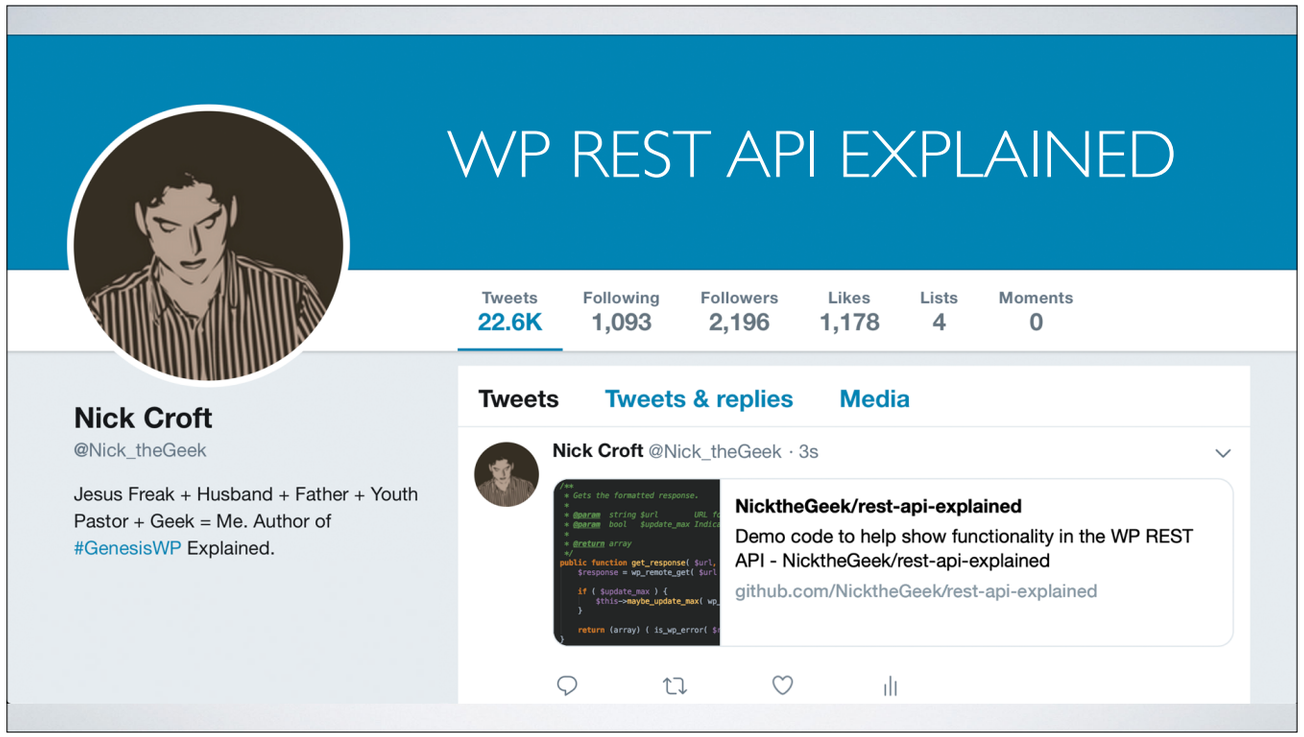
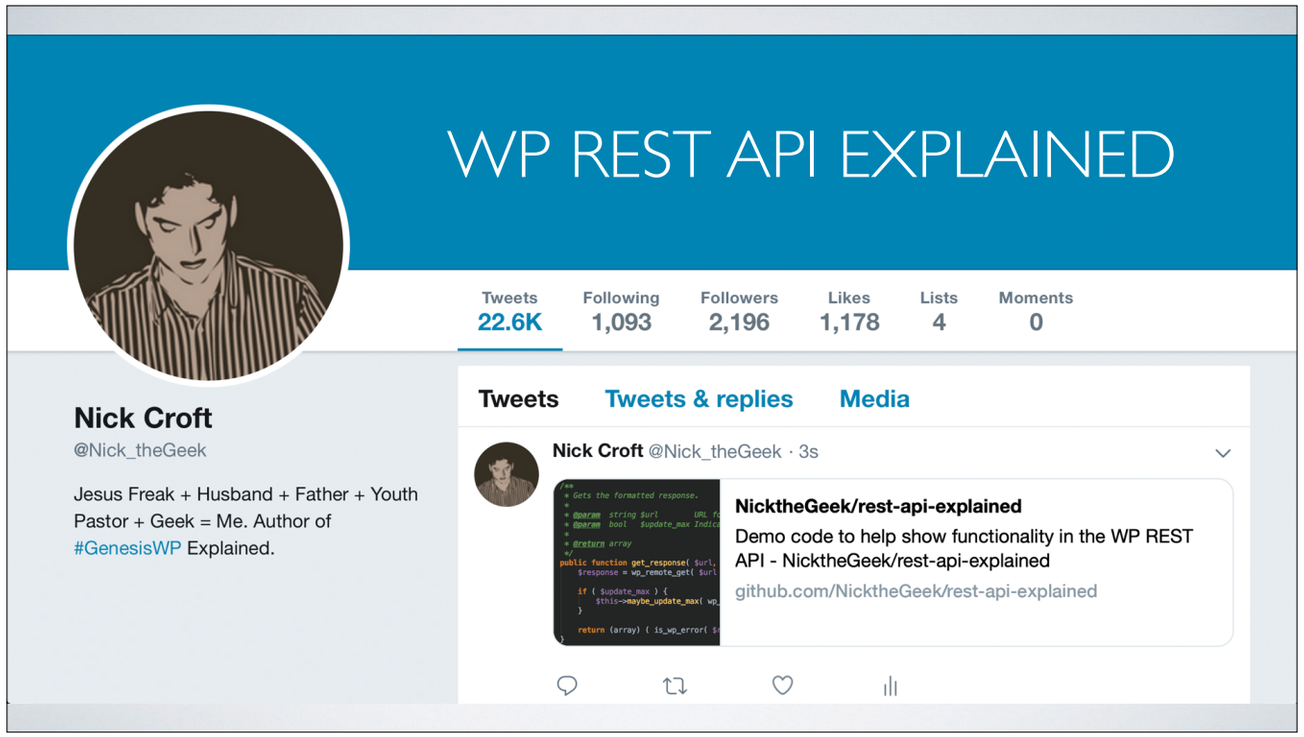


