LOGIN SIGN UP

Build a Guestbook with Laravel and Vue.js

Rachid Laasri (@rashidlaasri)

October 31, 2017

#vue #laravel #API

C O D E

Introduction

Welcome back again Scotch.io readers, in today's lesson we will be building a Guestbook that looks exactly like this:

It looks cool, doesn't it?

Table of Contents

You will not only build the Guestbook but you will learn few things about both Laravel and Vue.js:

- New Laravel 5.5 Model Factory structure.
- Testing API Endpoints with Postman and exporting them to be used by your teammates.
- New Laravel 5.5 presets.
- New Laravel 5.5 Transformers.
- Creating and working with Vue.js components.
- Making Ajax calls with Laravel and Axios.

Installing Laravel

Installing Laravel is as simple as running a command in your terminal, cd into your**www** and execute:

```
composer create-project --prefer-dist laravel/laravel guestbook
```

After that, you will need to create a configuration file and point your domain name to the public folder (for me it is gonna be http://guestbook.dev) and make sure the **storage** and the **bootstrap/cache** directories are writable by your web server or Laravel will not run.

> Note: Usually downloading Laravel via composer will set the application key for you, but if for whatever reason it didn't work for you and you are getting either "No application encryption key has been specified." or "The only supported ciphers are AES-128-CBC and AES-256-CBC with the correct key lengths." running this command will fixed it for you:

```
php artisan key:generate
```

If you have done everything correctly then by browsing tohttp://guestbook.dev should see this exact same page:

Database Configuration

Laravel's database configurations are stored inside the environment variables file, to set your database information copy the content of **.env.example** to a **.env** file:


```
cp .env.example .env
```

This is the part we are interested in:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

For me, I will be using **SQLite**, that means I'll set the **DB_CONNECTION** to sqlite and remove the rest of the configs.

if you removed **DB_DATABASE** config key, Laravel will assume you are working with database located in database/database.sqlite. Make sure you create that by running:

```
touch database/database.sqlite
```

Models and Migrations

For the Guestbook we will only need one model and migration, Let's name it **Signature**.

To create both these files we can run:

```
php artisan make:model Signature -m
```

When passing the `-m` flag to the `php artisan make:model` command a migration will be generated for you. This little trick will save you a lot of time and keystrokes.

Here is what our migration will look like:

```
class CreateSignaturesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('signatures', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email');
            $table->text('body');
            $table->timestamp('flagged_at')->nullable();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('signatures');
    }
}
```

The column names are self explanatory but if you are confused about what**flagged_at** does, it's basically a timestamp column that will hold the date and time for when the signature got flagged/reported and it can be null for when the post hasn't been flagged, pretty much the same way **created_at**, **updated_at** and **deleted_at** works.

Update your migration file, hit save and run the migration command:

```
php artisan migrate
```

Last thing on this chapter is adding our columns to the fillable array on the model itself to allow for mass-assignment.

```
/**
* Field to be mass-assigned.
*
* @var array
*/
protected $fillable = ['name', 'email', 'body', 'flagged_at'];
```

If you don't know what that means or what does it do, here's a good explanation from the docs:

> A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an is_admin parameter through an HTTP request, which is then passed into your model's create method, allowing the user to escalate themselves to an administrator.

Model Factories

Next step would be generating dummy data to work with and we will be usingLaravel Model Factories for that, Luckily for us Laravel 5.5 comes with a much cleaner way of storing these factories by putting each factory on its own file and giving us the ability to generate them using the command line.

So let's go ahead and run:

```
php artisan make:factory SignatureFactory
```

And use Faker to get fake data that matches our table structure:

```
$factory->define(App\Signature::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->safeEmail,
        'body' => $faker->sentence
    ];
});
```

Our Signature model factory is ready, it's time to generate some dummy data.

In your command line run: `php artisan tinker` Then:

```
factory(App\Signature::class, 100)->create();
```

You can create as many records as you want by replacing 100 with a number of your choice.

