WordPress CMS CRUD

The whole idea of the WordPress REST API is to allow developers to use whateveer tools they choose to create front-ends for WordPress systems. We're no longer limited to PHP and the Gutenberg editor and the possibilities are endless.

This post will cover authentication using JWT, listing pages of posts, viewing individual posts and the basic CRUD operations: create, update and delete. We covered setting up JWT authentication in WordPress in Adding JWT to WordPress

We will write this using ES Modules, so we'll have to build a quick WebPack build system to convert it into something that works in browsers.

The build system

Because we will be working with modules we need to write a quick WebPack build scrypt that will use Babel to convert this into something that browsers will read without having to inline scripts with `type=module`.

```
const path = require("path");
const TerserPlugin = require('terser-webpack-plugin');

module.exports = (env, arg) => {
  config = {
    entry: ["./src/index.js"],
    output: {
      filename: "bundle.min.js",
      path: path.resolve(__dirname, "dist")
    },
    devServer: {
      port: 3000,
      https: true,
      contentBase: path.join(__dirname, "dist")
    },
    module: {
      rules: [
        {
```

```
          test: /\.js$/,
          exclude: /(node_modules)/,
          use: {
            loader: /\.js$/"babel-loader",
            options: {
              presets: ["@babel/preset-modules"],
            }
          }
        }
      ]
    },
    optimization: {
      minimize: true,
      minimizer: [new TerserPlugin()],
    }
  };
  if (arg.mode === "development") {
    config.devtool = "source-map";
  }
  return config;
};
```

We then install the Node modules that we'll need for WebPack to run.

The only module I'll mention is @babel/preset-modules. It is an improvement over preset-env in that it fixes additional problems that would cause bundled code to be larger than necessary.

```
npm i -D @babel/cli \
@babel/core \
@babel/preset-modules \
babel-loader \
terser-webpack-plugin \
webpack webpack-cli \
webpack-dev-server
```

We will also install the packages that we'll use directly in the code we write.

```
npm i axios \
form-urlencoded \
js-cookie
```

axios is the package that we'll use to handle fetching data from the server.

form-urlencoded will serialize the data we send to the server

js-cookie provides a programmatic way to set and read cookies. We'll store the JSON Web Token in a cookie that we'll read in different parts of the code.

Create Initial State

We will create a sate constant where we'll set initial values for our authentication, and data storage for the values we get from the API.

We also create a function to add elements to the state. This is a very simplified version of what a state management library would do and, if we use React or Vue it would be unnecessary.

```
// Set state object with values that are changed programatically
const state = {
  loggedIn: false,
  restUrl: "http://localhost:8888/wordpress/wp-json/",
  token: "wp-token",
};

const setState = (toSet, newValue) => {
  state[toSet] = newValue;
};

export { state, setState };
```

Authenticate

To authenticate we first import all the packages that we'll use: Axios, js-cookie and form-urlencoded. We also import our state script.

```
// Import libraries
import axios from "axios";
import Coo"axios"om "js-cookie";
import formurlencoded from "form-urlencoded";

"js-cookie"// Import configs
import { state, setState } from "../state";
```

We then check if the cookie with our JWT token exists, meaning that the user is authenticated.

If it does then we initiate the login process and display the log out button.

It we're not logged in then we display the login form and initiate the logout process.

```
export function init() {
  if (Cookies.get(state.token) === undefined) {
    logout();
    initLogin();
  } else {
    login();
    initLogout();
  }
}
```

The code make the following assumptions:

- We have a login form where the user will enter his username and password
- On successful login the form will be replaced with a logout button

The code proceeds as follows:

1. Capture the username and password values from the form
2. Use Axios to make a POST request to the JWT token endpoint with the url-encoded credentials as the payload data
3. If we get back a 200 response code, and only then, we set a cookie with the token as the value
4. We call the `init()` function

5. If we do not receive a 200 code there was a problem and we tell the user that we couldn't log her in
6. If there was an error with the promise, the `catch()` block is called and we tell the user what the error was

```
export function initLogin() {
  getEl(loginForm).addEventListener("submit", event =>
    event.preventDefault();
    const creds = {
      username: document.getElementById(username).value,
      password: document.getElementById(password).value
    }; // 1

    // Make request to authenticate
    axios({
      method: "post",
      url: state.restUrl + "jwt-auth/v1/token",
      data: formurlencoded(creds),
      headers: {
        "Content-Type": "application/x-www-form-urlencoded"
      }
    }) "post"// 2
      .then(response => {
        if (200 === response.status) {
          Cookies.set(state.token, response.data.token, {
            expires: 1,
            secure: true
          }); // 3
          init(); // 4
        } else {
          // Executed when response code is not 200     alert("Login faile
        }
      }) // 5
 "Login failed, please check credentials and try again!"// 5
      .catch(error => {
        // Also log the actual error
        console.error(error);
      }); // 6
  });
```

```
}
```

The `login()` function does housekeeping. It removes the login form, it shows the listing of posts and does any other task that you need to do once the user is logged in.

```
export function login() {
  // Set the loggedIn status to true
  setState("loggedIn", true);

  "loggedIn"// Remove login form and replace it
  // with log out button

  // Show the listing of posts

  // Do whatever else you need to
}
```

The `initLogout()` function handles the logout functionality.