Welcome! Tonight I'll be explaining the WordPress REST API. The goal will be to quickly look at how this tool can be used in sites with real world examples from projects I've completed. While this is going to be a developer focused talk, one of the goals will be to give end users an understanding of how this can help them achieve goals for their sites.

Before getting into the weeds, let me share just a bit about myself.



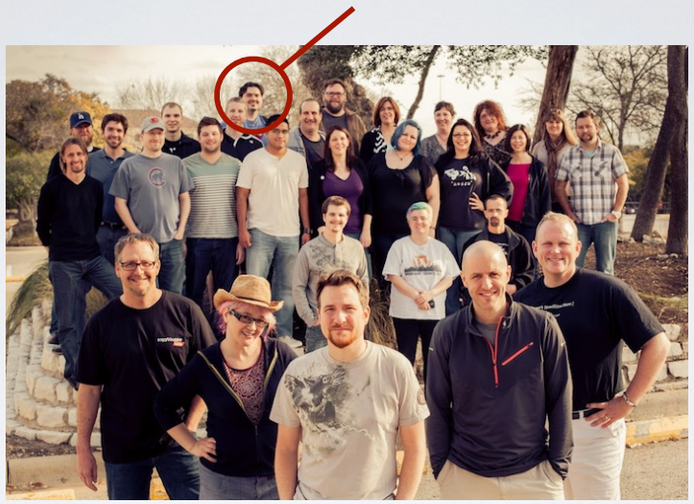
Welcome! Tonight I'll be explaining the WordPress REST API. The goal will be to quickly look at how this tool can be used in sites with real world examples from projects I've completed. While this is going to be a developer focused talk, one of the goals will be to give end users an understanding of how this can help them achieve goals for their sites.

Before getting into the weeds, let me share just a bit about myself.

