WordPress + Vue: Building a blog

One of the reasons why I've been so interested in learning and using the WordPress REST API is to learn how to use WordPress as a headless CMS, one where the content management is decoupled from the presentation layer.

I've also told myself that in 2020 I would start looking at Vue (and possibly React) more seriously as a way to create user interfaces and static sites.

In February 2020, Sarah Drasner wrote How To Create A Headless WordPress Site On The JAMstack for Smashing Magazine and released the code on Github and, as the lazy developer, I will use Sarah's code as the starting point for the project.

Project Goals

Before we start, we need to define what we're trying to accomplish. These items are not listed in any particular order of importance.

1. Build a site using Vue.js for the front end and WordPress for a headless CMS via the REST API
2. Create Vue components for the different parts of the site, even those you don't plan to use in the current project
3. Figure out how to incorporate Prism syntax highlighter to post content
4. Use Vuex object store to cache content
5. Expand from the basic prototype to a full working version of my publishing blog
6. Create a login system
7. Research how to build an Editor in Vue
8. What will it take to upgrade the project to Vue 3?

Getting started

The first step is to clone Sarah's repository from Github

If you use the Github CLI the command is

```
gh repo clone netlify-labs/headless-wp-nuxt
```

Otherwise use the traditional cloning process:

```
git clone https://github.com/netlify-labs/headless-wp-nuxt.git
```

Once you have cloned the repository you will see something like this. Note that I've removed `node_modules` from the list.

```
.
├── LICENSE
├── README.md
├── assets
|   └── mixins.scss
├── components
|   ├── AppIcon.vue
|   ├── AppMasthead.vue
|   └── AppNav.vue
├── jsconfig.json
├── layouts
|   └── default.vue
├── middleware
|   └── README.md
├── netlify.toml
├── nuxt.config.js
├── package-lock.json
├── package.json
├── pages
|   ├── about.vue
|   ├── blog
|   └── index.vue
├── plugins
|   ├── dateformat.js
|   ├── posts.server.js
|   └── tags.server.js
├── static
```

```
|   ├── favicon.ico
|   ├── icon.png
|   ├── lake-photo.jpg
|   └── mountains-masthead.jpg
└── store
    └── index.js
```

If you haven't already, install the required packages by running

```
npm install
```

Once the installation completes, start up the development server by running:

```
npm run dev
```

Learning to work with the store

The hardest part of this project, for me, is figuring out how to work with Vuex, the Vue data store.

The idea is that whenever we make a request, we also put the data in the store so components can request the data from the store instead of fetching it from the network every time we need it.

Vuex works with three primary concepts

- **state** holds the info we need to store
- **mutations** hold functions that update the state. Mutations are the only thing that can update state; actions cannot.
- **actions** can execute arbitrary asynchronous operations. They commit mutations

We'll use an action to make the call to WordPress API and then commit a mutation to update the state.

First, we'll check if there's any length to the posts array in state, which means we already called the API, so we don't do it again.

The code itself is fairly straight forward once you become familiar with data store concepts. The store, as initially impletemented, covers posts and tags. For illustration, we'll only work with posts.

The first thing we do is define the URL for the site in a constant. We do it to reduce typing and make the code less error prone.

The first step in setting up the store is to define the containers for the items we want to store.

In this case the post is an empty array. We can add additional state for other elements that we want to store.

```
const siteURL = "https://publishing-project.rivendellweb.net"

export const state = () => ({
  posts: [],
})
```

The mutation is also simple. When we call updatePosts we add the new posts to the posts array where we store the state for the posts.

```
export const mutations = {
  updatePosts: (state, posts) => {
    state.posts = posts
  },
}
```

It's in the `getPosts` action where the bulk of the work happens. We're using async functions so we can use await instead of the raw promises.