It is conceptually simpler than `initLogin()`. It removes the token cookie and it calls the `logout()` function.

```
export function initLogout() {
  // Setup event listeners for logout form
  getEl(logoutForm).addEventListener("click", event => {
    event.preventDefault();
    Cookies.remove(state."click"{ secure: true });

    logout();
  });
}
```

The `logout()` function, like its login counterpart, does the housekeeping we need to do to make sure this works.

We set the loggedIn state to false so that the workflow will prompt for a login.

We swap the logout button for the login form and we show the posts available when the user is not logged in.

```
export function logout() {
  // Set the loggedIn statis to false
  setState("loggedIn", false);

  "loggedIn"// Remove logout button and adds login form

  // Show unauthenticated posts listing
}
```

Read Content

To display the content of our WordPress blog we need to do two things: display a list of all posts, like what you'd see in a default WordPress home page and display individual posts.

We'll tackle these tasks separately.

Post Listing

To get a listing of the first posts on the database, 10 by default, we run code similar to the init function below.

We use Axios to make a GET request to the `posts` API endpoint. We save the posts data to our state object and call the render function.

```
export function init(event) {
  // If coming from an event,
  // prevent default behavior
  if (event) event.preventDefault();

  // Make API request with Axios
  axios
    .get(state.restUrl + "wp/v2/posts", {
      params: {
```

```
        per_page: 10
      }
    })
    .then(({ data: posts }) => {
      setState("posts", posts);
      render();
    });
}
"wp/v2/posts"
```

The `render()` function is what creates the content that will create the contetn we display to the users. It is cumbersome and, for a production applications, I would definitely consider using Handlebars or somekid of templating solution but in this case I'm more concernned with the authentication and how it works.

For each post that we get from the database via ou state object we do the following:

1. We create and article element
2. We add the `post` class to the article we just created
3. We then create the content using the article's innerHTML attribute with a Template Literal
    1. The template literal interpolates values from the posts we store in our state object
4. We add an event listener to the post title to link to the post and then call the `post()` function

```
export function render() {
  // Map through the posts
  state.posts.map(post => {
    // Setup the post article element
    const article = createEl("article"); "article"// 1cle.classList.add("p
    "post"// 2nerHTML = `
      <h2 class="entry-title">
        <a href="#${post.slug}">
          ${post.title.rendered}
        </a>
      </h2>
```

```
    <div class="entry-content">
      ${post.excerpt.rendered}
    </div>
  `; // 3

a"entry-title"`
    <h2 class="entry-title">
      <a href="#${post.slug}">
        ${post.title.rendered}
      </a>
    </h2>
    <div class="entry-content">
      ${post.excerpt.rendered}
    </div>
  `// 3

article.querySelector(".entry-title a")
  .addEventListener("click", event => {
    event.preventDefault();
    setState("post", post);
    Post();
}); // 4
```

Single Post

The single post render function does as the title says, it renders the content of a single post on screen and, if the user is logged in, displays links to edit and delete the current post.

The functions does the following:

1. Create the article and a
2. Attach the post class to it.
3. Add the content using innerHTML.
    1. The content includes a link to go back to the post listing; we'll attach an event to the link later
    2. We also use the rendered versions of the title and the content for the post
4. We attach a click event to the back link so that clicking on the link will

display a list of the available posts

5. If the user is logged in, provide links to edit and delete the post

```
export function render(event) {
  // Setup the post article element
  const article = createEl("article"); "article"// 1e.classList.add("post"
  ar"post"// 2rHTML = `
    <p><a id="${backBtn}" href="#">
    &lt; Back to Posts</a></p>
    <h1 class="entry-title">
      ${state.post.title.rendered}
    </h1>
    <div class="entry"${backBtn}"`
    <p><a id="${backBtn}" href="#">
    &lt; Back to Posts</a></p>
    <h1 class="entry-title">
      ${state.post.title.rendered}
    </h1>
    <div class="entry-content">
      ${state.post.content.rendered}
    </div>
  `// 3

  // Attach event listeners to back button }); .// 4

  // If logged in, display edit link
  if (state.loggedIn) {
    article.append(editLink(state.post));
    article.append(deleteLink(state.po// 4// 4

  // If logged in, display edit link
  if (state.loggedIn) {
    article.append(editLink(state.post));
    article.append(deleteLink(state.post));
  } // 5

  // Clear the posts from the page
```

```
  // Add the single post to the page
}
```

The last two functions of this section, `editLink()` and `deleteLink()` woork similarly so we'll describe them together.