Routes and Controllers

Defining Our Routes

We can define these three routes by registering a new resource and excluding the ones we won't be using:

- GET: **api/signatures** this endpoint is responsible for fetching all signatures.
- GET: **api/signature/:id** this endpoint is responsible for fetching a single signature by its ID.
- POST: **api/signatures** this is the endpoint we will be hitting to save a new signature.

routes/api.php:

```
Route::resource('signatures', 'Api\SignatureController')
    ->only(['index', 'store', 'show']);
```

- PUT: **api/:id/report** this endpoint is the one we will use to report a signature.

routes/api.php

```
Route::put('signatures/{signature}/report', 'Api\ReportSignature@update');
```

Creating The Controllers

As you can already see in the routes definition section, the controllers we will need are**SignatureController** and **ReportSignature**.

- Generating and writing SignatureController

```
php artisan make:controller Api/SignatureController
```

And this is what it will contain:

```
<?php

namespace App\Http\Controllers\Api;

use App\Signature;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Http\Resources\SignatureResource;

class SignatureController extends Controller
{
    /**
     * Return a paginated list of signatures.
```

```
     *
     * @return SignatureResource
     */
    public function index()
    {
        $signatures = Signature::latest()
            ->ignoreFlagged()
            ->paginate(20);

        return SignatureResource::collection($signatures);
    }

    /**
     * Fetch and return the signature.
     *
     * @param Signature $signature
     * @return SignatureResource
     */
    public function show(Signature $signature)
    {
        return new SignatureResource($signature);
    }

    /**
     * Validate and save a new signature to the database.
     *
     * @param Request $request
     * @return SignatureResource
     */
    public function store(Request $request)
    {
        $signature = $this->validate($request, [
            'name' => 'required|min:3|max:50',
            'email' => 'required|email',
            'body' => 'required|min:3'
        ]);

        $signature = Signature::create($signature);

        return new SignatureResource($signature);
    }
}
```

As you can see in our **index** method, we are using a scope with the name of**ignoreFlagged** to only return the

signatures that hasn't been flagged. You can define it by adding these lines to your **Signature** Model:

```
/**
 * Ignore flagged signatures.
 *
 * @param $query
 * @return mixed
 */
public function scopeIgnoreFlagged($query)
{
    return $query->where('flagged_at', null);
}
```

- Generating and writing ReportSignature

```
php artisan make:controller Api/ReportSignature
```

And this is what it will contain:

```php
<?php

namespace App\Http\Controllers\Api;

use App\Signature;
use App\Http\Controllers\Controller;

class ReportSignature extends Controller
{
    /**
     * Flag the given signature.
     *
     * @param Signature $signature
     * @return Signature
     */
    public function update(Signature $signature)
    {
        $signature->flag();

        return $signature;
    }
}
```

When we retrieve the signature using Laravel Model Binding feature we call a flag method on it which simply sets the **flagged_at** column value to the current datetime, same way Laravel Soft Delete works. You can add this functionality by defining this method in your **Signature** Model:

```php
/**
 * Flag the given signature.
 *
 * @return bool
 */
public function flag()
{
    return $this->update(['flagged_at' => \Carbon\Carbon::now()]);
}
```

Creating Transformers

Laravel 5.5 ships with a really cool feature, if you are familiar with creating APIs then you clearly know that you always need to transform your data and not actually exposing your database table structure to your clients because if you'll break anything that is relying on your API if you changed your table structure and for security purposes too. Nothing good comes from exposing your database structure.

In our case we will only need a Signature transformer, to create it we can run this command:

```
php artisan make:resource SignatureResource
```

And this will be its content:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\Resource;

