PUSHER We're hiring

BUILD A CMS WITH LARAVEL AND VUE - PART 4: BUILDING THE DASHBOARD

Neo Ighodaro

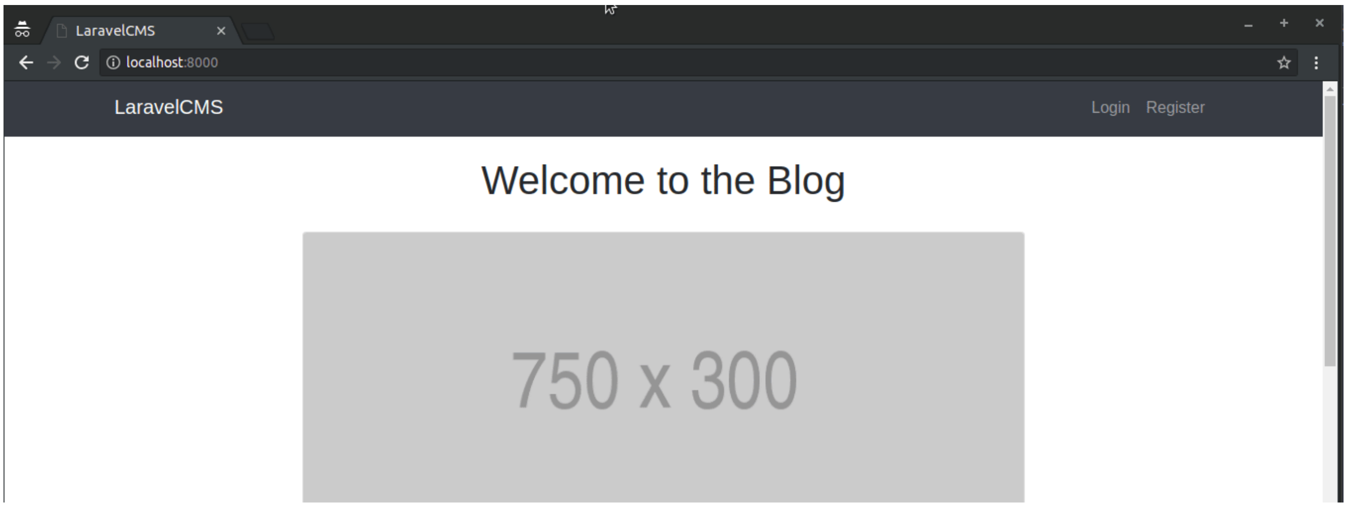
October 4th, 2018

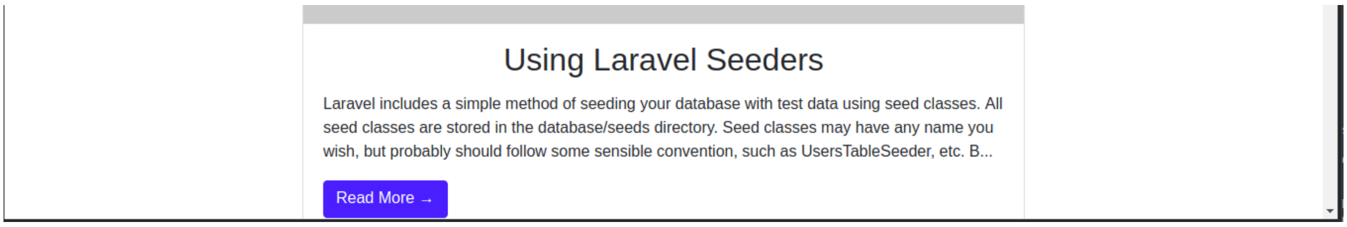
Basic knowledge of Laravel and Vue will be helpful.

In the last article of this series, we built the API interface and used Laravel API resources to return neatly formatted JSON responses. We tested that the API works as we defined it to using Postman.

In this part of the series, we will start building the admin frontend of the CMS. This is the first part of the series where we will integrate Vue and explore Vue's magical abilities.

When we are done with this part, our application will have some added functionalities as seen below:




Using Laravel Seeders

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in the database/seeds directory. Seed classes may have any name you wish, but probably should follow some sensible convention, such as UsersTableSeeder, etc. B...

The source code for this project is available here on GitHub.

Prerequisites

To follow along with this series, a few things are required:

- Basic knowledge of PHP.
- Basic knowledge of the Laravel framework.
- Basic knowledge of JavaScript (ES6 syntax).
- Basic knowledge of Vue.

Building the frontend

Laravel ships with Vue out of the box so we do not need to use the Vue-CLI or reference Vue from a CDN. This makes it possible for us to have all of our application, the frontend, and backend, in a single codebase.