We're also using a try/catch

```
export const actions = {
  async getPosts({ state, commit, dispatch }) {
    if (state.posts.length) return
```

```
    try {
      let posts = await fetch(
        `${siteURL}/wp-json/wp/v2/posts?page=1&per_page=10&`
      ).then(res => res.json())

      posts = posts
        .filter(el => el.status === "publish")
        .map(({ id, slug, title, excerpt, date, tags, content }) => ({
          id,
          slug,
          title,
          excerpt,
          date,
          tags,
          content
        }))
"publish"mmit("updatePosts", posts)
    } catch (err) {
      console.log(err)
    }
  },
}
```

In the try block, we fetch the content that we want to work with, in this case the last 10 posts as they appear in the database and make sure we read them as JSON.

Once we have the array of posts, we use the filter method to only return the posts that have been published. Once we have an editor or another way to publish them from the front end, we may change this.

Then we use the map method to create a new array with only the information that we will use. Otherwise, WordPress will provide a lot of information we won't use.

We could also use the _fields parameter to the API so using map to get the data we need is no longer necessary. It is up to you.

With the data now prepared the way we want it to, we finally commit the mutation to our posts store.

When we discuss loading additional pages of posts, we'll touch on pushing items to the store in

Reviewing Vue Components

The other area of Vue.js that we need to look at before we can look at what additional components we might need.

At its most basic level, a Vue component is made of three parts:

1. A template that shows how the content of the component will appear on the page
2. A script that exports different methods and functions related to the component
3. An optional CSS stylesheet that will style the component. These styles can be scoped to the component so they won't bleed out

The example below, taken from vue-wordpress shows the basic elements of a Vue component.

```
<template>
  <main>
   <main>{ title }}</h1>
    <p>{{ message }}</p</h1>/main>
</template>

<script></template>
export default {
  data() {
    return {
      title: '404 - Page Not Found',
      message: 'Apparently nothing exists at this location.'
    }
  },
  created() {
    this.$store.dispatch('updateDocTitle', { parts: [ 'Page not found', th
  }
}
style scoped></style>
```

```
</style></style>
```

To me, the most important thing to note about Vue components are their lifecycle hooks that allow you to run code at different points in the component's lifecycle.

There are many references for lifecycle hooks, some of them are

- Single File Components
- An Overview of VueJS Dynamic Components

However you must look at the version of Vue the documentation is for. I don't expect it to change a lot, but some changes are inevitable.

This project works with version 2 of Vue and its components.

Adding functionality

This is a first attempt at adding functionality to the blog. These ideas go beyond the basics, some took me a long time to figure out.

Pages

Creating pages proved to be significantly harder than I expected. At first I thought that cloning the post template and associated code in the store and moving it to a different directory would be enough but that was not the case.

For the pages to work as intended, I had to delve into the internals of Nuxt and found the entry for `dynamicRoutes`. Since Nuxt uses Axios, each route calls the corresponding API endpoint and maps the JSON data to a URL in the application.

```
let dynamicRoutes = () => {
  const routes = axios
    .get("https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?
    .then(res => {
      return res.data.map(post => `/blog/${post.slug}`)
    })
    .get("https://publishing-project.rivendellweb.net/wp-json/wp/v2/pages?
```

```
    .then(res => {
      return res.data.map(pages => "https://publishing-project.rivendellwe
    })
  return routes
}
```

The store need to be update to also capture the page data. The first step is to add a pages array to the `state` object.

```
export const state = () => ({
  posts: [],
  tags: [],
  pages: [],
})
```

We add a method to the `mutation` object to capture the changes to the state.

```
  updatePages: (state, pages) => {
    state.pages = pages
  },
```

The `getPages action` is very similar to what we used to download and store pages but slimmer since we don't use previous and next navigation links and don't require to change the map to include next and previous. The code looks like this:

```
  async getPages({ state, commit, dispatch }) {
    if (state.pages.length) return

    try {
      let pages = await fetch(
        `${siteURL}/wp-json/wp/v2/pages?page=1&per_page=10`
      ).then(res => res.json())

      pages = pages
        .filter(el => el.status === "publish")
```

```
        .map(({ id, slug, title, excerpt, date, tags, content }) => ({
          id,
          slug,
          title,
          excerpt,
          date,
          tags,
      "publish"nt
        }))
      commit("updatePages", pages)
    } catch (err) {
      console.log(err)
    }
  },
```

