesn't exist
	$post_type = get_post_type( $post_id );
	if ( false == $post_type ) {
		return;
	}

	//change 'post' to 'posts' if $post_type is 'post'
	if ( 'post' == $post_type ) {
		$post_type = 'posts';
	}

	$term_url = rest_url( 'wp/v2/terms/' . $taxonomy );
	$response = wp_remote_request( $term_url,
		array(
			'method' => 'GET',
		)
	);

	$body     = wp_remote_retrieve_body( $response );
	if ( ! is_wp_error( $body ) ) {
		$terms = json_decode( $body );
		if ( ! empty( $terms ) ) {
			//set $term_id false, then try and find it (IE see if it exists
			$term_id = false;
			foreach ( $terms as $term ) {
				if ( $term->slug == $term_slug ) {
					$term_id = $term->id;
					break;
				}

			}

			//prepare auth header, needed for the POST requests in next two steps
			$headers['Authorization'] = $auth_header;

			//if we never found a term_id we must create term
			if ( ! $term_id ) {
```

```
                        //put term slug and name in body of request
                        //you could also add term description or parent here
                        $body = array(
                               'slug' => sanitize_title( $term_slug ),
                               'name' => $term_name
                        );

                        //URL for POST request
                        $create_term_url = rest_url( 'wp/v2/terms/' . $taxonomy );

                        //create term
                        $response = wp_remote_request( $create_term_url,
                               array(
                                      'method'  => 'POST',
                                      'headers' => $headers,
                                      'body'    => $body,
                               )
                        );

                        //wp_die( print_r( $response ) );
                        //if possible, find term ID in response
                        $body = wp_remote_retrieve_body( $response );
                        if ( ! is_wp_error( $body ) ) {
                                    $term = json_decode( $body );
                                    if ( is_object( $term ) && isset( $term->id ) ) {
                                           $term_id = $term->id;
                                    }

                        }

                 }

                 //if we have a term id, which we should by now, add it to this post
                 if ( $term_id ) {

                        //create url
                        $post_term_url = rest_url( 'wp/v2/' . $post_type . '/' . $post_id . '/
terms/' . $taxonomy . '/' . $term_id );

                        //make POST request to add term to post
                        $response = wp_remote_request( $post_term_url,
                               array(
                                      'method'  => 'POST',
                                      'headers' => $headers
                               )
                        );

                        //fire action to expose response
                        do_action( 'slug_post_term_update', $response, $post_id, $post_type,
$taxonomy, $term_slug )
                 }

          }

          return $term_id;

    }

}
```

Using Taxonomies In JavaScript

Let's look at how you get all posts in a specific tag by using jQuery to make an AJAX request. The purpose of this code is to make a list of all posts with a specific tag when clicking on a tag link in a post.

The code is very similar to the code used in the earlier chapter on processing forms with the REST API. When enqueuing this code, be sure to localize the vale of rest_url() using wp_localize_script(). Next, place that into the variable rootURL so you can use it to build the URL string for the request.

Note: This code uses the standard markup for tags and posts, and is tested using the _s starter theme. You will need to make some adjustments if your theme uses another way to mark tags.

Here's the complete code:

```
jQuery(document).ready(function($) {
  $( '[rel="tag"]' ).click( function( event ) {
      event.preventDefault();
      var href = this.href;
      if (href.substr(-1) == '/') {
          href = href.substr( 0, href.length - 1 );
      }

      var slug = href.split('/').pop();

      $.ajax({
          type: 'GET',
          cache: true,
          url: rootURL + 'wp/v2/posts?filter[tag]=' + slug,
          dataType: 'json',
          success: function(posts) {
              var html = '<ul>';
              $.each( posts, function(index, post ) {
                  html += '<li><a href="' + post.link + '">' + post.title + '</a></li>';
              });
              html += '</ul>';
              $('article').append( html);
          }
      });
  });
});
```

Whenever someone clicks on a tag link, this code will retrieve the list of posts with that particular tag and list them below the post, rather than taking them directly to the archive.

You'll probably want to put it in a more user-friendly location on your page or in a tooltip or dialog. Regardless, the basic concept is the same. This code happens inside of a function that runs whenever a tag is clicked. The first part works to get the tag's slug by chopping down the link's target to its last segment.

Obviously this only works if you are using standard permalinks. Once that's done, the slug is used to create a URL string to make an AJAX request. If that request is successful, a simple jQuery each loop is used to build the list, which is then appended to the article.

Again, it's pretty straightforward, though you may need to tweak it to make it compatible with your theme.

Use Taxonomies Better

Most WordPress sites use taxonomy archives and widgets as their main way of interacting with taxonomies. This means that to get to a related post via that taxonomy you must click a link, load a new page, click another link, and then load another page. This is a bad user experience.

Using the WordPress REST API, we can make the interaction with taxonomies on our sites more interactive and get users directly to the content they need.

Chapter 6: Working With Users

With the capabilities of the REST API, WordPress's user management can be leveraged in web applications. It also allows theme developers to create more dynamic links between content that will highlight the author and their posts. This chapter provides an introduction to working with user data via the WordPress REST API. You'll learn how to create a profile editor and viewer — a process that is fairly straightforward since the REST API does most of the heavy lifting.

If you've worked with any of the other object types such as posts, terms, or comments, then working with user data should be familiar. If you haven't, don't worry, one of the many impressive features about version two of the REST API is its consistentency in how each object is treated. In other words, once you learn the pattern for one of the objects, you can apply it to all of the objects.

Retrieving User Data Via The API

Viewing Public User Data

In published posts, you can find a limited set of data for all users when making non-authenticated requests. This includes the user's avatar in a variety of sizes, their username, ID and description, and the url for their author archive and website.

This data can be found for all users by making a GET request to wp-json/wp/v2/users or for a specific user by adding the user's ID to the end of the url.

Keep in mind that when requesting all users, the results will be paged. You can control which pages of results will show up by using per_page and page arguments. For example, you can use wp-json/wp/v2/users?filter[page]=2&filter[per_page]=10 to get the results for pages 11 through 20.

This code is a basic example that uses jQuery AJAX to list all users with published posts. The second step uses the posts endpoint to query by the author.

Although it's a very basic example, when combined with some styling and a templating system, you could use it to build a directory system or a highly-dynamic author preview system. Be sure to explore the data available in the responses for users and posts, and try adding it to your markup.

If data you need is unavailable, keep in mind that adding fields to responses, using register_api_fields, applies to user endpoints as well. You can also use this code to create a modal with an author's recent post that was populated asynchronously when you clicked on what is normally a link to the author's post archive.

Viewing Privileged Data

So far we have only worked with publically available data. When you authenticate your request, you can add a query parameter "context" set to view more information about the user. This includes their capabilities and email address.

You can see the difference when authenticated as an admin by comparing the response of a GET request to wp-json/wp/v2/user/1 to a request to wp-json/wp/v2/users/1?context=view. The former looks exactly like a non-authenticated request while the latter shows a lot more information.

As an admin, all users are available once you're authenticated — not just those with public posts.

Editing Users Via The REST API

The REST API can also be used to update users. The same endpoints I showed earlier for viewing user data, can also be used to create and edit users.

For example, a POST request to wp/v2/users can be used to create a new user. When doing so, you must send at least a username, password, and email address.

Here's an example using jQuery AJAX to create a user:

```
$.ajax( {
   url: Slug_API_Settings.root + 'wp/v2/users/',
   method: 'POST',
   beforeSend: function ( xhr ) {
       xhr.setRequestHeader( 'X-WP-Nonce', Slug_API_Settings.nonce );
   },
   data:{
       email: 'someone@somewhere.net',
       username: 'someone',
       password: Math.random().toString(36).substring(7)
   }
} ).done( function ( response ) {
    console.log( response );
} )
```

Keep in mind that this example and the next one use cookie-based authentication with a nonce check, as documented here.
This means it will only work when the current user is logged in and has the right capabilities to allow them to create users.

If you are using a front-end decoupled from WordPress, you will need to use an alternative solution for authentication. It also requires that you localize data for the nonce, the root API url, and the current user ID. Here's my example:

```
add_action( 'wp_enqueue_scripts', function() {

  wp_enqueue_script( 'user-editor', plugin_dir_url( __FILE__ ) .'user-editor.js', array('jquery')
);
  wp_localize_script( 'user-editor', 'Slug_API_Settings', array(
     'root' => esc_url_raw( rest_url() ),
     'nonce' => wp_create_nonce( 'wp_rest' ),
     'current_user_id' => (int) get_current_user_id()

  ) );
});
```

For a more complex example, create a form to update the current user's email address. It will use the most basic HTML markup:

```
<form id="profile-form">
  <div id="username"></div>
  <input type="text" name="email" id="email">
  <input type="submit">
</form>
```

Assuming you have the same setup for your JavaScript as in the last example, here's how to get the current user via the REST API and put that information into the form:

```
jQuery( document ).ready(function( $ ) {

   $.ajax( {
       url: Slug_API_Settings.root + 'wp/v2/users/' + Slug_API_Settings.current_user_id +
'?context=edit',
       method: 'GET',
       beforeSend: function ( xhr ) {
           xhr.setRequestHeader( 'X-WP-Nonce', Slug_API_Settings.nonce );
       }
   } ).done( function ( user ) {
       $( '#username' ).html( '<p>' + user.name + '</p>' );
       $( '#email' ).val( user.email );
   } );

});
```

Upon page load, this makes a GET request for the current user and then adds their username and email to the form. Notice that I used ?conext=edit in the request. As I explained above, without it, only a limited response would be shown, which would not include the user's email address.

To make this form work for updating a current user, a similar request can be made when the form is submitted — except it will be a POST request.

Here's the JavaScript needed to do so:

```
jQuery( document ).ready(function( $ ) {

   var get_user_data;
   (get_user_data = function () {
       $.ajax( {
           url: Slug_API_Settings.root + 'wp/v2/users/' + Slug_API_Settings.current_user_id +
'?context=edit',
           method: 'GET',
           beforeSend: function ( xhr ) {
               xhr.setRequestHeader( 'X-WP-Nonce', Slug_API_Settings.nonce );
           }
       } ).done( function ( user ) {
           $( '#username' ).html( '<p>' + user.name + '</p>' );
           $( '#email' ).val( user.email );
       } );
   })();

   $( '#profile-form' ).on( 'submit', function(e) {
       e.preventDefault();
       $.ajax( {
           url: Slug_API_Settings.root + 'wp/v2/users/' + Slug_API_Settings.current_user_id,
           method: 'POST',
           beforeSend: function ( xhr ) {
               xhr.setRequestHeader( 'X-WP-Nonce', Slug_API_Settings.nonce );
           },
           data:{
               email: $( '#email' ).val()
           }
       } ).done( function ( response ) {
           console.log( response )
       } )
   });

});
```

I encourage you to use what you've learned here to explore the available fields and add more capabilities to this simple setup.

Go make something awesome!

In this chapter, you've learned the basics of viewing, updating, and creating users via the WordPress REST API. I've shown all of my examples in jQuery to encourage you to use this to make your themes, apps, and plugins more dynamic.

Now it's time for you to take what you've learned and apply it to what you're working on to create exciting new interfaces for user management.

Chapter 7: Preparing Your Website To Power A Single-Page Web App

The WordPress REST API affords developers the ability to transfer the process of rendering front-end content entirely into JavaScript, or some other more front-end-friendly language. It's not only easier to work with, but it also creates opportunities to increase interactivity and overall performance.

The next three chapters focus on creating single page web and mobile apps powered by the WordPress REST API, with a front-end separated entirely from WordPress. In fact, the front-end will run on a node.js server.

In this chapter you'll learn how to create the front-end for a site or app, without disrupting the use of WordPress for content management, by using the WordPress REST API.

Preparing Your WordPress Site

Installing the WordPress REST API is technically all that's required to get your WordPress site to act as the backend for a non-WordPress-powered front-end.

That said, there are several issues that you may need to address. Let's take a look at four of them:

Cross Origin Issues

The most common issue you can run into when working with a separate front-end is the restraints placed by cross-origin restrictions. For security reasons, most browsers by default won't let you load content from one site to another if they are served from two separate domains. This prevents AJAX requests from succeeding when requesting data from a separate domain. To mitigate the issue, make sure all of the correct Cross Resource Origin Sharing (CORS) headers are set.

CORS headers can be used to selectively allow certain domains — or alternatively any site — to access your site. While CORS headers can be set globally for a WordPress site, in this case, you want to set the header to apply only to the REST API's output. All headers, including CORS headers, must be output before any HTML content.

By default, the REST API sets CORS headers at the rest_pre_serve_request filter. To change the headers sent there, you should remove the function rest_send_cors_headers that's hooked by default and provides default CORS headers.

One option is to allow requests to originate from any origin by setting the CORS header appropriately. You can accomplish that like this:

```
remove_filter( 'rest_pre_serve_request', 'rest_send_cors_headers' );
add_filter( 'rest_pre_serve_request', function( $value ) {
	header( 'Access-Control-Allow-Origin: *' );
	header( 'Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE' );
	header( 'Access-Control-Allow-Credentials: true' );

	return $value;

});
```

While this is the easiest option, it may not be preferable in most situations. You can replace the asterisk in the above code with a specific URL to allow that site to access the REST API remotely.

You cannot, however, add more than one CORS header. If you want more than one site to have access, you will need to detect the referring site before setting the header. This will let you set the header if you want to allow requests to originate from the referring site.

You can do this by defining an array of acceptable domains and checking if the origin domain (i.e. the value of the $_SERVER['HTTP_ORIGIN'];) is in that array. If it is, you can set it as the allowed domain.

Here's how it works:

```
remove_filter( 'rest_pre_serve_request', 'rest_send_cors_headers' );
	add_filter( 'rest_pre_serve_request', function( $value ) {

		$origin = get_http_origin();
		if ( $origin && in_array( $origin, array(
				'http://torquemag.io',
				'http://wordpress.org

			) ) ) {
			header( 'Access-Control-Allow-Origin: ' . esc_url_raw( $origin ) );
			header( 'Access-Control-Allow-Methods: POST, GET, OPTIONS, PUT, DELETE' );
			header( 'Access-Control-Allow-Credentials: true' );
		}

		return $value;