Every newly created instance of a Laravel installation has some Vue files included by default, we can see these files when we navigate into the `resources/assets/js/components` folder.

Setting up Vue and VueRouter

Before we can start using Vue in our application, we need to first install some dependencies using NPM. To install the dependencies that come by default with Laravel, run the command below:

```
$ npm install
```

We will be managing all of the routes for the admin dashboard using `vue-router` so let's pull it in:

```
$ npm install --save vue-router
```

When the installation is complete, the next thing we want to do is open the `resources/assets/js/app.js` file and replace its contents with the code below:

```
// File: ./resources/assets/js/app.js
require('./bootstrap');

import Vue from 'vue'
import VueRouter from 'vue-router'
import Homepage from './components/Homepage'
import Read from './components/Read'

Vue.use(VueRouter)

const router = new VueRouter({
    mode: 'history',
    routes: [
        {
            path: '/admin/dashboard',
            name: 'read',
            component: Read,
            props: true
        },
    ],
});

const app = new Vue({
    el: '#app',
    router,
    components: { Homepage },
});
```

In the snippet above, we imported the `VueRouter` and added it to the Vue application. We also imported a `Homepage` and a `Read` component. These are the components where we will write our markup so let's create both files.

Open the `resources/assets/js/components` folder and create four files:

1. `Homepage.vue` - this will be the parent component for the admin dashboard frontend.
2. `Read.vue` - this will be component that displays all the available posts on the admin dashboard.
3. `Create.vue` - this will be the component where an admin user can create a new post.

4. `Update.vue` - this will be the component that displays the view where an admin user can update an existing post.

> Note that we didn't create a component file for the delete operation, this is because it is going to be possible to delete a post from the `Read` component. There is no need for a view.

In the `resources/assets/js/app.js` file, we defined a `routes` array and in it, we registered a `read` route. During render time, this route's path will be mapped to the `Read` component.

In the previous article, we specified that admin users should be shown an `admin.dashboard` view in the `index` method, however, we didn't create this view. Let's create the view. Open the `resources/views` folder and create a new folder called `admin`. Within the new `resources/views/admin` folder, create a new file and called `dashboard.blade.php`. This is going to be the entry point to the admin dashboard, further from this route, we will let the `VueRouter` handle everything else.

Open the `resources/views/admin/dashboard.blade.php` file and paste in the following code:

```
<!-- File: ./resources/views/admin/dashboard.blade.php -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title> Welcome to the Admin dashboard </title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/boots
    <style>
        html, body {
        background-color: #202B33;
        color: #738491;
        font-family: "Open Sans";
        font-size: 16px;
        font-smoothing: antialiased;
        overflow: hidden;
        }
    </style>
</head>
```

```
<body>

  <script src="{{ asset('js/app.js') }}"></script>
</body>
</html>
```

Our goal here is to integrate Vue into the application, so we included the `resources/assets/js/app.js` file with this line of code:

```
<script src="{{ asset('js/app.js') }}"></script>
```

For our app to work, we need a root element to bind our Vue instance unto. Before the `<script>` tag, add this snippet of code:

```
<div id="app">
  <Homepage
    :user-name='@json(auth()->user()->name)'
    :user-id='@json(auth()->user()->id)'
  ></Homepage>
</div>
```

We earlier defined the `Homepage` component as the wrapping component, that's why we pulled it in here as the root component. For some of the frontend components to work correctly, we require some details of the logged in admin user to perform CRUD operations. This is why we passed down the `userName` and `userId` props to the `Homepage` component.

We need to prevent the `CSRF` error from occurring in our Vue frontend, so include this snippet of code just before the `<title>` tag:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
<script> window.Laravel = { csrfToken: 'csrf_token() ' } </script>
```

This snippet will ensure that the correct token is always included in our frontend, Laravel provides the `CSRF` protection for us out of the box.

At this point, this should be the contents of your `resources/views/admin/dashboard.blade.php` file:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <meta name="csrf-token" content="{{ csrf_token() }}">
    <script> window.Laravel = { csrfToken: 'csrf_token() ' } </script>
    <title> Welcome to the Admin dashboard </title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootst
    <style>
      html, body {
        background-color: #202B33;
        color: #738491;
        font-family: "Open Sans";
        font-size: 16px;
        font-smoothing: antialiased;
        overflow: hidden;
      }
    </style>
</head>
<body>
<div id="app">
  <Homepage
    :user-name='@json(auth()->user()->name)'
    :user-id='@json(auth()->user()->id)'>
  </Homepage>
</div>
<script src="{{ asset('js/app.js') }}"></script>
</body>
</html>
```