class SignatureResource extends Resource
{
    /**
     * Transform the resource into an array.
     *
     * @param  \Illuminate\Http\Request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'avatar' => $this->avatar,
            'body' => $this->body,
            'date' => $this->created_at->diffForHumans()
        ];
    }
}
```

Because we are calling an **avatar** property we don't have, we need to write an accessor for it. Having this in our **Signature** Model is also perfect because we don't want to expose our guests email addresses.

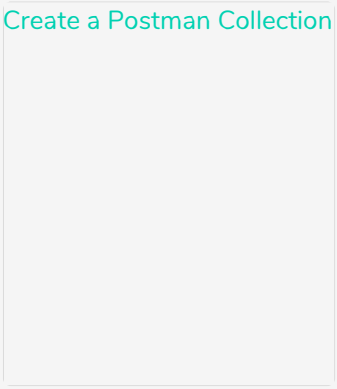
```
/**
 * Get the user Gravatar by their email address.
 *
 * @return string
 */
public function getAvatarAttribute()
{
    return sprintf('https://www.gravatar.com/avatar/%s?s=100', md5($this->email));
}
```

Testing Endpoints With Postman

After creating our endpoints, controllers and transformers, it's time to test it! Let's make sure everything is working as we'd expect. You might be asking me "But, Rachid? Why are we using Postman? We can simply use our browser for this?" And yes! I agree with you; we can use our browser to test this API, but if you are not actually writing your tests (which you should do BTW!) then I recommend you to test it with Postman because at least you can save those tests and run them again instead of opening the browser every time you make a change.

To install Postman, browse to their website Postman | Supercharge your API workflow and pick the one that will work for you based on your OS.

I created a collection named **Scotch Guestbook** and put all my tests in it, you can do the same by clicking the **New** button and selecting the **Collection** option



Give it a name, a description and hit **Create**:

Create a Postman Collection

After creating your test, click **Save** and give it a name, a description, select a collection and hit **Save**:

Save a Postman test to a Collection

Testing Our API Endpoints

- List of all signatures:

- Finding a signature by its ID:

- Creating a new signature:

- Reporting a signature:

I will include this collection with the project code source in GitHub.

The Frontend Setup

By now, we are pretty much done with backend, the only remaining thing is to link our back with the frontend.

Homepage Setup

GET: **/** This is our GuestBook entry point, it is responsible for rendering the home page.

routes/web.php:

```
Route::get('/', 'SignaturesController@index')->name('home');
```

Then we create our controller by running:

```
php artisan make:controller SignaturesController
```

And this will be its content:

```
<?php

namespace App\Http\Controllers;

class SignaturesController extends Controller
{
    /**
     * Display the GuestBook homepage.
     *
     * @return \Illuminate\Contracts\View\Factory|\Illuminate\View\View
     */
    public function index()
    {
        return view('signatures.index');
    }
}
```

After that, we need to create our **signatures.index** view. Inside **/resources/views/** create a **master.blade.php** file which will hold the website layout and allow for the other pages to extend it:

```
<!doctype html>
<html lang="{{ app()->getLocale() }}">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Scotch.io GuestBook</title>
    <meta name="csrf-token" content="{{ csrf_token() }}">
    <link href="{{ mix('css/app.css') }}" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="app">
        <nav class="navbar navbar-findcond">
            <div class="container">
                <div class="navbar-header">
                    <a class="navbar-brand" href="{{ route('home') }}">GuestBook</a>
                </div>
                <div class="collapse navbar-collapse" id="navbar">
                    <ul class="nav navbar-nav navbar-right">
                        <li class="active">
                            <a href="{{ route('sign') }}">Sign the GuestBook</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
        @yield('content')
    </div>
    <script src="{{ mix('js/app.js') }}"></script>
</body>
</html>
```

Then, we create a new view inside a **signatures** folder with the name of **index.blade.php**

```
@extends('master')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <signatures></signatures>
            </div>
        </div>
    </div>
@endsection
```

Signature Creation Page Setup

- GET: **/sign** This page is responsible for displaying the form for creating a new signature.

```
Route::get('sign', 'SignaturesController@create')->name('sign');
```

We have already created that controller, so let's add this method to it:

```
/**
 * Display the GuestBook form page.
 *
 * @return \Illuminate\Contracts\View\Factory|\Illuminate\View\View
 */
public function create()
{
    return view('signatures.sign');
}
```

And the view will be named **sign.blade.php** and located at **resources/views/signatures/**

```
@extends('master')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <signature-form></signature-form>
            </div>
        </div>
    </div>
@endsection
```

Working With Laravel Presets

Before 5.5, Laravel has shipped with Bootstrap and Vue.js scaffolding, but not everyone wants to use either of those technologies, so with Laravel 5.5 you can now replace those with the ones you like simply by running this to switch to React

```
php artisan preset react
```

Or this if you are only interested in Bootstrap but not in any of those JS Frameworks:

```
php artisan preset bootstrap
```

Or you can run this if you don't want any scaffolding to be generated out of the box by Laravel:

```
php artisan preset none
```

In our case, we will keep Vue.js and Bootstrap preset, so go ahead and run to install our JavaScript dependencies:

```
npm install
```

Open **/resources/assets//sass/app.scss** and add this styling I already created for our project

```
$color_1: #f14444;
$color_2: #444;
$color_3: #fff;
$border_color_1: #ccc;
$border_color_2: #fff;
$border_color_3: #f14444;

nav.navbar-findcond {
 background: #fff;
 border-color: $border_color_1;
 box-shadow: 0 0 2px 0 #ccc;
 a {
  color: $color_1;
 }
 ul.navbar-nav {
  a {
   color: $color_1;
   border-style: solid;
   border-width: 0 0 2px 0;
   border-color: $border_color_2;
   &:hover {
    background: #fff;
    border-color: $border_color_3;
   }
   &:visited {
    background: #fff;
   }
   &:focus {
    background: #fff;
   }
   &:active {
    background: #fff;
   }
  }
 }
 ul.dropdown-menu {
```

```
    >li {
      >a {
        color: $color_2;
        &:hover {
          background: #f14444;
          color: $color_3;
        }
      }
    }
  }
}
button[type="submit"] {
  border-radius: 2px;
  color: $color_3;
  background: #e74c3c;
  padding: 10px;
  font-size: 13px;
  text-transform: uppercase;
  margin: 0;
  font-weight: 400;
  text-align: center;
  border: none;
  cursor: pointer;
  width: 10%;
  transition: background .5s;
  &:hover {
    background: #2f3c4e;
  }
}
```

Then run this command to compile it:

```
npm run dev
```

Vue.js Components

At this point, the only thing remaining before we launch our amazing app is to create the two components we referenced:

```
<signatures></signatures> <!-- In index.blade.php -->
<signature-form></signature-form> <!-- In sign.blade.php -->
```

Go ahead and create those two files in **/resources/assets/components/js/**

- Signatures.vue

```
<template>
    <div>
        // Our HTML template
    </div>
</template>

<script>
    export default {
        // Our Javascript logic
    }
</script>
```

- SignatureForm.vue

```
<template>
    <div>
        // Our HTML template
    </div>
</template>

<script>
    export default {
        // Our Javascript logic
    }
</script>
```

And register them (right before we create the new Vue instance) so that our app can know about them. Open **/resources/assets/app.js**

```
Vue.component('signatures', require('./components/Signatures.vue'));
Vue.component('signature-form', require('./components/SignatureForm.vue'));

const app = new Vue({
    el: '#app'
});
```

Displaying All Signatures

To display a paginated list of signatures, I will be usingthis package you can install it by running this command:

```
npm install vuejs-paginate --save
```

Register it in our **/resources/assets/app.js** file

```
Vue.component('paginate', require('vuejs-paginate'));
```

And this will be content of our Signatures component:

```
<template>
    <div>
        <div class="panel panel-default" v-for="signature in signatures">
            <div class="panel-heading">
                <span class="glyphicon glyphicon-user" id="start"></span>
                <label id="started">By</label> {{ signature.name }}
            </div>
            <div class="panel-body">
                <div class="col-md-2">
                    <div class="thumbnail">
                        <img :src="signature.avatar" :alt="signature.name">
                    </div>
```

```
                </div>
                <p>{{ signature.body }}</p>
            </div>
            <div class="panel-footer">
                <span class="glyphicon glyphicon-calendar" id="visit"></span> {{ signature.date }} |
                <span class="glyphicon glyphicon-flag" id="comment"></span>
                <a href="#" id="comments" @click="report(signature.id)">Report</a>
            </div>
        </div>
        <paginate
                :page-count="pageCount"
                :click-handler="fetch"
                :prev-text="'Prev'"
                :next-text="'Next'"
                :container-class="'pagination'">
        </paginate>
    </div>
</template>

<script>
    export default {

        data() {
            return {
                signatures: [],
                pageCount: 1,
                endpoint: 'api/signatures?page='
            };
        },

        created() {
            this.fetch();
        },

        methods: {
            fetch(page = 1) {
                axios.get(this.endpoint + page)
                    .then(({data}) => {
                        this.signatures = data.data;
                        this.pageCount = data.meta.last_page;
                    });
            },

            report(id) {
                if(confirm('Are you sure you want to report this signature?')) {
                    axios.put('api/signatures/'+id+'/report')
```