	});
```

This example uses the WordPress function get_http_origin(), which is a safe way to get the value of the key HTTP_ORIGIN in the global $_SERVER.

Now you can check if the originated URL request is in the array of allowed domains. If it is, it will check if that domain is used for the CORS header. If not, the CORS header is not set.

Protecting Against Requests For Too Many Posts At Once

On the post endpoint, the posts_per_page filter can be used without authentication. This is a potential security issue if you have a lot of content on your site because requesting too many posts at once could be used as part of a Distributed Denial of Service (DDoS) attack.

This is particularly true if you set a CORS header to allow any domain to access the REST API remotely; and, even if you don't, this is still an issue because origin headers can be faked.

All of the parameters for making post queries via the REST API have a filter. They are created dynamically using the pattern rest_query_var-<name-of-filter>. This means you can override any value set for the posts_per_page filter by hooking rest_query_var-posts_per_page.

I recommend limiting the maximum amount you intend to use in your app. For example, if you intend to use 10 posts per page, you would use this:

```
add_filter( 'rest_query_var-posts_per_page', function( $posts_per_page ) {
	if ( 10 < intval( $posts_per_page )  ) {
		$posts_per_page = 10;
	}

	return $posts_per_page;

});
```

This filter only takes effect if the number of posts per page exceeds 10. If it does, then the value is changed to 10. As a result, a request to wp/v2/posts?filter[post_per_page]=5 would return five posts, while a request to wp/v2 /posts?filter[post_per_page]=50000 would return 10, not 50,000.

Optimization

The REST API doesn't reinvent how posts or other information are queried from the database. It uses WP_Query for posts, WP_User_Query for users, and so forth. As a result, most of the same methods used to optimize our sites can be applied here. For example, WP_Query leverages object caching, therefore using a persistent object cache — such as Memcached or Redis — to improve its performance no matter how it is used.

Page caching is one of the most common WordPress optimization strategies, but it has no effect here because it serves static HTML files of visitors to the front-end of your site. These requests are not generating front-end HTML. The REST API came through with a filter that allows us to intercept requests and serve a response directly.

I created a very simple cache for REST API requests using the WP-TLC-Transients library to handle the caching. This simple caching system checks if it has already served a request for the same URL and, if it has, it returns that response. If it hasn't, it allows the REST API to build the response and then caches it before serving the response.

This is all made possible via the json_pre_dispatch filter. This filter exposes two variables: $result and $server. By default, $result is false. If this filter does not return false, its value is then used as the response and the rest of the plugin is skipped. The other variable, $server, is the JSON server itself, which allows you to properly format your results.

Here is a slightly simplified version of my REST API cache library, which can be used to cache results, like this:

```
add_filter( rest_pre_dispatch', 'jp_rest_cache_get', 10, 2 );
function jp_rest_cache_get( $result, $server ) {

     //make sure rebuild callback exists, if not return unmodified.
       if ( ! function_exists( 'jp_rest_cache_rebuild') ) {
              return $result;

       }

       //get the request and hash it to make the transient key
       $request_uri = $_SERVER[ 'REQUEST_URI' ];
       $key = md5( $request_uri );

       //return cache or build cache
       $result =  tlc_transient( __FUNCTION__ . $key  )
              ->updates_with( 'jp_rest_cache_rebuild', array( $server  ) )
              ->expires_in( 600 )
              ->get();

       return $result;
}
```

The TLC-Transient-Library looks for an existing transient name for the hashed value of the URL that was requested. If it's not found, it calls the function jp_rest_cache_rebuild to generate the cached value and pass the $server variable to it. This function uses the dispatch method of the server's class to set in motion the REST API's default behavior for building a response, like this:

```
function jp_rest_cache_rebuild( $server ) {
       return $server->dispatch();
}
```

Disabling The Default Routes

The REST API makes it very easy to add custom endpoints. This means that there is no limit to how it can be used. It can be all things to all sites. In the next chapter I will discuss creating custom routes, which for app development, is almost always a better idea than using the default routes.

Default routes were not specifically designed for your app, which might not need all of the standard routes that the WordPRess REST API provides. If that is the case, you might consider removing some or all of the default routes.

You should be very careful about doing so. The default routes will be counted on by many third party tools. In the future, it is likely that the WordPress admin will begin to utilize the REST API, and removing the default routes will break the admin.

You can remove the registration of the default routes, wholesale, by removing the hook where those endpoints were are added, like this:

```
remove_action( 'rest_api_init', 'create_initial_rest_routes', 0 );
```

Removing a specific endpoint is a little trickier. To do so you will need to use the rest_endpoints filter, which exposes all registered endpoints, before any request are served.

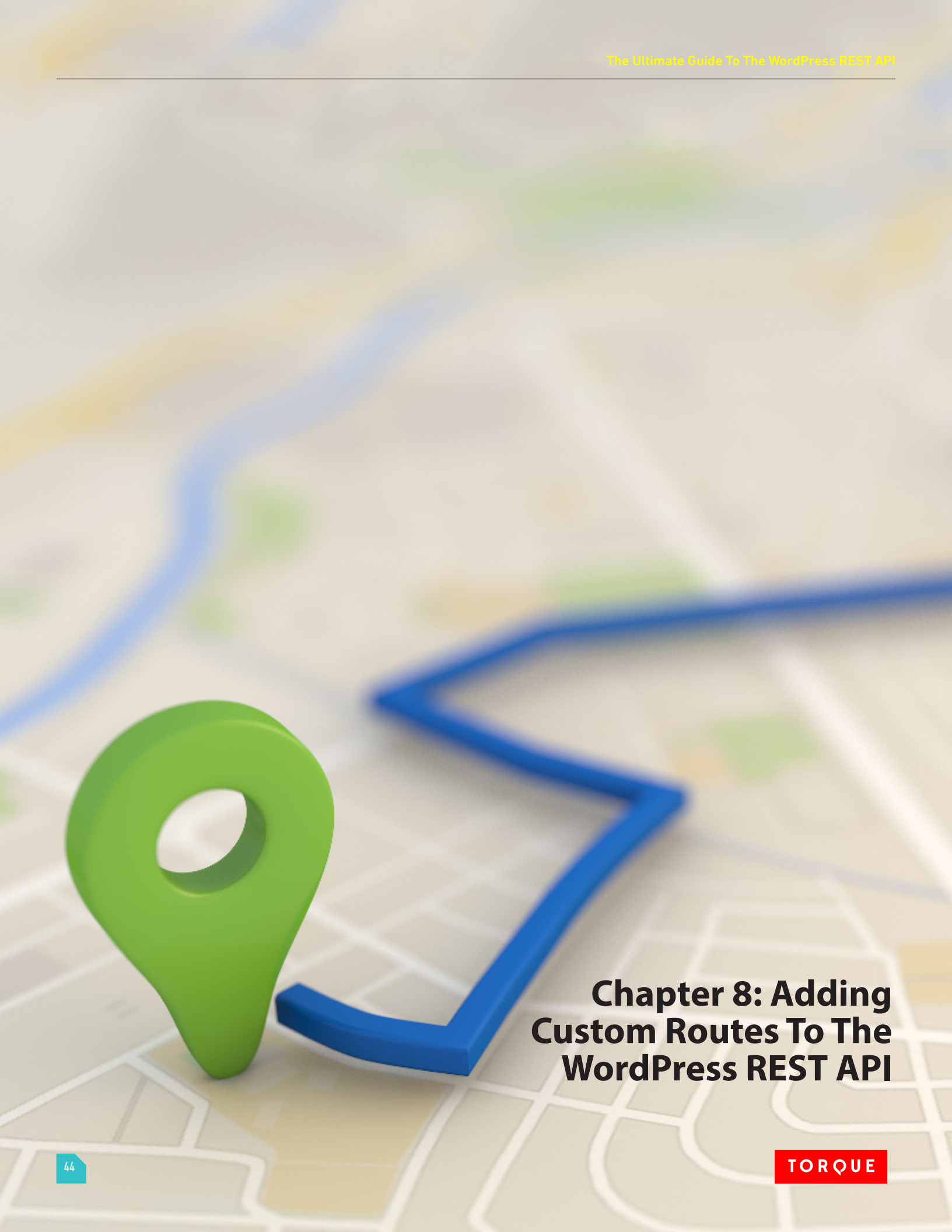
If, for example, you wanted to remove all endpoints for media, you would need to search the URLs for the endpoints in the wp/v2 namespace that include "media" somewhere in their URL, like this:

```
add_filter( 'rest_endpoints', function( $endpoints ) {
  foreach( $endpoints as $endpoint => $args ) {
     if ( isset( $args[ 'namespace' ] ) && 'wp/v2' == $args[ 'namespace' ] ) {
        $parsed_route = explode( '/', $endpoint );
        if ( in_array( 'media', $parsed_route ) ) {
           unset( $endpoints[ $endpoint ] );
        }

     }

  }

  return $endpoints;
});
```

Notice that we are only working in the namespace for the default routes, which means that no custom media routes will be affected.

Now We're Ready To Rock

Not all of the information in this chapter is applicable to every single site; however, you should keep these issues in mind to keep your WordPress site performing as well as possible when used as the backend for a web app. This chapter applies equally for using the REST API to power another site or application, or to integrate with one or more sites or services.



Chapter 8: Adding Custom Routes To The WordPress REST API

Most of the discussion around the WordPress REST API has been about querying the default routes. In that sense, it's treated as a monolithic API — like the Twitter API, for example.

The WordPress REST API is not one API, but millions of highly-customizable APIs that can be leveraged as a tool for making APIs.

Although it comes with default routes, by necessity, those routes are a compromise between tens of millions of sites, including many that haven't even been made yet.

Just like WordPress isn't just the global WP_Query object, the REST API isn't just the default API. Sticking to defaults is like having a traditional WordPress project without ever creating your own WP_Query object or overwriting the default query at pre_get_posts. It's possible, but not every job can be done with the default WordPress URL routes alone.

The same is true with the REST API.

REST API's lead developer, Ryan McCue, has spoken about how the project is split into two parts: default routes and the infrastructure for creating RESTful APIs. The default routes provide great examples of how to create your own routes.

The system used for adding these routes and endpoints is incredibly well done. In this chapter, you'll learn how to use it and how to create a custom route with two endpoints that show information about products in an ecommerce site powered by Easy Digital Downloads (EDD).

This example is based on an API add-on that I built for my own site. You can see the full source on GitHub.

Alternatively, you can view the API in action here.

Although EDD does provide its own RESTful API, I want to expose the specific custom fields that I use on my own site. In my implementation, I incorporated a second route called docs, which is wrapped around a custom post type that I use for documentation. For simplicity, I made my own routes and endpoints.

Adding Routes

Meet My New Favorite Function

Let's talk about a function called register_rest_route(). This lets you add a route to the REST API and pass in an array of endpoints. For each endpoint, you don't just provide a callback function for responding to the request, but you can also define what fields you want in your query — this includes defaults, sanitation, and validation callbacks, as well as a separate permissions callback.

There are additional features here that are still evolving. I recommend reading the class for the default posts routes.

I'll focus on these three things: callback, field arguments, and permissions check. These three functions will illustrate how the architecture of the API works. They're also really useful because once you get to your callback, you will have all of the fields you need, and they will all be sanitized. You will also know that the request is authorized.

This architecture enforces separation of concerns and helps keep your code modular.

Setting Up The Route

When defining a custom route, use the register_rest_route() in a function hooked to rest_api_init, which is the action that runs when the REST API is initialized. It's an important action that will likely be as valuable as plugin_loaded and init.

This function accepts four arguments.

The first is the namespace for the route. All routes must be namespaced, which is then used as the next URL segment after wp-json. The default routes are namespaced with wp. This means that the core routes have URLs like wp-json/wp/posts while a custom route "sizes" in the namespace happy-hats-store would have the url wp-json/happy-hats-store/sizes.

These act like PHP namespaces or unique slugs for a plugin's functions. They avoid clashes between routes so that if you write a plugin that adds a route called menus, it can be used side by side with a plugin that adds a route called menus — just as long as you use different namespaces that correspond to the plugin's name. Namespaces for routes are a smart system since it's likely that two or more developers will add routes with the same name.

The second argument is the URL after the namespace for your route. In this example, my first route is /products and the second is /products' . '/(?P<id>[\d]+). The second route allows for a number, for example, a post ID, in the last URL segment. These route URLs get joined to the namespace. So, if your namespace is chewbacca-api and your route is /products, then the URL will be /wp-json/chewbacca-api/products.

```
register_rest_route( 'chewbacca-api', '/products', array() );
```

It's good practice to include a version number in your namespaces. I used calderawp_api/v2 for my namespace.

The third argument is where the real magic happens—it's where you add endpoints to a route. That's what the rest of this chapter is about, so let's skip it for now.

The fourth and final argument is an optional boolean argument, called override. It is there to help deal with clashes that may occur intentionally or unintentionally with already defined routes. By default, this argument is false; and when it is false, an attempt will be made to merge routes. You can optionally set this to true to replace already declared routes.

Setting Up Your Endpoints

So far we talked about setting up routes, but routes are only useful if they have endpoints. For the rest of this chapter, we will talk about adding endpoints to the route using the third argument of register_rest_route().

Transport Method

All endpoints need to define one or more HTTP transport methods — GET, POST, PUT, and DELETE. By defining an endpoint as only working via GET requests, you are telling the REST API where to get the correct data and how to create errors for invalid requests.

In the array that defines your endpoint, you can define your transport methods in a key called methods. The class WP_REST_Server provides constants for defining transport methods and types of JSON bodies to request. For example, here is how you would define an endpoint that allows for GET requests only:

```
register_rest_route( 'chewbacca-api', '/products', array(
        'methods'         => WP_REST_Server::READABLE,
) );
```

And here is how you would add a route that accepts all transport methods:

```
register_rest_route( 'chewbacca-api', '/products', array(
        'methods'         => WP_REST_Server::ALLMETHODS,
) );
```

Using these constants ensures that, as the REST server evolves, your routes are properly set up for it.

Defining Your Fields

The specification of fields — like what their defaults are and how to sanitize them — is a useful way to define endpoints. It allows the callback function for processing the request to actually trust the data it's retrieving. The REST API handles all of this for you.

Here's an example of how I set up the fields main endpoint that returns a collection of products:

```
register_rest_route( "{$root}/{$version}", '/products', array(
              array(
                     'methods'         => \WP_REST_Server::READABLE,
                     'callback'        => array( $cb_class, 'get_items' ),
                     'args'            => array(
                           'per_page' => array(
                                  'default' => 10,
                                  'sanitize_callback' => 'absint',
                           ),
                           'page' => array(
                                  'default' => 1,
                                  'sanitize_callback' => 'absint',
                           ),
                           'soon' => array(
                                  'default' => 0,
                                  'sanitize_callback' => 'absint',
                           ),
                           'slug' => array(
                                  'default' => false,
                                  'sanitize_callback' => 'sanitize_title',
                           )

                     ),

                     'permission_callback' => array( $this, 'permissions_check' )
              ),
       )

);
```

You will notice that most of these are number or boolean fields, so they're set up to be sanitized using absint(). There is one field for querying by post slug, santize_title, since that is the same way they are sanitized before being written to the database.

My other route is for showing a product by ID. In that route's endpoint I didn't specify any fields because the ID passed in the last URL segment is enough.