I got started working with computers as a young man … ok not this young.

But this was my first computer. I started tinkering when I was about 8, figuring out how to change values to make games easier, then as I got better to make the games more difficult. Later I helped build a 386 with dos, a 486 with Windows 3.11, and eventually our first Pentium with Win 95 which was amazing.

My first website was built on geocities and probably looked this amazing! Somehow I moved on and built other sites using HTML, then CSS + HTML, and eventually I picked up PHP and a little over a decade ago I found WordPress.

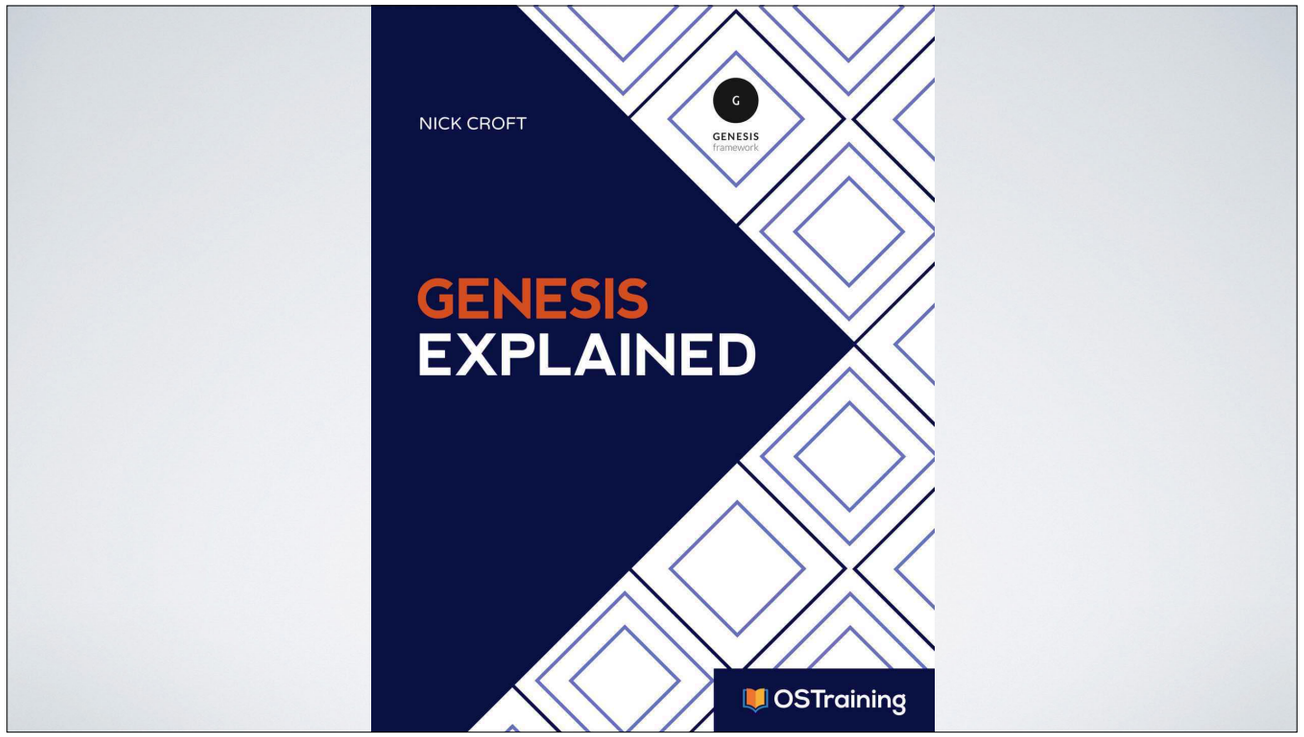


Not long after that I ended up getting involved with the StudioPress/Genesis community and eventually joined the team officially just as they merged with Copyblogger. I'm the tall guy way in the back. Eventually I ended up as senior developer with them before starting work with another great group.

The past year and a half I've been a senior dev with ReaktivStudios, a WordPress VIP partner. The highlight of my career was solving an escape room with the team.