```
                .then(response => this.removeSignature(id));
            }
        },

        removeSignature(id) {
            this.signatures = _.remove(this.signatures, function (signature) {
                return signature.id !== id;
            });
        }
    }
}
</script>
```

As you can see above, when the component gets created we call the **fetch** method which is making a **GET** request to the endpoint we defined in the **data object**, then we set our **signatures** array to the value returned from our API.

In our HTML, we iterate through the **signatures** and display them. When a user clicks on the report link we fire the **report** method which takes the ID of the signature as a param and makes a **PUT** request to hide the reported signature and calls the **removeSignature** which is responsible of removing it from the array.

Sign the GuestBook

For the **SignatureForm** component, we created the form, bound our inputs to our **data object**. When the guest fills in the form and clicks the submit button we fire a **POST** request to save the new signature, if everything goes well we change our **saved** property to true and reset the form, if not we assign our **errors** property to whatever Laravel Validation returns and display them.

```
<template>
    <div>
        <div class="alert alert-success" v-if="saved">
            <strong>Success!</strong> Your signature has been saved successfully.
        </div>

        <div class="well well-sm" id="signature-form">
            <form class="form-horizontal" method="post" @submit.prevent="onSubmit">
                <fieldset>
                    <legend class="text-center">Sign the GuestBook</legend>

                    <div class="form-group">
                        <label class="col-md-3 control-label" for="name">Name</label>
```

```
                    <div class="col-md-9" :class="{'has-error': errors.name}">
                        <input id="name"
                               v-model="signature.name"
                               type="text"
                               placeholder="Your name"
                               class="form-control">
                        <span v-if="errors.name" class="help-block text-danger">{{ errors.name[0] }}</span>
                    </div>
                </div>

                <div class="form-group">
                    <label class="col-md-3 control-label" for="email">Your E-mail</label>
                    <div class="col-md-9" :class="{'has-error': errors.email}">
                        <input id="email"
                               v-model="signature.email"
                               type="text"
                               placeholder="Your email"
                               class="form-control">
                        <span v-if="errors.email" class="help-block text-danger">{{ errors.email[0] }}</span>
                    </div>
                </div>

                <div class="form-group">
                    <label class="col-md-3 control-label" for="body">Your message</label>
                    <div class="col-md-9" :class="{'has-error': errors.body}">
                            <textarea class="form-control"
                                      id="body"
                                      v-model="signature.body"
                                      placeholder="Please enter your message here..."
                                      rows="5"></textarea>
                        <span v-if="errors.body" class="help-block text-danger">{{ errors.body[0] }}</span>
                    </div>
                </div>

                <div class="form-group">
                    <div class="col-md-12 text-right">
                        <button type="submit" class="btn btn-primary btn-lg">Submit</button>
                    </div>
                </div>
            </fieldset>
        </form>
    </div>
</div>
</template>

<script>
```

```
    export default {

        data() {
            return {
                errors: [],
                saved: false,
                signature: {
                    name: null,
                    email: null,
                    body: null,
                }
            };
        },

        methods: {
            onSubmit() {
                this.saved = false;

                axios.post('api/signatures', this.signature)
                    .then(({data}) => this.setSuccessMessage())
                    .catch(({response}) => this.setErrors(response));
            },

            setErrors(response) {
                this.errors = response.data.errors;
            },

            setSuccessMessage() {
                this.reset();
                this.saved = true;
            },

            reset() {
                this.errors = [];
                this.signature = {name: null, email: null, body: null};
            }
        }
    }
</script>
```

After creating these two components or making changes to them, don't forget to run this command to compile them.

```
npm run dev
```

Final Word

I hope you learnt a thing or two from this post, if you are working along and faced an issue I can help with, please do comment below and let me know of it. If you want to have a chat, I am RashidLaasri on Twitter, come and say Hi!

Stay tuned for more Vue.js tutorials and see you soon!

C O D E

Rachid Laasri

5 posts

Yet another web developer.