```
register_rest_route( "{$root}/{$version}", '/products' . '/(?P<id>[\d]+)', array(
      array(
         'methods'         => \WP_REST_Server::READABLE,
         'callback'        => array( $cb_class, 'get_item' ),
         'args'            => array(

         ),
         'permission_callback' => array( $this, 'permissions_check' )
      ),
   )
);
```

You can use these examples to craft your own routes. Just keep in mind that these examples are written in object context, i.e. they are going to be used inside a method of class. Also, that method needs to be hooked to rest_api_init.

The Callback Function

The callback function, which can be specified for each route in the key callback, is the method that the request will be dispatched to if the permissions callback passes.

In the preceding example, the main route is passed to a method of the callback class called get_items, and the single product route to a method called get_item. This follows the conventions set out in the core post query class, which is important because the callback class actually extends the class in the core API WP_REST_Post_Controller. This allows you to absorb a lot of its functionality while defining your own routes.

The Permissions Callback

Like the main callback, this permissions callback is passed an object of the WP_Request_Class, which allows you to use parts of the request for your authentication scheme. The permissions callback just needs to return true or false; how you get there is up to you.

One strategy is to apply the traditional check of the current user capabilities logic that's used in WordPress development. This is possible because the permissions check will run after the current user is set; so if you use any of the authentication methods, they will already have run.

You do not have to rely on WordPress's current user or authentication at all. One thing you can do is add specific authorization for your custom routes and check the specific parts of the request for the right keys.

If your site has implemented social login, you could check for the oAuth keys, authenticate them against that network, and, if they pass, login the user who is associated with that account.

I'll discuss these strategies more in the future.

For the example here, I will show how to create a public, read only API, so we can either create one function that always returns true to use as our permissions callback, or use WordPress's __return_true. I went with the first option, so I would have it in place for the future when I will start adding authenticated POST requests.

Processing And Responding To Requests

The callback function for each endpoint will be passed an object of the WP_REST_Request class.

Most of the time, you can just use the method get_params() to get all of the data from the request sanitized and validated, with all of the defaults filled in. This gives you the parameters from the request, mapped from whichever transport method we provided. Using this method, instead of accessing the global POST or GET variables is important for several reasons.

First, the array that's returned is validated and sanitized. It also handles the switches between transport methods. This means that if you switch the endpoints definition from using GET to using PUT (that's a one-line change), all of the code in the callback will work just fine.

It also leads to better abstraction. I'm showing a basic version of my API add-on plugin in this chapter, but if you look at the source for the plugin it's based on, you'll see that all of the queries for the products' and docs' endpoints are handled by an abstract class that handles creating WP_Query arguments by looping through the results and returning them.

Regardless of how you handle your endpoint processing, you will want to end with an instance of the WP_REST_Response class. The best way to do so is by using the function ensure_rest_response(), which returns an instance of this class, and can also handle errors well.

This class ensures that your response is properly formed JSON and has the minimum headers necessary. It also provides methods for adding extra headers.

Here you can see how I used it to add headers based on how the core post routes add headers for total results, pages, and previous/next links:

```
/**
 * Create the response.
 *
 * @since 0.0.1
 *
 * @access protected
 *
 * @param \WP_REST_Request $request Full details about the request
 * @param array $args WP_Query Args
 * @param array $data Raw response data
 *
 * @return \WP_Error|\WP_HTTP_ResponseInterface|\WP_REST_Response
```

```
 */
protected function create_response( $request, $args, $data ) {
   $response    = rest_ensure_response( $data );
   $count_query = new \WP_Query();
   unset( $args['paged'] );
   $query_result = $count_query->query( $args );
   $total_posts  = $count_query->found_posts;
   $response->header( 'X-WP-Total', (int) $total_posts );
   $max_pages = ceil( $total_posts / $request['per_page'] );
   $response->header( 'X-WP-TotalPages', (int) $max_pages );


   if ( $request['page'] > 1 ) {
      $prev_page = $request['page'] - 1;
      if ( $prev_page > $max_pages ) {
         $prev_page = $max_pages;
      }
      $prev_link = add_query_arg( 'page', $prev_page, rest_url( $this->base ) );
      $response->link_header( 'prev', $prev_link );
   }

   if ( $max_pages > $request['page'] ) {
      $next_page = $request['page'] + 1;
      $next_link = add_query_arg( 'page', $next_page, rest_url( $this->base ) );
      $response->link_header( 'next', $next_link );
   }

   return $response;

}
```

It's up to you how to get your data together for the response. You can use WP_Query, wpdb, get_post_meta, or a plugin's built-in functions. It's up to you, it's your API.

These are already skills you have as WordPress developer. In many cases, if you're adding a RESTful API to an existing plugin or site, you should already have classes for getting the data you need.

You can use the REST API to get parameters for those classes from an HTTP request, and then pass the results to the REST API's response class.

In my API, I use WP_Query to get the posts. Here is the method I use to loop through the WP_Query object and get the data I need:

```
/**
 * Query for products and create response
 *
 * @since 0.0.1
 *
 * @access protected
 *
 * @param \WP_REST_Request $request Full details about the request
 * @param array $args WP_Query args.
 * @param bool $respond. Optional. Whether to create a response, the default, or just return the
data.
 *
 * @return \WP_HTTP_Response
 */
protected function do_query( $request, $args, $respond = true) {
   $posts_query  = new \WP_Query();
   $query_result = $posts_query->query( $args );