In addition to that I've got 4 amazing kids and a wonderful wife who has put up with me for nearly 20 years.

and the most adorable littler fur baby.



oh, and speaking of Genesis, I also happen to have written a book.

So hopefully everyone can see where I’m coming from :)

REST EXPLAINED

REST is acronym for REpresentational State Transfer

6 Principles for a RESTful API

1. Client server
2. Stateless
3. Cacheable
4. Uniform Interface
5. Layered system
6. Code on demand

During Presentation, skip. This is available in the slide deck download.
REST is acronym for REpresentational State Transfer

6 principles for a RESTful API
Client–server – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.

Stateless – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

Cacheable – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Uniform interface – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.

Layered system – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.

API EXPLAINED

Application Program Interface

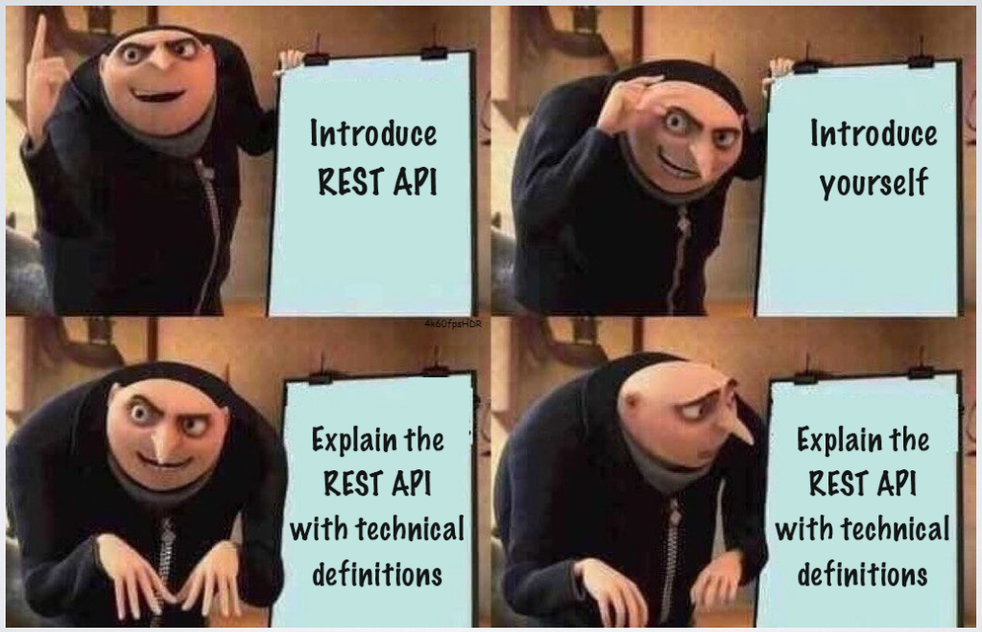
- Plugin API
- Shortcode API
- Theme API
- Over a dozen more

During Presentation, skip. This is available in the slide deck download.
An application program interface (API) is a set of routines, protocols, and tools for building software applications. Basically, an API specifies how software components should interact. Additionally, APIs are used when programming graphical user interface (GUI) components. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