Setting up the Homepage view

Open the `Homepage.vue` file that we created some time ago and include this markup template:

```
<!-- File: ./resources/app/js/components/Homepage.vue -->
<template>
  <div>
    <nav>
      <section>
        <a style="color: white" href="/admin/dashboard">Laravel-CMS</a> &r
        <a style="color: white" href="/">HOME</a>
        <hr>
        <ul>
```

```
            <li>
              <router-link :to="{ name: 'create', params: { userId } }">
                NEW POST
              </router-link>
            </li>
          </ul>
        </section>
      </nav>
      <article>
        <header>
          <header class="d-inline">Welcome, {{ userName }}</header>
          <p @click="logout" class="float-right mr-3" style="cursor: pointe
        </header>
        <div>
          <router-view></router-view>
        </div>
      </article>
    </div>
  </template>
```

We added a `router-link` in this template, which routes to the `Create` component.

We are passing the `userId` data to the `create` component because a `userId` is required during `Post` creation.

Let’s include some styles so that the page looks good. Below the closing `template` tag, paste the following code:

```
  <style scoped>
    @import url(https://fonts.googleapis.com/css?family=Dosis:300|Lato:300,4
    @import url("https://netdna.bootstrapcdn.com/font-awesome/4.2.0/css/font
    * {
      -moz-box-sizing: border-box;
      -webkit-box-sizing: border-box;
      box-sizing: border-box;
    }
    header {
      color: #d3d3d3;
    }
    nav {
      position: absolute;
      top: 0;
      bottom: 0;
      right: 82%;
      left: 0;
      padding: 22px;
```

```
        border-right: 2px solid #161e23;
      }
      nav > header {
        font-weight: 700;
        font-size: 0.8rem;
        text-transform: uppercase;
      }
      nav section {
        font-weight: 600;
      }
      nav section header {
        padding-top: 30px;
      }
      nav section ul {
        list-style: none;
        padding: 0px;
      }
      nav section ul a {
        color: white;
        text-decoration: none;
        font-weight: bold;
      }
      article {
        position: absolute;
        top: 0;
        bottom: 0;
        right: 0;
        left: 18%;
        overflow: auto;
        border-left: 2px solid #2a3843;
        padding: 20px;
      }
      article > header {
        height: 60px;
        border-bottom: 1px solid #2a3843;
      }
    </style>
```

We are using the scoped attribute on the `<style>` tag because we want the CSS to only be applied on the `Homepage` component.

Next, let's add the `<script>` section that will use the props we passed down from the parent component. We will also define the method that controls the `log out` feature here. Below the closing `style` tag, paste the following code:

```
<script>
export default {
  props: {
    userId: {
      type: Number,
      required: true
    },
    userName: {
      type: String,
      required: true
    }
  },
  data() {
    return {};
  },
  methods: {
    logout() {
      axios.post("/logout").then(() => {
        window.location = "/";
      });
    }
  }
};
</script>
```

Setting up the Read view

In the `resources/assets/js/app.js` file, we defined the path of the `read` component as `/admin/dashboard`, which is the same address as the `Homepage` component. This will make sure the `Read` component always loads by default.

In the `Read` component, we want to load all of the available posts. We are also going to add **Update** and **Delete** options to each post. Clicking on these options will lead to the `update` and `delete` views respectively.

Open the `Read.vue` file and paste the following:

```
<!-- File: ./resources/app/js/components/Read.vue -->
<template>
    <div id="posts">
        <p class="border p-3" v-for="post in posts">
            {{ post.title }}
            <router-link :to="{ name: 'update', params: { postId : post.i
                <button type="button" class="p-1 mx-3 float-right btn btn-
```

```
                    Update
                </button>
            </router-link>
            <button
                type="button"
                @click="deletePost(post.id)"
                class="p-1 mx-3 float-right btn btn-danger"
            >
                Delete
            </button>
        </p>
        <div>
            <button
                v-if="next"
                type="button"
                @click="navigate(next)"
                class="m-3 btn btn-primary"
            >
              Next
            </button>
            <button
                v-if="prev"
                type="button"
                @click="navigate(prev)"
                class="m-3 btn btn-primary"
            >
              Previous
            </button>
        </div>
    </div>
</template>
```

Above, we have the template to handle the posts that are loaded from the API. Next, paste the following below the closing `template` tag:

```
<script>
export default {
  mounted() {
    this.getPosts();
  },
  data() {
    return {
      posts: {},
      next: null,
      prev: null
    };
  },
  methods: {
    getPosts(address) {
      axios.get(address ? address : "/api/posts").then(response => {
        this.posts = response.data.data;
        this.prev = response.data.links.prev;
        this.next = response.data.links.next;
      });
    },
    deletePost(id) {
      axios.delete("/api/posts/" + id).then(response => this.getPosts())
    },
    navigate(address) {
      this.getPosts(address)
    }
  }
};
</script>
```

In the script above, we defined a `getPosts()` method that requests a list of posts from the backend server. We also defined a `posts` object as a data property. This object will be populated whenever posts are received from the backend server.