   $data = array();
```

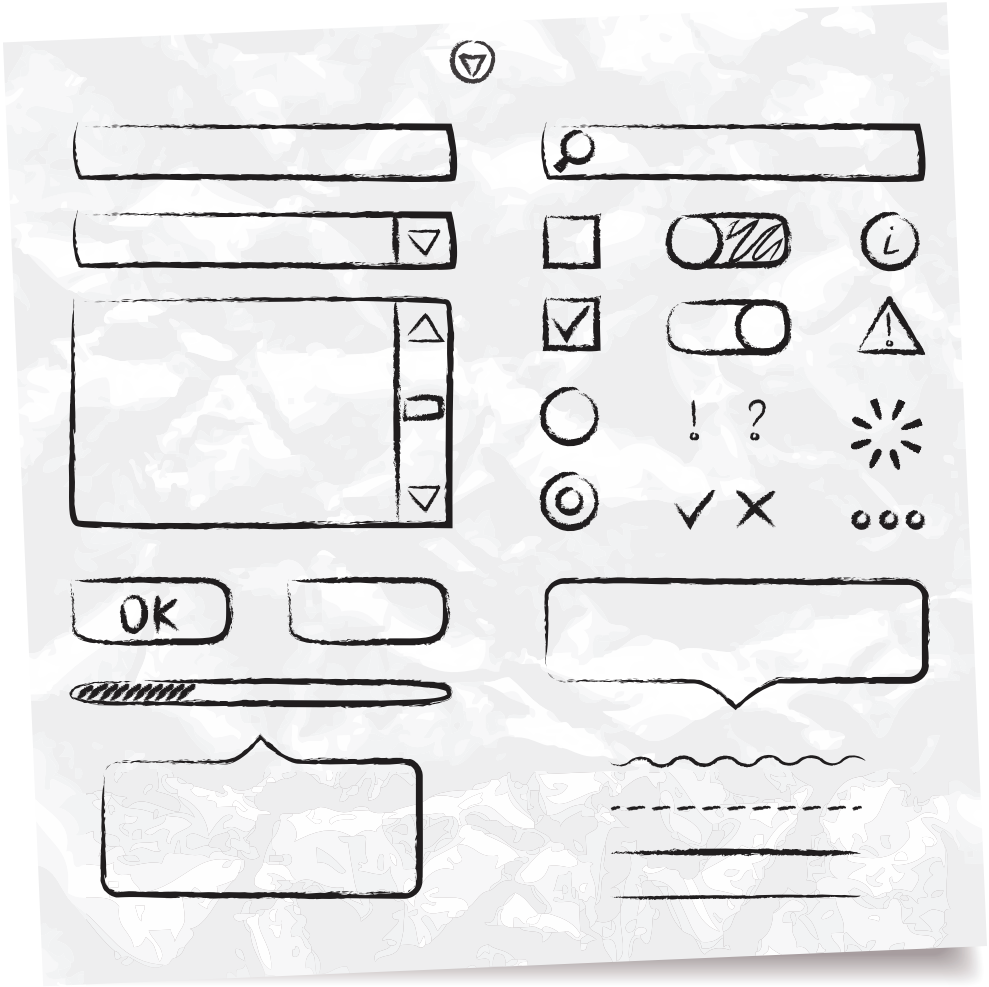
```
   if ( ! empty( $query_result ) ) {
      foreach ( $query_result as $post ) {
         $image = get_post_thumbnail_id( $post->ID );
         if ( $image ) {
            $_image = wp_get_attachment_image_src( $image, 'large' );
            if ( is_array( $_image ) ) {
               $image = $_image[0];
            }

         }

         $data[ $post->ID ] = array(
            'name'         => $post->post_title,
            'link'         => get_the_permalink( $post->ID ),
            'image_markup' => get_the_post_thumbnail( $post->ID, 'large' ),
            'image_src'    => $image,
            'excerpt'      => $post->post_excerpt,
            'tagline'      => get_post_meta( $post->ID, 'product_tagline', true ),
            'prices'       => edd_get_variable_prices( $post->ID ),
            'slug'         => $post->post_name,
         );

         for ( $i = 1; $i <= 3; $i++ ) {
            foreach( array(
               'title',
               'text',
               'image'
            ) as $field ) {
               if ( 'image' != $field ) {
                  $field                          = "benefit_{$i}_{$field}";
                  $data[ $post->ID ][ $field ] = get_post_meta( $post->ID, $field, true );
               }else{
                  $field                          = "benefit_{$i}_{$field}";
                  $_field = get_post_meta( $post->ID, $field, true );
                  $url = false;

                  if ( is_array( $_field ) && isset( $_field[ 'ID' ] )) {
                     $img = $_field[ 'ID' ];
                     $img = wp_get_attachment_image_src( $img, 'large' );

                     if ( is_array( $img ) ) {

                        $url = $img[0];
                     }

                  }
                  $_field[ 'image_src' ] = $url;
                  $data[ $post->ID ][ $field ] = $_field;
               }

            }

         }


         return $data;
      }
   }

   if ( $respond ) {
      return $this->create_response( $request, $args, $data );
   } else {
      return $data;
   }