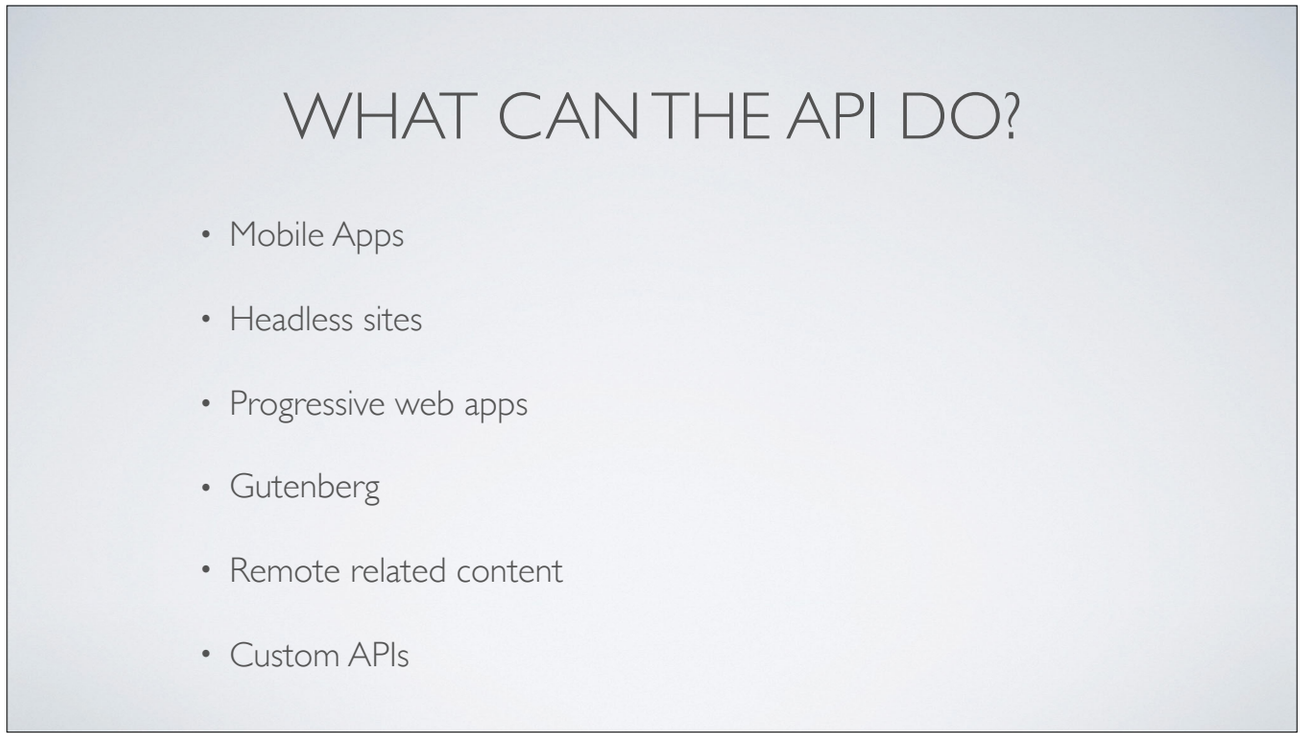
It's important to note that the WordPress REST API is not the only API in WordPress. In fact, there are at least 18 total APIs in WordPress including:
Plugin API
Shortcode API
Theme API

and several others.

These are sets of components, functions, and methods to create themes, plugins, access the database, and do pretty much everything possible in WordPress.



During presentation, previous two slides have been skipped.
OK, so if you are like me, hearing these technical definitions of what a REST API is won't really help, so let's talk just a bit about what the WordPress REST API does. At the simplest it creates a standard way to interact with a WordPress site. Some people might be concerned about leaving this open, but consider that one of the big goals of the REST API is to make it possible to interact with a site without requiring specific design elements. These interactions have always been happening, but with specific HTML, Javascript, and CSS, which slows everything down. It would also require some parsing engine to figure out the content and from one site to the next there are going to be differences because of themes, plugins, and settings. The REST API follows the design principals to decouple all of the extra stuff to speed up the interaction and make it possible to extend in a discoverable way. When themes and plugins extend the REST API following the correct patterns, it means apps can interact with those changes with expected behavior.



Some things the REST API can do include:
- Allow mobile app developers to work with WordPress data without custom APIs
- Allow a single site to serve data to multiple headless sites
- Progressive web apps
- Makes Gutenberg possible
- Remote related content
- Dependable and secure custom APIs

Because the main idea is to expose everything you can do in WordPress in a secure, consistent, and discoverable way, the REST API can do anything you might do via other APIs but available without the limitations cause by the wide variety of markup, styling, and scripts found via combinations of plugins, themes, and settings across the wide web.

For the rest of this talk, I'm going to focus on two specific examples: fetching related content from a remote site and a custom API.