They will create a link element and add the respective class (edit and delete) and content text.

They will add a click event handler that will call the appropriate function, `loadPost()` for `editLink()` and `deletePost()` for `deleteLink()`.

```
export function editLink(post) {
  // Setup the edit link
  const link = document.createElement("a");
  link.href = "#edit-post";
  link.classList.add("e"a"");
  link.innerText = "Edit";

  link.addEventListener("click", () => {
    setState("editorPost", post.id);
    loadPost();
  });

  "Edit"// Return the link element
  return link;
}

export function deleteLink(post) {
  // Setup the delete linkst link = document.createElement("a");
  link.href = "#delete-post";
  link.classList.add("delete-post");
  link.innerText = "Delete";

  link.addEventListener("click", event => {
    event.preventDefault();
    deletePost(post);
  });
```

```
  // Ret"a"// Return the delete link
  return link;
}
```

With the code so far we have the basic authenticaation structure and a way listing of posts and individual posts.

> We’ll explore more details in the next post.

WordPress CMS CRUD (Part 3): Some additional thoughts and ideas

In the last posts we created a CRUD system for authenticated WordPress requests but it's not complete. It currently only deals with the post content, it doesn't provide an easy way to paginate content, and it doesn't provide for how to access other parts of a WordPress site or app.

This post will discuss solutions to these problems.

Embedding content related to the post

When we load the data from the `posts` REST endpoint we get references to the post assets like featured image, author, and comments.

This is good when we're exploring the API but it becomes troublesome when we are building the content out for displaying it to the user. We would have to make additional requests for these assets and we're not guaranteed that they exist so these additional requests and delays may be for nothing.

Instead we can leverage the API and use the `_embed` query parameter to include the additional post resources when we fetch the posts. This makes the response larger but it guarantees that we will have all the data available to build post listings and individual posts.

TO make the embedded assets available to individual posts, we need to modify the `init()` function to add the `_embed: ""` parameter to the `get` Axios call

The modified code looks like this

```
export function init(event) {
  if (event) {
    event.preventDefault();
```

```
  }

  .get(state.restUrl + "wp/v2/posts/", {
    params: {
      _embed: "",
      ""// Set number of posts to get
      per_page: 10,
    }
  })
}
```

This code will embed all associates assets to each individual post on the response, making it easier to create posts the match what we get with a WordPress theme.

Paginating content for navigation

By default, the WordPress REST API will return ten items per request. We can change that using the `per_page` parameter to change the number of items that the API returns up to one hundred.

Creating the pages

But in and of itself this is not enough. Using `per_page` will always returns the same data. We need to tell WordPress what page of content we want.

The way to do it is to combine `per_page`, that will give us the number of items that we want the API to return, and `page` that provides the offset where to start the count of items to return.

For example the following request will return the second page of 5 items from the site's REST API.

```
/wp-json/wp/v2/posts/?per_page=5&page=2
```

Both parameters are optional.

Leveraging X-WP-TotalPages

So we know how to create pages of content but how do we make sure that we don't go over the last page?

WordPress's REST API provides two custom headers: `X-WP-Total` that gives you the total number of the type's items (post, pages,etc) in the collection and `X-WP-TotalPages` that gives the total number of pages available given the number of items per page we specify.

Using `X-WP-TotalPages` we can build page navigation links and we can build links to each individual page.

Accessing other elements of a WordPress site

So far we've only worked with posts and their embedded content but there are many other items that make up a WordPress application.

The following table, taken from REST API Handbook Reference shows you the default routes available throught he API.

Default endpoints available on the WordPress REST API

Resource	Base Route
Posts	`/wp/v2/posts`
Post Revisions	`/wp/v2/posts/<id>/revisions`
Categories	`/wp/v2/categories`
Tags	`/wp/v2/tags`
Pages	`/wp/v2/pages`
Page Revisions	`/wp/v2/pages/<id>/revisions`
Comments	`/wp/v2/comments`
Taxonomies	`/wp/v2/taxonomies`
Media	`/wp/v2/media`

Users	`/wp/v2/users`
Post Types	`/wp/v2/types`
Post Statuses	`/wp/v2/statuses`
Settings	`/wp/v2/settings`
Themes	`/wp/v2/themes`
Search	`/wp/v2/search`

This table does not include API routes added by plugins or generated outside the core API. Your overall API route table will certainly look different.

Creating custom post types and adding them to the REST API

One of the things I've always liked about WordPress is its flexibility. We can extend WordPress and create pretty much any type of content that we need for specific projects.