}
```

As you can see, it is a mix of post fields, meta fields, and functions defined by EDD. If you want to see the full source, including how I added a second route and more endpoints to each of the routes, head over to GitHub and take a look.

Custom APIs, Like Winter, Are Coming

The fact that the WordPress REST API is adding a useful set of default routes to your site is extremely useful. What's even more exciting, and more useful, however, is that in service of creating routes, you are getting a really awesome RESTful API server that gives you powerful tools for creating your own custom APIs.

So, next time you can't make a default route do exactly what you want, just make your own.



Chapter 9: The WordPress REST API And Web Apps

At this point, it's known that you can, in fact, use WordPress to power both web and mobile apps. However, this poses another question: Should you use WordPress to power mobile apps, and, if so, how?

These questions don't have one simple answer. In fact, they are highly contingent on a number of variables: project, budget, requirements, and the competencies of those involved.

In this chapter, you'll learn how to determine if it's the right time to start using WordPress to power your mobile app. You'll also learn the first steps in building a mobile app using WordPress and the WordPress REST API.

Do You Know Why You're Using WordPress To Build Your App?

Building web applications is different than building WordPress sites, themes, or plugins. Similarly, how you use it may invalidate some of the reasons you originally chose WordPress. As a result, you need to think through which reasons still apply, and decide if it still makes sense to use WordPress at all.

Here are some of the big reasons to use WordPress to build a mobile app:

- Its highly extensible content management and editing
- It has a solid and secure relational database that is easy to query
- It has a totally customizable REST API
- It has a strong commitment to backward-compatibility
- It uses PHP — a mature, solid, and rapidly improving language
- It has a powerful and extendable URL mapping system — for example, WP Rewrites
- You're already using and working with WordPress

Will You Use The WordPress Backend For Administration?

One of the great things about using WordPress to build a web app is that, while you are building it (and even before you have a working front-end interface), you still have a completely functional backend. This means you can still use the WordPress backend, as well as WP-CLI and the REST API, while you are developing and testing.

Once the app functions properly, you'll need to ask yourself if you need front-end administrative interfaces, or if your privileged users will still use the WordPress backend to do things like create content and moderate users.

This also extends to managing WordPress and plugin updates. Will you use WordPress's plugin and theme admin or will you use Composer or Git to keep dependencies up to date?

Personally I don't see a need to create too many redundant admin features. In fact, I love having wp-admin available to me, as it allows me to avoid accounting for every possibility in the first version of my app. If something needs to be fudged, an admin can do it.

As far as managing dependencies, Composer is the way to go. It makes it easy to keep dependencies in sync between production and development environments.

Will Your End-Users Use The WordPress Backend?

Many apps require user interaction and front-end data input. For this type of application, you'll need to create a front-end interface for the interactions.

For this, knowing what interactions and what types of data should be input is incredibly important.

Will You Decouple Your Front-end(s)?

When deciding whether or not you should decouple your front-end(s), don't forget to consider how users will submit data. If you're not decoupling your front-end, using a form-builder plugin for building the interactive UI is a huge timesaver.

I used Caldera Forms to create a large part of my UI, and it has saved me time getting to MVP because I inherit its layout tools, validation, sanitization, and entry tracking.

For every form-builder plugin you use, be sure you can easily move the form configuration between environments. This generally means opting not to store the form configuration in the database. Also, think through what happens if or when you decouple the front-end of your app from WordPress, as most of these plugins are not designed to work in this scenario.

URL mapping can be useful for developing mobile apps in WordPress. It uses WordPress to create a response from any URL passed to it, and, based on that URL, performs a query or returns an error page. The URL mapping in WordPress, however, powers its templating system, which, although powerful, might not be the best system for a mobile app. It involves a lot of unnecessary overhead processing, which is strongly coupled with the traditional way WordPress runs queries. This function may be redundant to the API requests that your app is making.

While these are good reasons to decouple your front-end from WordPress, the most significant reason is because it makes it easier to build out multiple front-ends for different platforms — web, desktop, iOS, Android, Windows Mobile, etc.

Using separate front-ends is very useful, but it does create a challenge for me because WordPress is my core competency. Until I master a front-end framework, or partner with someone who has, I'm going to keep coming back to what I know — WordPress's templating and routing system.

Will You Use An MVC Framework For Your Front-End?

People assume that to make an app you have to implement the model view controller (MVC) pattern. This is not entirely true, however MVC frameworks are very helpful in creating a maintainable and scalable app.

This does not mean that you can't use an MVC framework. All you really need is a RESTful API, like WordPress's, and a framework in a language that you know. Since you are presumably a WordPress developer — i.e. PHP and JavaScript developer — you already understand that when talking about MVC frameworks, you are likely talking about a JavaScript MVC framework.

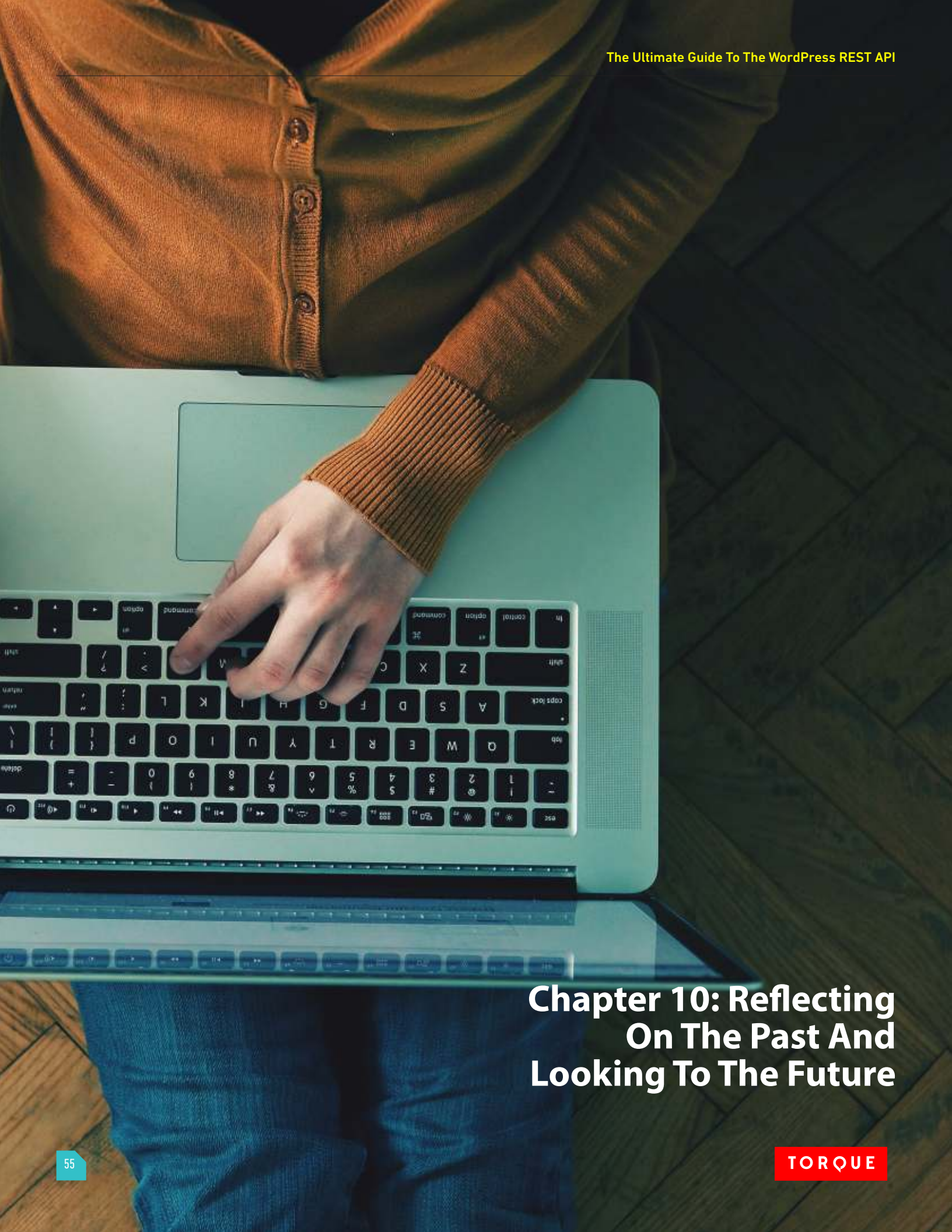
There are several popular JavaScript MVC frameworks, including Angular, React, and Backbone. If you're already familiar with one, then the choice of which one to use is obvious.

An important question to ask yourself — if you (or your team) don't know any of these frameworks — is if you should even use one. It's not required; and, while they are extremely helpful, learning one converts you from an expert to a novice.

New World, New Questions

WordPress has added custom post types, improved the backend interface, and added a RESTful API; and, in doing so has matured into a great application framework. That's awesome for WordPress developers because it sets us up to do work that is more complex, fun, and profitable.

That said, it changes a lot of things. Knowing what has changed, what hasn't, and which questions we need to answer before starting a project is important.



Chapter 10: Reflecting On The Past And Looking To The Future

The Universal API

I asked Ryan about the challenges of making a "universal API" to be included in so many sites. He said that the biggest challenge is balancing all of the different concerns. Adding the default endpoints to core has a lot of complexity in terms of ensuring the right data is there. Balancing the varied and sometimes competing concerns is also a major challenge. This is why it has taken much longer than expected.

He sees two primary core use cases for the API — both of which have conflicting needs.

The first use case is what he calls "service developers." An example of this type of user is a news organization that wants to create an app based on their website providing API data as a service to the app. These users are more likely to use custom endpoints. So, for them, the infrastructure is more important than the default endpoints themselves.

A good example of this is Wired Magazine. Wired replaced multiple WordPress installs with one install, linked to multiple front-ends via the REST API. Ryan's favorite part about this project is that the REST API will allow developers to choose their own way of working with WordPress. No longer will developers have to use a theme in the traditional way. They can use the JSON data any way they want.