REMOTE RELATED CONTENT

- Embed links to remote content
- Fetch that content automatically
- Remote server is secured and difficult to change, requiring a custom VPN, multiple reviews, committing changes to two different repos, and potential changes out of our control

Several months back we had a client who needed to get links to another site they maintained. The goal was to drive more traffic to that site so they could justify some new expenses for the site and build it into a bigger program that would eventually become a part of the sites we help maintain.

There were several issues that we had to overcome. Because of the way content on the site showing the links was generated, it couldn’t be done manually but security and oversight of the remote site made it very difficult for us to get the changes up within the proposed timeline.

Fortunately I found that the REST API was available and working and after doing some tests to see how the term searches we did on the remote site resulted in acceptable results I started working up a way to request the data.

REMOTE RELATED CONTENT

At this point, let's look at a tool called "Postman" that we can use to query the REST API. It isn't required, you could do this in a browser, via terminal using CURL, or anything else that can make an HTTP request.

That is one of the requirements of a RESTful API.

I like Postman because it is free, and there are some helpful utilities for browsing server responses.

Open https://designsbynickthegeek.com/wp-json/wp/v2/ in PostMan

This is the base URL to access the WordPress API routes. We'll look at other routes in a little bit.

Since the REST API is discoverable, we can get information about the routes available from this point. There are several routes, and I would encourage anyone interested in the API to browse through them. Each route lists most of the possible arguments and the methods for the communication as "endpoints."

The route we will be looking at is wp/v2/posts. As you can see there are two methods available, GET and POST. In general GET retrieves data, POST updates data, and if available DELETE removes data. In this usage it would be similar to WP_Query to get posts or wp_update_post to update a post. Because POST allows modifying the database, it also requires the user to be authenticated. Attempting to POST to this route without proper authentication, which includes user capabilities to create posts (or other post types) will result in an error just like trying to create a post in WordPress admin dashboard without being logged in or having the correct access.

So let's look at the GET request.

CUSTOM API

- Select stores via geo location or zipcode
- Potential to link with internal API in the future
- Fixed timeline with delays accessing the internal API
- Due to caching, everything needed to happen in browser via javascript

There have been several times I've used the WordPress REST API to generate custom routes to serve as an API for specific features. For a recent client we needed a solution that let a central site subscribe to feeds from a few dozen other sites then select posts that would be included in a syndication feed for third parties. The REST API with custom routes made this surprisingly easy so everything could be done centrally.

Further back we had a client with an internal API to allow selecting stores, configuring shopping lists, and several other features that were planned for the site. Unfortunately the API wasn't available to us due to security restrictions. Because of the timeline and the need to handle search, display, select, and some changes in information available on the page all via javascript due to caching that would make it impossible to have the dynamic content from the server.

With the timeline concerns I proposed developing custom routes and using an interface to define the classes so I could work with a mock object first then when we had access to the API I'd be able to convert their API response to match the interface. This means the javascript wouldn't need updates because the data coming from our custom REST routes would remain standardized.

This is an important pattern when dealing with various APIs that may use XML, JSON, or other data formats and will most certainly have differently shaped data. Good practices with an interface can help improve the portability of code when interacting with multiple APIs.

By making the custom routes and using a mock object in order to develop the HTML, CSS, and Javascript I was able to hit the project timeline even when we ended up with a very late change to use a custom JSON list with all of the stores that was created from an XML file the client provided to us instead of getting access to their API.

CUSTOM API

Let’s look at the PHP side of this first. That is where the custom routes are registered and the responses are generated.

The first file to look at is the main plugin file: rest-api-explained.php

This file requires the autoloader, a functions file, and adds an action
add_action('rest_api_init', 'REST\API\Explained\Routes\register_routes');

That is where the routes are registered. We can look at the functions.php file and we’ll see the the register_routes function initializes the Rest->register_routes() method. We’ll look at that in a minute.

One other function to note is the get_locator() function. I’m using that to initiate the mock locator object. In the production code this uses a production locator class.

Another pattern for this would be to initiate the in the register_routes() function and pass the object to the Rest class when it is initialized. I opted against that pattern because I didn’t want to load and initialize the class unless it was needed.

Now let’s look at the inc/classes/routes/class-rest.php file. This is where the routes are registered.