This may be overkill but I added a `pages()` computed property in `index.vue`. If I'm understanding it correctly this will get the JSON for pages and put it in the store before we actually make a request for a page.

```
pages() {
  return this.$store.state.pages;
},
```

The final step, and what will actually render the page content, is to create `_slug.vue` inside `blog-pages`. It is a simplified version of the `_slug.vue` we use for posts; it has a lighter workload.

```
<template>
  <div>
    <app-masthead></app-masthead>
    <div class="page">
      <main>
          <h1>{{ page.title.rendered }}</h1>
          <small>{{ page.date | dateformat }}</small>
          <div v-html="page.content.rendered"></div>
      </main>
```

```
    </div>
  </div>
</template>
```

The component script imports the AppMasthead component. It also has a data property, two computer methods and a created method that will also retrieve the pages data from the store.

```
<script>
import AppMasthead from "@/components/AppMasthead.vue";

export default {
  data() {
    return {
      slug: this.$route.params.slug
    };
  },
  computed: {
    pages() {
      return this.$store.state.pages;
    },
    page() {
      return this.pages.find(el => el.slug === this.slug);
    }
  },
  created() {
    this.$store.dispatch("getPages");
  },
};
</script>
```

We also provide some basic styling for the pages. We will revisit this when we decide on overall styles for the blog.

```
<style>
.page {
```

```
  width: 80vw;
  margin: 0 auto;
}

h1 {
  margin-bottom: 0.5em;
}
</style>
```

Previous / next post navigation

The other type of navigation I'm interested in is the previous/next navigation inside a single post. This will help the reader navigate the content of the blog.

This is more difficult because we don't have a previous and next links in the JSON we get from the server so we'll have to figure out a way to do it.

I think that the easiest way to do this is to modify the posts endpoint on the WordPress side and add the previous and next link to the REST API post route, which we'll fetch as JSON from our Vue application

```
<?php
function rivendellweb_add_navlinks_to_post_rest( $response, $post, $reques
  global $post;
  // Get the next post.
  $next = get_adjacent_post( false, '', false );
  ''// Get the previous post. $previous = get_adjacent_post( false, '', tr
  //''// Format them and only send the data we need
  // or null, if there is no next/previous postresponse->data['next'] = (
  $response->data['previous'] = ( is_a( $previous, 'WP_Post') ) ? array( '

  return $response;
}

add_filter"title"'next't_prepare_post', 'rivendellweb_add_navlinks_to_post
```

We use the get_adjacent_post function to get the next and previous posts. Once we get them, want the the title of the post (that we'll show to the user) and the slug or

name of the post (that we'll use to build the URL to link to), that we'll use to build the URL. To get slug we use get)post_field to retrieve the previous and next values, something like: `get_post_field( 'post_name', $previous )`.

It would be tempting to just get the permalink with get_the_permalink() but that will produce a link that will point to the original server, not to the local system we're working with.

Next we need to edit the store to make sure that the `next` and `previous` fields are added to the store. This makes it easier to use elsewhere on the application.

The final step to add previous and next navigation links is to change the template of our `blog/_slug.vue` component to use v-if directive and only render the previous and next items if the item is not empty. The template now looks like this:

```
<template>
  <main class="post individual">
    <h1>{{ post.title.rendered }}</h1>
    <small class="date">{{ post.date | dateformat }}</small>
    <section v-html="post.content.rendered"></section>
    <div class="footer-nav">
      <div v-if="post.next !== null" class="nav-item next">
        <p class="next-item-title">
          <strong>Next:</strong>
        </p>
        <p><a :href="post.next.link">{{ post.next.title }}</a></p>
      </div>
      <div v-if="post.previous !== null" class="nav-item previous">
        <p class="previous-item-title">
          <strong>Previous</strong>
        </p>
        <p><a :href="post.previous.link">{{ post.previous.title }}</a></p>
      </div>
    </div>
  </main>
</template>
```

I don't like that if the next condition on the first post is false then the next post moves to the left. We might use a combination of `v-if`, `v-else` and CSS to keep the previous post from moving.