"That's one of the use cases I really like," Ryan said."The fact that someone who doesn't know WordPress that well can go and start creating front-end websites with this. It expands how many people can use and develop for WordPress."

The other use case Ryan predicts is WordPress-powered mobile apps. For these cases, the goal is to make the API work "consistently across every site in existence." This leads to an impossible expectation of the same data being provided from every site.

"Some of these problems we don't want to try and handle because there is way too much there," Ryan said. "The flip-side of all of this is that even if we know the sites have the same data, they may not have the same fields... We are trying to work out the best balance... between the flexibility we need to allow for plugin developers and the strictness that clients need."

The Ever-Expanding Scope Of The REST API

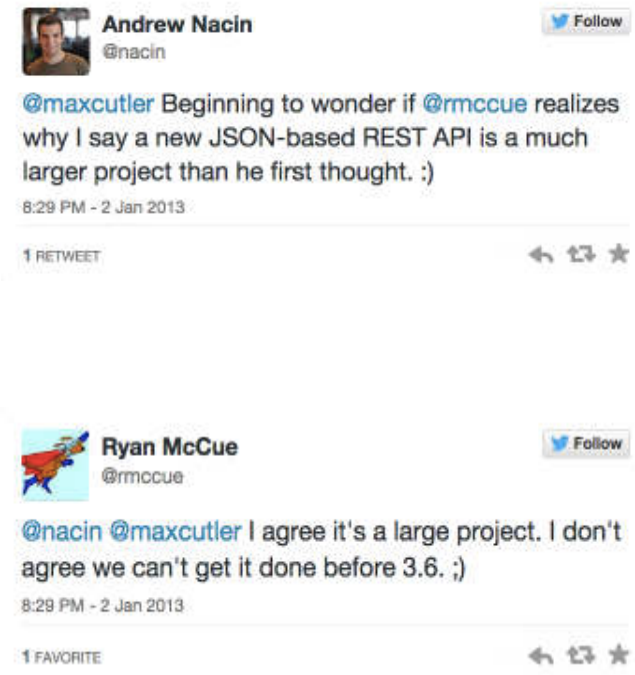
Ryan worked on the WordPress REST API for a very long time. To give you an idea of exactly how long that is, when he started, he thought it could be included in WordPress 3.6. The first half wasn't introduced until 4.4 in 2015.

The extended time frame for this project was not just because he underestimated the amount of work, but also because the scope of the project grew tremendously over time.

Originally, he saw the REST API as " just a way to get at the underlying data in WordPress and not much more than that." Some of the code from the first release, which was very minimalistic and handled a much smaller amount of data, still exists to this day.

The limit on just the four core objects—posts, users, taxonomies, and meta data—was decided early on. Although this may seem somewhat limiting, it sets up the REST API to cover options and other data types.

Like WordPress core, the REST API is designed to be extensible. It includes the infrastructure for handling the rest of the data types or doing whatever you want as a developer.



"The infrastructure of the API itself, supports basically anything you can throw at it. If you take away the core endpoints, it is essentially a framework for building APIs, and you can build those however you like," said Ryan. In his mind, the biggest misconception about the REST API project is that people are "going to see it as a magic bullet that solves all of these problems," Ryan added.

Take generic response handling, for example. A mobile app cannot automatically handle every single response that any WordPress site will be able to send to them. The WordPress REST API doesn't satisfy WordPress's need for a generic API.

"We want to allow this flexibility to plugin developers," Ryan said, "plugin developers also have to understand that with the flexibility they are granted the have to take into mind things like this, and it is tough to convey this, especially for service sites... Trying to convey to people that they can remove endpoints, but they need to be aware that this will break stuff. Adding data to a response will be easier to do, as it's not a big problem for clients, but it will be harder to remove data that is expected by a client."

Forward The Foundation (For Your Own API)

Ryan and more than 50 contributors were tasked with adding a RESTful API to tens of millions of websites.

Before speaking with Ryan, I didn't understand that this project is about more than just creating a perfect API that works for tens of millions of sites. It's really about adding a standard set of tools for anyone to make their own API, as well as a totally functional, but generic API.

The REST API is a reflection of WordPress itself. WordPress is a tool that provides a basic starting point for a CMS in five minutes or less, which can be expanded on pretty much infinitely. The REST API is no different. It provides a fully-functional API out of the box, ready for anyone to use, with all the power needed for site and plugin developers to easily customize.

Chapter 11: Conclusion

You should now have a solid primer on how to get started with the WordPress REST API.

I was introduced to the WordPress REST API at WordCamp Milwaukee in 2014. Since then, the REST API has continued to gain momentum t has been added to WordPress core and, it now symbolizes the hope and enthusiasm of an entire generation of WordPress developers — developers who see WordPress and its community as an application framework out to democratize the internet.

Adding a universal API to tens of millions of WordPress sites on the internet was an unbelievable challenge, but the history of open-source software is about developers who aren't jaded enough to shy away from a seemingly impossible challenge.

Ryan McCue started the REST API project while he was a student at Google's Summer of Code. He was the driving force behind the project. I met him and co-lead developer Rachel Baker the day before WordCamp Milwaukee at an informal gathering of speakers before the speaker and sponsor dinner. Their introduction to the WordPress REST API session went on to be a hit of the WordCamp.

Although I knew about this project, and RESTful APIs, prior to WordCamp Milwaukee, the truth is they always intimidated me — I guess they seemed too advanced for me.

Like many WordPress professionals, I don't have any formal training as a programmer. I just Googled how to do what I now do for a living. It took me a long time to get over the inferiority complex that comes with that.

I learned about the WordPress REST API mainly by writing about it for Torque — which has since been collected and updated for the purpose of this ebook.

I believe that my relationship with the WordPress REST API reflects that of the community at large. This reflection is something we all see, and is part of our enthusiasm for it.

The WordPress REST API really does represent everything that's makes WordPress so powerful: A small team of developers from all over the world, working for different companies, coming together to create something we all need, and attracting 50 other minor contributors, and the support of an entire community along the way.

The presence of the WordPress REST API is allowing developers and entrepreneurs, both inside and outside of our community, to use WordPress to power any app or service they are creating. It has brought a new respect to WordPress.

While I do have a few minor commits to the REST API, I haven't been a major contributor to its code base, but I have been able to educate people about it. In an open-source community, we give what we can. So I wrote about it, created working example code for others to build on, and wrote a few plugins to extend it.

A few years ago, while just a teenager in Australia, Ryan McCue thought that WordPress needs a RESTful API, and instead of complaining about how terrible WordPress was for not having one, he started working to fix that problem.

At the time, he didn't realize just how much of an undertaking it was, but I'm sure he realized that he was going to need the help and support of the community.

We fix what we can fix, find help with what we can't fix, and give in whatever way we can. It was through this mantra that Matt Mullenweg was able to successfully lead the development of WordPress — a force that currently powers more than 20 percent of the Internet. WordPress empowers its users by enabling them tell personal stories, sell products, create apps, and, in many cases, make a living. I hope this book has helped you understand and appreciate the capabilities and power of the WordPress REST API.

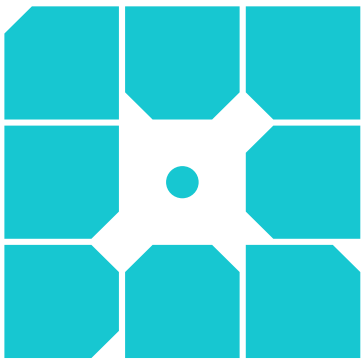
Torque Magazine

Torque is a news site featuring all things WordPress. We are dedicated to informing new and advanced WordPress professionals, users, and enthusiasts about the industry. Torque focuses primarily on WordPress News, Business, and Development, but also covers topics relating to open source and break-through technology.

Torque made its debut in July 2013, at WordCamp San Francisco, and has since produced valuable content that reflects the evolution of WordPress, both as a platform and a community.

Torque is a WP Engine publication, though maintains complete editorial independence.

https://torquemag.io



WP Engine

WP Engine, the WordPress technology company, provides the most relied-upon brands and developer-centric WordPress products for companies and agencies of all sizes, including managed WordPress hosting, enterprise WordPress, headless WordPress, Flywheel, Local, and Genesis. WP Engine's tech innovation and award-winning WordPress experts help to power more than 1.5 million sites across 150 countries.

https://wpengine.com