The first item of note is the $version property. This is set at 1 and is used to build the name space for the routes.

This will allow version control to the available routes, but do remember that when you update this version, existing routes will be removed, so if the version is being used,

CUSTOM API

- register_rest_route()
  - $namespace
  - $route
  - $args
  - $override

namespace: this is forward slash separated string and should include a version.

route: this is also a string and can include URL safe characters as well as regular expressions similar to the regex markup used in the rewrite API.

args: There are several args available including method and callback, which is used in these examples. It is also possible to set a permission_callback, which is used to verify the user permission for authenticated requests, and an args argument. When setting the args key you can add an additional associative array that allows additional validation and sanitization for the arguments generated by the request. We'll look at all of this in more detail in a moment.

override: This is a boolean option. True indicates that the route should replace a previously registered route.

Now let's go back to the file: inc/classes/routes/class-rest.php

CUSTOM API

The first route is very simple. We use the $namespace and $base method and in the arguments set the methods to use WP_REST_Server::READABLE and the callback to use the get_current_store method in the object.

One important thing about the WP_REST_Server::READABLE property. It is technically possible to use methods GET, POST, DELETE, etc. directly, but it is a better practice to use the properties in WP_REST_Server because they can be aware of how the server is configured, which can save your sanity in certain instances.

Let's look at a little bit more complex route. The nest includes a regular expression. If you've worked with WordPress's rewrite API, this will be familiar.

Regular expressions can be very powerful and specify the acceptable characters for routes. In this case there is a string, geo, that is included in the route and then two arguments registered in the route. The word inside the greater than/less than brackets is used to set the argument key. The expression following the bracket defines what to look for in the route. For this case it accepts almost anything.

The next route is a little more picky. The regular expression for zip code accepts digits only, which is a level of validation, but let's look at the args for this. I've included a sanitize_callback and validate_callback. To sanitize this, the code uses absint, which ensures the code will be forced as an integer, but it doesn't ensure the zip code is valid.

For the sake of this demo I'm using a simple validate function. The is_zipcode() method checks to see if there are exactly 5 digits in the parameter. This could be updated to account for a potential +4 zip code or match post codes from other countries.

The next route uses the WP_REST_Server::EDITABLE method. This is a POST method and generally requires an additional permissions check since often this means

CUSTOM API

- Javascript Requests
  - jQuery.ajax()
  - wp.api

With the routes registered, there are a few ways to request the data via javascript. jQuery.ajax is a very common method and one many developers are at least casually familiar with. Other methods for requesting a response from a given URL will work as well, but WordPress also has an object that is built to work with the API.

Before we look at the javascript, let's jump back to the main rest-api-explained.php file to see how the script is enqueued and localized.

The script is enqueued and requires jQuery and wp-api. It is certainly possible to do this script with use wp-api but the project for the client was using jQuery, so it is still present in some of the code in this demo.

This also uses wp_localize_script to set the URL for the REST API.

Now let's open up the assets/src/js/locator.js file to look at the wp.api method.

This opens at the end, so let's go there. Since the code is using wp.api it's important to make sure that it is initialized before attempting to use it. The wp.api.loadPromise.done() callback will ensure the API is available.

Now we can call the locator init using wp.api.init(). This accepts an object where the namespace and rest URL are set. This is also a promise, so we can run a done() callback that invokes our init method. This way we know the API is completely initialized before we try to use it. Otherwise we might have instances where the code intermittently fails and no idea why it happens.

Now we can see how it is used. Let's jump up to near the top of the file and look at the get_stores_by_geo and get_stores_by_zip methods.

That brings us to the end. Remember, the REST API is a method for interacting with a WordPress site that is independent of the styling on the site. It is possible to interact with the core API elements and everything is discoverable to make it easier to find the routes. The API is platform agnostic, so you can use PHP, Javascript, or any other method that can make an HTTP request and parse JSON. It is possible to register custom routes, and there is a javascript api for working with the WordPress REST API.

That's an awful lot to cover so before I go into QA I want to remind you that this entire presentation with notes and the demo code are available at github.com/NicktheGeek/rest-api-explained.

Now, does anyone have questions?