Adding third party functionality

Because we're pulling content from a CMS we can't use functionality via Vue components. The best example, in this blog is how to use Prism.js to do syntax highlighting.

Add syntax highlighting

I love Prism.js for syntqax highlighting. Luckily, I found a good tutorial on how to use Prism.js with Vue

> First we install Prism as a dependency using the primsjs Node Package.

```
npm install -D prismjs
```

In `_slug.vue` we add import statements to the component. In an ideal world we'd be able to only load the languages we want, but we can't tell in advance what those languages will be so we load all the languages we want to have available, just like if we were loading a bundle with all javascript together.

```
// Import Prism
import Prism from 'prismjs';
'prismjs'// Import the theme you want to use
import 'prismjs/themes/prism-solarizedlight.css'
// Import language plugins, this may be better bundledprismjs/components/p
import 'prismjs/components/prism-css';
import 'prismjs/components/prism-markup';
import 'prismjs/components/prism-scss'
// NOTE: 'prismjs/components/prism-css'// NOTE: php depends on markup-temp
import 'prismjs/components/prism-markup-templating'
import 'prismjs/components/prism-php';
```

Once we've imported the code, we initialize Prism by adding its `highlightAll()`

method to the mounted() lifecycle hook.

```
export default {
  // code ...
  mounted() {
    Prism.highlightAll()
  }
}
```

The last remaining issue is that, according to the documentation, “This hook is not called during server-side rendering.” but it doesn’t tell us what we can replace it with or if there is a replacement at all.

Future Work

As it stands, the project does is a good example of how to pull data from WordPress REST API and use it in a Vue application. There are a few additional features I want before I can call this an MVP. I’ll discuss them here along with ideas of how they may work.

Load additional pages of posts

Right now the blog pulls in only the latest ten posts. This is the default for a query for the posts API and it has been hardcoded this way.

But the blog has more than one page, sixty pages if we keep the default of ten posts per page, so the question is ***how do we load these additional pages of posts? where do we trigger the download from?***

One way to do so would be to add a next and previous link to `index.vue` and let the user decide how to navigate the content.

The response to the posts query includes a `link` header that looks like this:

```
<https://publishing-project.rivendellweb.net/wp-json/wp/v2/posts?page=1&pe
```

The header is accessible, the `access-control-expose-headers` includes `link` as one of the exposed headers.

The idea is to follow these steps (or something similar)

1. When we query for a page of posts, extract the link header
2. From the link header extract the previous and next URLs
3. Make the API calls to both the previous and next URLs
4. Append the returned items only if the first item in the page is not in the posts store
5. Display a next and previous button as approriate if the links exist

One of the remaining questions about these buttons is how to tell the UI what posts to display for which page.

Another way to view paginated content is to create a navigation item for the available pages.

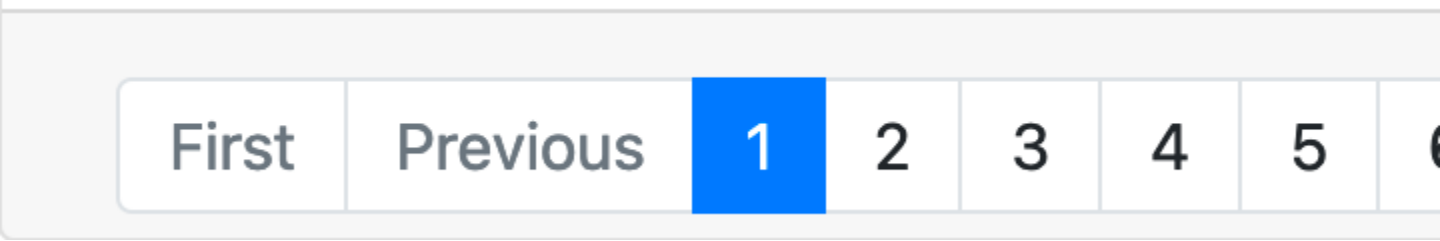

Figure 1: Example of a pagination bar using numbers, single page navigation using previous and next and jump to beginning and end using first and last

