PUSHER We're hiring

BUILD A CMS WITH LARAVEL AND VUE - PART 5: COMPLETING OUR DASHBOARDS

Neo Ighodaro

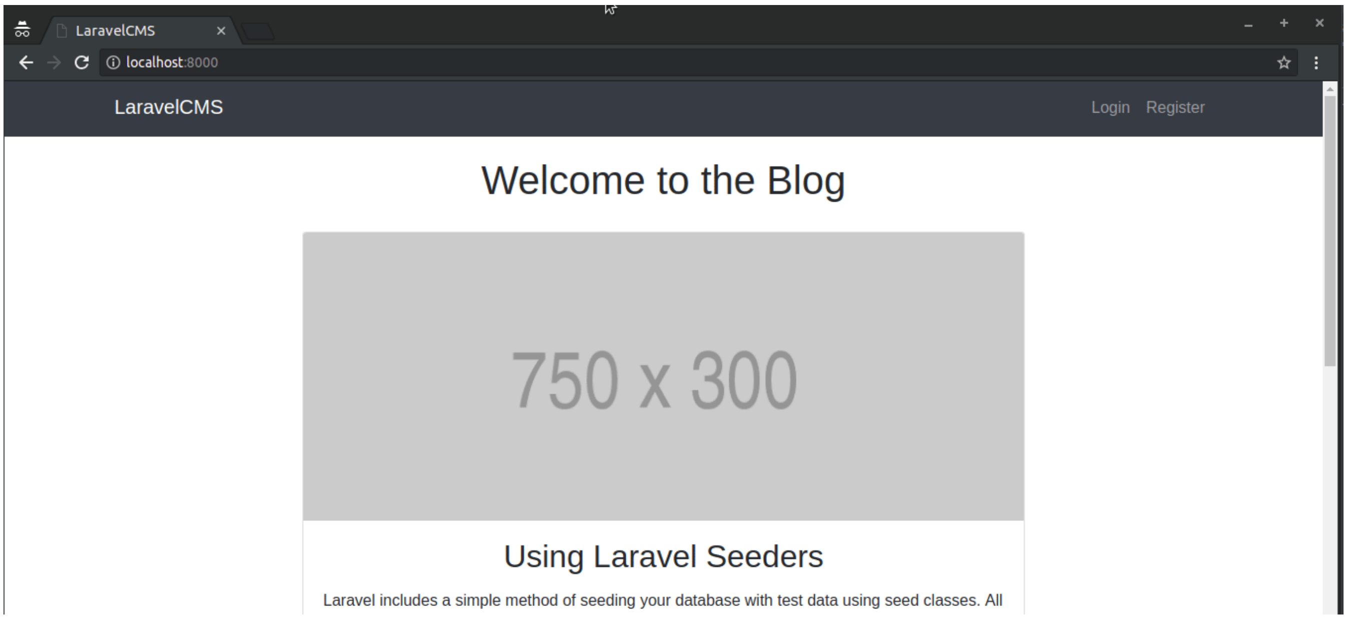
October 9th, 2018

Basic knowledge of Laravel and Vue will be helpful.

In the previous part of this series, we built the first parts of the admin dashboard using Vue. We also made it into an SPA with the `VueRouter`, this means that visiting the pages does not cause a reload to the web browser.

We only built the wrapper component and the `Read` component that retrieves the posts to be loaded so an admin can manage them.

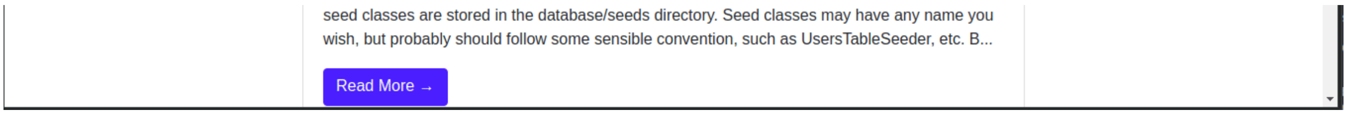
Here's a recording of what we ended up with, in the last article:



Welcome to the Blog

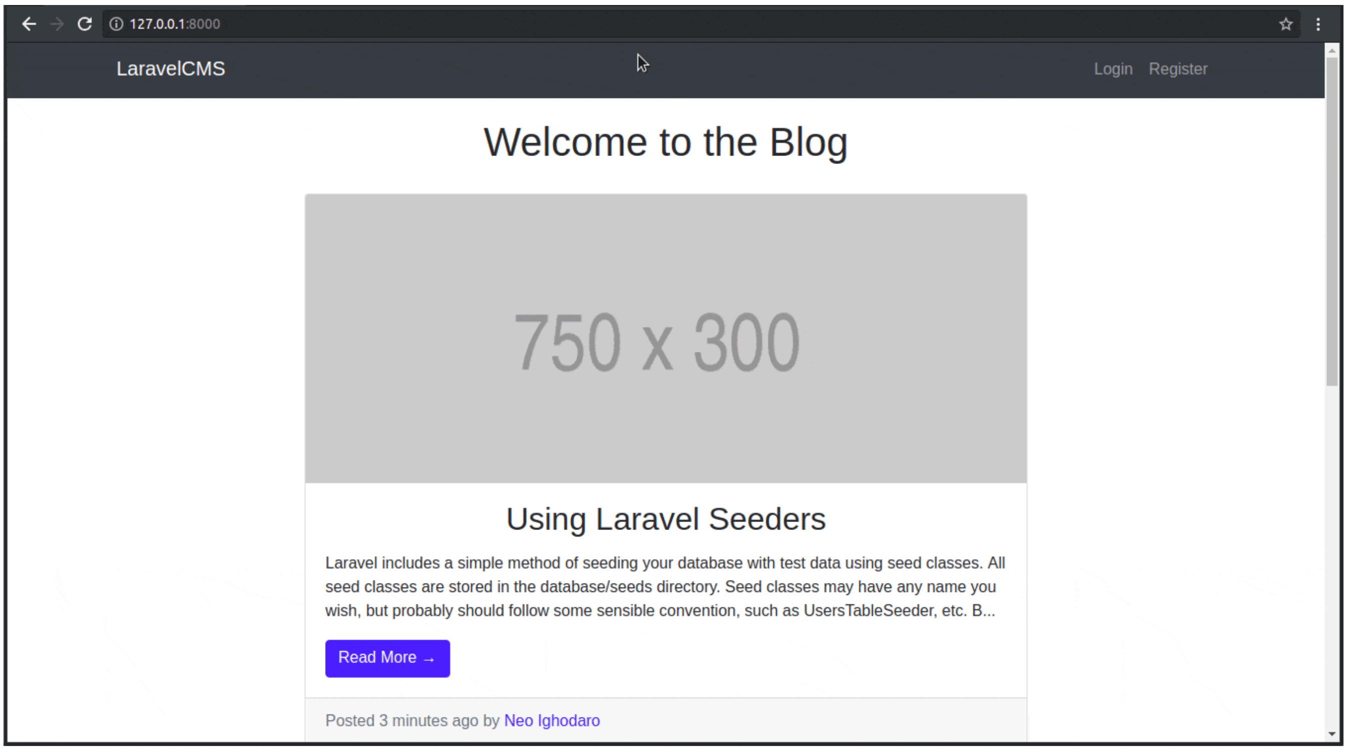
Using Laravel Seeders

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...

In this article, we will build the view that will allow users to create and update posts. We will start writing code in the `Update.vue` and `Create.vue` files that we created in the previous article.

When we are done with this part, we will have additional functionalities like create and updating:



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...

Posted 3 minutes ago by Neo Ighodaro

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.

Including the new routes in VueRouter

In the previous article, we only defined the route for the `Read` component, we need

In the previous article, we only defined the route for the `Read` component, we need to include the route configuration for the new components that we are about to build; `Update` and `Create`.

Open the `resources/assets/js/app.js` file and replace the contents with the code below:

```
require('./bootstrap');

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

Vue.use(VueRouter)

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

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

Above, we have added two new components to the JavaScript file. We have the `Create` and `Read` components. We also added them to the `router` so that they can be loaded using the specified URLs.

Building the create view

Open the `Create.vue` file and update it with this markup template:

```
<!-- File: ./resources/app/js/components/Create.vue -->
<template>
  <div class="container">
    <form>
      <div :class="['form-group m-1 p-3', (successful ? 'alert-success' :
        <span v-if="successful" class="label label-sucess">Published!</spa
      </div>
      <div :class="['form-group m-1 p-3', error ? 'alert-danger' : '']">
        <span v-if="errors.title" class="label label-danger">
          {{ errors.title[0] }}
        </span>
        <span v-if="errors.body" class="label label-danger">
          {{ errors.body[0] }}
        </span>
        <span v-if="errors.image" class="label label-danger">
          {{ errors.image[0] }}
        </span>
      </div>