We defined `next` and `prev` data string properties to store pagination links and only display the pagination options where it is available.

Lastly, we defined a `deletePost()` method that takes the `id` of a post as a parameter and sends a `DELETE` request to the API interface using Axios.

Testing the application

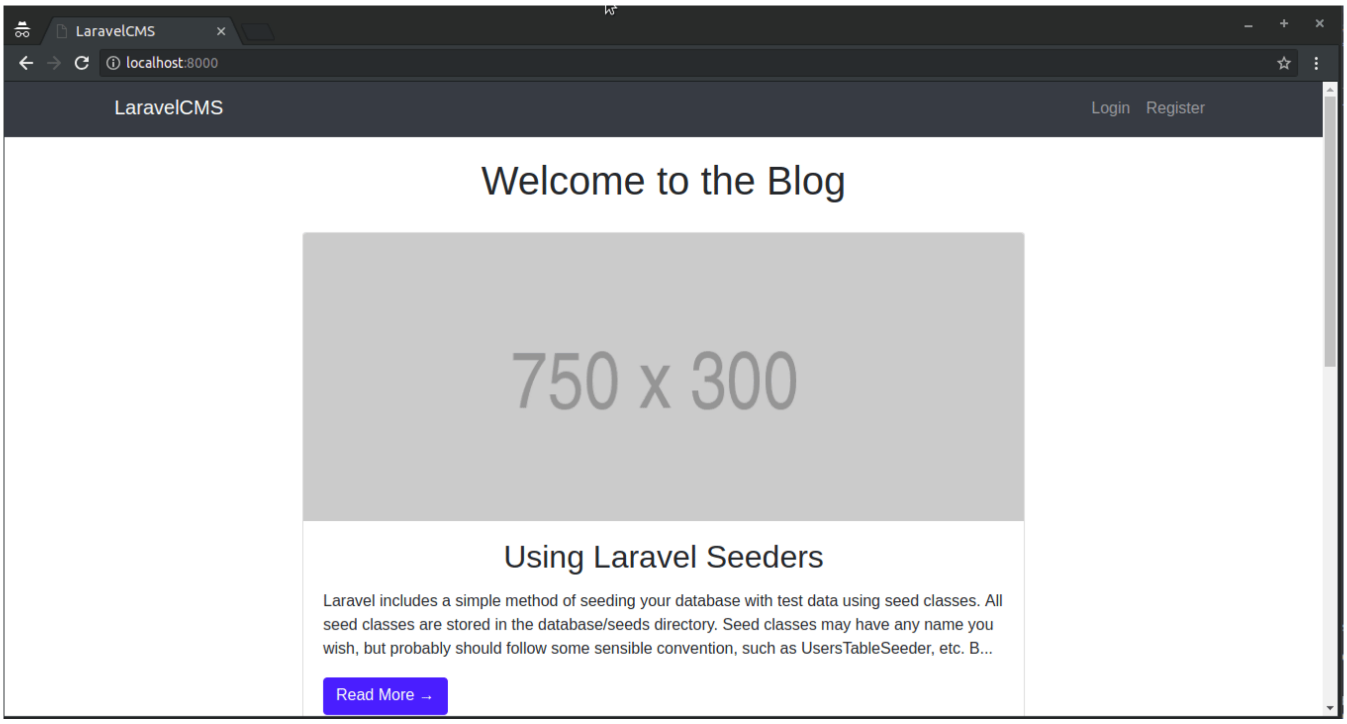
Now that we have completed the first few components, we can serve the application using this command:

```
$ php artisan serve
```

We will also build the assets so that our JavaScript is compiled for us. To do this, will run the command below in the root of the project folder:

```
$ npm run dev
```

We can visit the application's URL http://localhost:8000 and log in as an admin user, and delete a post:



Welcome to the Blog

Using Laravel Seeders

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in the database/seeds directory. Seed classes may have any name you wish, but probably should follow some sensible convention, such as UsersTableSeeder, etc. B...

Conclusion

In this part of the series, we started building the admin dashboard using Vue. We installed `VueRouter` to make the admin dashboard a SPA. We added the homepage view of the admin dashboard and included read and delete functionalities.

We are not done with the dashboard just yet. In the next part, we will add the views that lets us create and update posts.

The source code for this project is available here on Github.

JAVASCRIPT LARAVEL PHP VUE.JS

NO PUSHER TECH

Products Developers