WordPress provides the `x-wp-totalpages` response header with the number of pages available based on the number of posts per page you've specified (Number of posts divided by the number of post per page, rounded up).

The idea is to do the following:

1. Query for a single page of posts to get the total number of pages by capturing the `x-wp-totalpages` header.
    1. We don't need to keep the query, we just need the header
    2. We may be able to push the data to the store when we capture our initial page of posts
2. Create the navigation using the number of pages and links to each individual page
    1. The issue becomes that, because it is not a linear navigation, it becomes harder to put the data in the store
3. Style the UI we created in the previous step

Categories

Like the original project does with tags, we can create a component that will show all categories and the associated posts to them.

We need to research in more detail how the tag component work and how to display it if there are no categories.

This is something we need to do for tags as well

We also need to work on styling so we can display them both on the same page at the same time.

One final consideration is whether tags and categories need their own component or if it's OK to keep them on the homepage.

Latest Posts

This is a subset of the posts functionality we already have. What we need is a component that will take specified number of posts and display them as URLs, similar to what we did with pages in the navigation bar.

The steps I think we need:

1. Configure the number of posts that we want to see when we see the latest posts
2. Query the store and receive the last `number` of posts, where `number` is configurable
3. Build a URL using the `title` as what the user will see and the `post_name` as the URL
    1. We cannot use the permalink because it will default to the server URL, not the application's
4. Display the URLs for the latest posts as a list
5. Provide hooks to style the component

Creating a search tool

I've wanted to experiment with Elastic Search as a WordPress search replacement but the cost is too high for something I don't know if I want to use long term.

An interesting alternative is vuex-search. It will require going into the internals

of the Nuxt application I've created but I think it's a fair compromise, at least for a development system.

One thing to remember with all these solutions is that the Vuex store is not persistent by default and we don't want it to be. So, whatever solution we implement, it has to account for dynamic data and how to search for content that might not have been downloaded yet.

Building a PWA

I love progressive web applications. They provide app-like features and help improve performance of web content. Nuxt.js, the library this project uses, has a PWA plugin. All we have to do is configure it for our application as indicated in the module setup documentation, and voila, we have a working PWA.

Because the PWA module is a collection of modules bundled together, we need to test this exhaustively to make sure none of the submodules that make up the PWA module cause any problems.

i18n

For the most part English is OK as the language for my blog but as soon as you want to use this as the basis for other projects you have to consider internationalization.

In theory, it should be possible to use Gettext-based solutions as outlined in Using Gettext based translations in Node

How to localize Vue.js app with vue-i18n and Localazy presents a more idiomatic Vue way of doing localization. It's true that it makes you dependent on a single translation tool and, if they decide to go to paid model, you will have to choose if the cost is worth it, but it appears as a fast and easy to implement solution.

Evaluate plugins for equivalent functionality

One of WordPress's advantages (or curses depending on who you ask) is the huge number of plugins available. Some of these plugins add functionality to the front end.

It is a good idea to take a look at the plugins on your site and see if there's

equivalent tools in the Vue ecosystem and how they can be added to the project.

Links and resources

- How To Create A Headless WordPress Site On The JAMstack
- Build a Vue.js SPA on Top of Headless WordPress
- vue-press
- how to use Prism.js with Vue
- Nuxt.js PWA plugin
- How to localize Vue.js app with vue-i18n and Localazy