      <div class="form-group">
        <input type="title" ref="title" class="form-control" id="title" pl
      </div>

      <div class="form-group">
        <textarea class="form-control" ref="body" id="body" placeholder="E
      </div>

      <div class="custom-file mb-3">
        <input type="file" ref="image" name="image" class="custom-file-inp
        <label class="custom-file-label" >Choose file...</label>
      </div>

      <button type="submit" @click.prevent="create" class="btn btn-primary
        Submit
      </button>
    </form>
  </div>
</template>
```

Above we have the template for the `Create` component. If there is an error during post creation, there will be a field indicating the specific error. When a post is successfully published, there will also a message saying it was successful.

Let's include the `script` logic that will perform the sending of posts to our backend server and read back the response.

After the closing `template` tag add this:

```
    <script>
    export default {
      props: {
        userId: {
          type: Number,
          required: true
        }
      },
      data() {
        return {
          error: false,
          successful: false,
          errors: []
        };
      },
      methods: {
        create() {
          const formData = new FormData();
          formData.append("title", this.$refs.title.value);
          formData.append("body", this.$refs.body.value);
          formData.append("user_id", this.userId);
          formData.append("image", this.$refs.image.files[0]);

          axios
            .post("/api/posts", formData)
            .then(response => {
              this.successful = true;
              this.error = false;
              this.errors = [];
            })
            .catch(error => {
              if (!_.isEmpty(error.response)) {
                if ((error.response.status = 422)) {
                  this.errors = error.response.data.errors;
                  this.successful = false;
                  this.error = true;
                }
              }
            });

          this.$refs.title.value = "";
          this.$refs.body.value = "";
        }
      }
    };
    </script>
```

In the script above, we defined a `create()` method that takes the values of the `input` fields and uses the Axios library to send them to the API interface on the backend server. Within this method, we also update the status of the operation, so that an admin user can know when a post is created successfully or not.

Building the update view

Let's start building the `Update` component. Open the `Update.vue` file and update it with this markup template:

```
<!-- File: ./resources/app/js/components/Update.vue -->
<template>
  <div class="container">
    <form>
      <div :class="['form-group m-1 p-3', successful ? 'alert-success' : '
        <span v-if="successful" class="label label-sucess">Updated!</span
      </div>

      <div :class="['form-group m-1 p-3', error ? 'alert-danger' : '']">
        <span v-if="errors.title" class="label label-danger">
          {{ errors.title[0] }}
        </span>
        <span v-if="errors.body" class="label label-danger">
          {{ errors.body[0] }}
        </span>
      </div>

      <div class="form-group">
        <input type="title" ref="title" class="form-control" id="title" pl
      </div>

      <div class="form-group">
        <textarea class="form-control" ref="body" id="body" placeholder="E
      </div>
```

```
        <button type="submit" @click.prevent="update" class="btn btn-primary
          Submit
        </button>
      </form>
    </div>
  </template>
```

This template is similar to the one in the `Create` component. Let’s add the `script` for the component.

Below the closing `template` tag, paste the following:

```
  <script>
  export default {
    mounted() {
      this.getPost();
    },
    props: {
      postId: {
        type: Number,
        required: true
      }
    },
    data() {
      return {
        error: false,
        successful: false,
        errors: []
      };
    },
    methods: {
      update() {
        let title = this.$refs.title.value;
        let body = this.$refs.body.value;

        axios
          .put("/api/posts/" + this.postId, { title, body })
          .then(response => {
            this.successful = true;
            this.error = false;
            this.errors = [];
          })
          .catch(error => {
            if (!_.isEmpty(error.response)) {
              if ((error.response.status = 422)) {
                this.errors = error.response.data.errors;
              this.successful = false;
              this.error = true;
            }
          }
        });
      },
      getPost() {
        axios.get("/api/posts/" + this.postId).then(response => {
          this.$refs.title.value = response.data.data.title;
          this.$refs.body.value = response.data.data.body;
        });
      }
    }
  };
  </script>
```

In the script above, we make a call to the `getPosts()` method as soon as the component is `mounted`. The `getPosts()` method fetches the data of a single post from the backend server, using the `postId`.

When Axios sends back the data for the post, we update the input fields in this component so they can be updated.

Finally, the `update()` method takes the values of the fields in the components and attempts to send them to the backend server for an update. In a situation where the fails, we get instant feedback.

Testing the application

To test that our changes work, we want to refresh the database and restore it back to a fresh state. To do this, run the following command in your terminal:

```
$ php artisan migrate:fresh --seed
```

Next, let's compile our JavaScript files and assets. This will make sure all the changes we made in the Vue component and the `app.js` file gets built. To recompile, run the command below in your terminal:

```
$ npm run dev
```

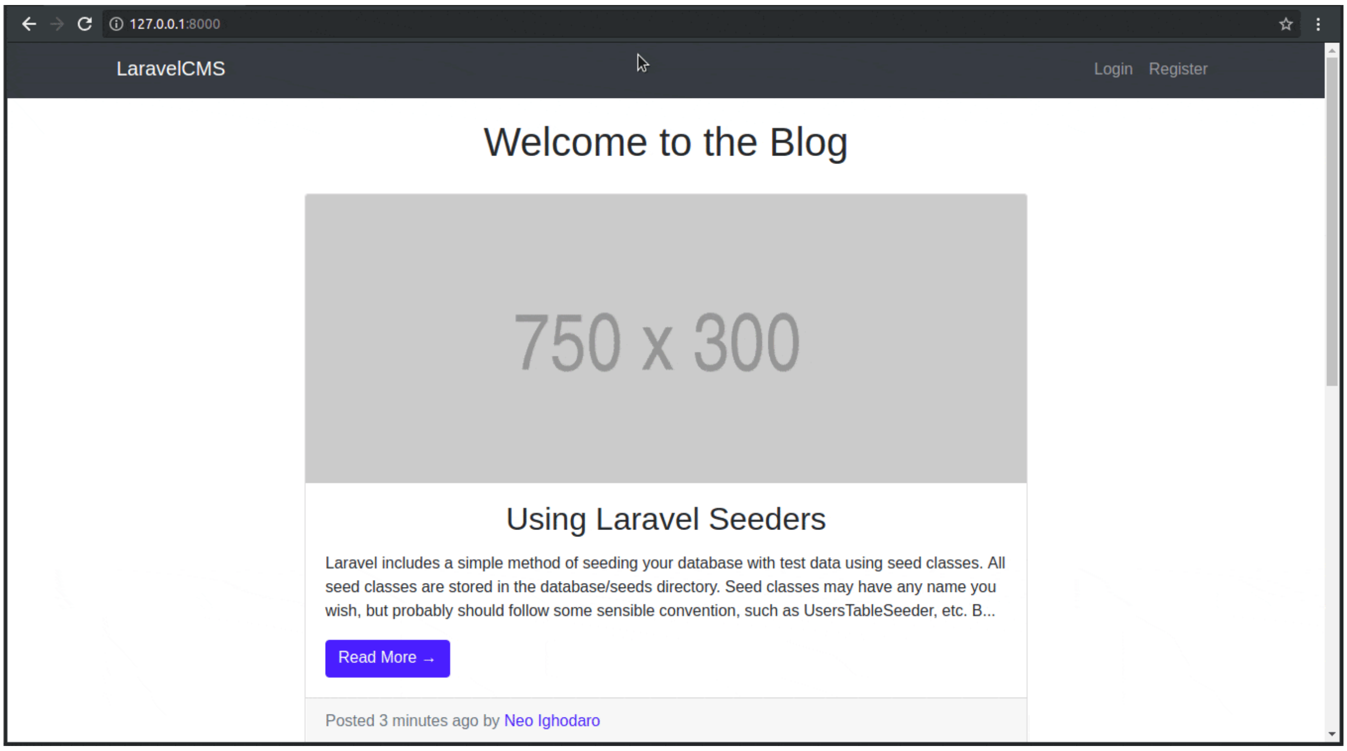
Lastly, we need to serve the application. To do this, run the following command in

Lastly, we need to serve the application. To do this, run the following command in your terminal window:

```
$ php artisan serve
```

If you had the serve command running before, then you might need to restart it.

We will visit the application's http://localhost:8000 and log in as an admin user. From the dashboard, you can test the create and update feature:



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...

Posted 3 minutes ago by Neo Ighodaro

Conclusion

In this part of the series, we updated the dashboard to include the `Create` and `Update` component so the administrator can add and update posts.

In the next article, we will build the views that allow for the creation and updating of a post.

The source code for this project is available [here]() on Github.

JAVASCRIPT LARAVEL PHP VUE.JS

NO PUSHER TECH

Products

Channels

Chatkit

Beams

Developers

Docs

Tutorials

Status

Support

Sessions

Company

Contact Sales

Terms & Conditions

Security

Careers

Blog

Connect

Twitter

Medium

YouTube

LinkedIn

GitHub