We can choose whether to build it inside a theme or a plugin. For the most part I would recommend building it in a plugin so that we will retain the capability even if we switch themes.

Like all plugins we first declare a preamble that looks like this. This is the block that makes it a plugin that we can load and use.

```
<?php
   /*
   Plugin Name: Awesomeness Creator
   Plugin URI: https://example.com
   description: a plugin to create awesomeness and spread joy
   Version: 1.0
   Author: Mr. Awesome
   Author URI: https://site.com
   License: GPL2
   */
?>
```

The actual meat of the plugin is the custom post type registration. As with all WordPress related code, it has two parts: a callbackk function and an hook that tells WordPress what to do with the callback.

```
<?php
function rivendellweb_custom_book_type() {
    $args = array(
      'publ'public'  => true,
```

```
      'show_in_rest' => true,
      'rest_base'    => 'books',
      'label'        => 'Books'show_in_rest'egister_post_type( 'book', $ar
}
add_action( 'init', 'rivendellweb_custom_book_type' );
'book'?>
```

There are two arguments that are important when working with REST API routes.

`show_in_rest` tells WordPress that we want this custom post type to show in the REST API. When I activate the plugin it adds 4 additonal books-related routes to the list of APIs available

`rest_base` allows us to rewrite the name of the route in the REST API. The default is the same name as the custom type but there may be cases where we want a different name. In our example I chose books instead of book (singular) for the name of the route.

We can also create one of more taxonomies for our custom post types and, with the `show_in_rest` attribute, we can use it from the REST API.

Because we've used a custom name in the route with `base_rest` we have to rewrite the taxonomy to match using the `rewrite` attribute in the custom taxonomy arguments.

The custom taxonomy declaration looks like this

```
<?php
function rivendellweb_custom_book_genre_taxonomy() {
  $labels => array(
    'name'          => __x('Genres'),
    'singular_name' => __x('Genre', 'taxonomy singular name'),
    'search_items'  => __('Search Genres'),
    'edit_item'     => __('Edit Genre'),
    'add_new_item'  => __('Add New Genre')
  );

  $args = array(
```

```
    'labels'       => $labels,
    'show_in_rest' => true,
    'hierarchical' => true,
    'query_var'    => true,
    'rewrite'      => array(
      'slug' => 'books/genre',
      'with_front' => false
    )
  );
  register_taxonomy( 'genre', 'book', $args );
}
add_action( 'init',
  'rivendellweb_custom_book_genre_taxonomy', 0 );
```

With this functionality plugin installed we've created a new custom post type, added an interface to it on the admin UI and made sure we can access it through the REST API.

Creating Templates for Custom Post Types

Adding attributes to a REST API endpoint

When working with APIs we should never, under any circumstance, remove elements from the endpoint. Even if you're the owner there is no real way to know who's consumming your data and how they are doing it so removing elements of the API can cause unexpected and unpleasant problems down the road.

However, there is no problem on adding attributes to your custom endpoints. This post will discuss how to do it using built-in WordPress functions and coding practices.

Before we jump into adding attributes we need to ask ourselves if we really need a new attribute or if we can use combination of existing attributes to accomplish our tasks.

For example we can get the 3 latest posts by using `per_page` and a value of 3 rather than having to create a custom field for our application.

If we decide that we need a custom field in the REST API, then the process is a two step one.

We first create the REST API field using the regiser_rest_field function.

The first parameter is the type of post that you're adding the attribute for.

The second parameter is the name of the attribute that we're adding.

The third parameter is an array of callbacks for the field. The callbacks are:

'get_callback' ***Optional***.
: The callback function used to retrieve the field value. Default is 'null', the field will not be returned in the response.

'update_callback' ***Optional***.
: The callback function used to set and update the field value. Default is 'null', the value cannot be set or updated.

'schema' ***Optional***.
: The callback function used to create the schema for this field.

In this particular case we only need to retrieve the value of the field so we only define a `get_callback`.

```
<?php
function rivendellweb_add_new_field() {
  register_rest_field(
    'post',
    'catlinks',
      array(
        'get_callback'    => 'rivendellweb_get_category_links',
        'update_callback' => null,
        'schema'          => null,
    )
  );
}

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

We then define the functions for each of the callbacks that we want to implement. In this case we only want to retrieve the value for this field, so we only need to define the `get_callback` function.

This function retrieves the categories from the current post, formats the categories and returns the results the categories to use as the value for the REST field.

```
<?php
function riivendellweb_get_category_links() {
  $categories = get_the_category();
  $separator = ', ';', 'output = '';
  if ( ! empty( $categories ) ) {
    foreach( $categories as $category ) {
      $output .= '<a href="' . esc_url( get_category_link( $category->ter
}
'">'?>
```