BUILD A CMS WITH LARAVEL AND VUE - PART 6: ADDING REALTIME COMMENTS

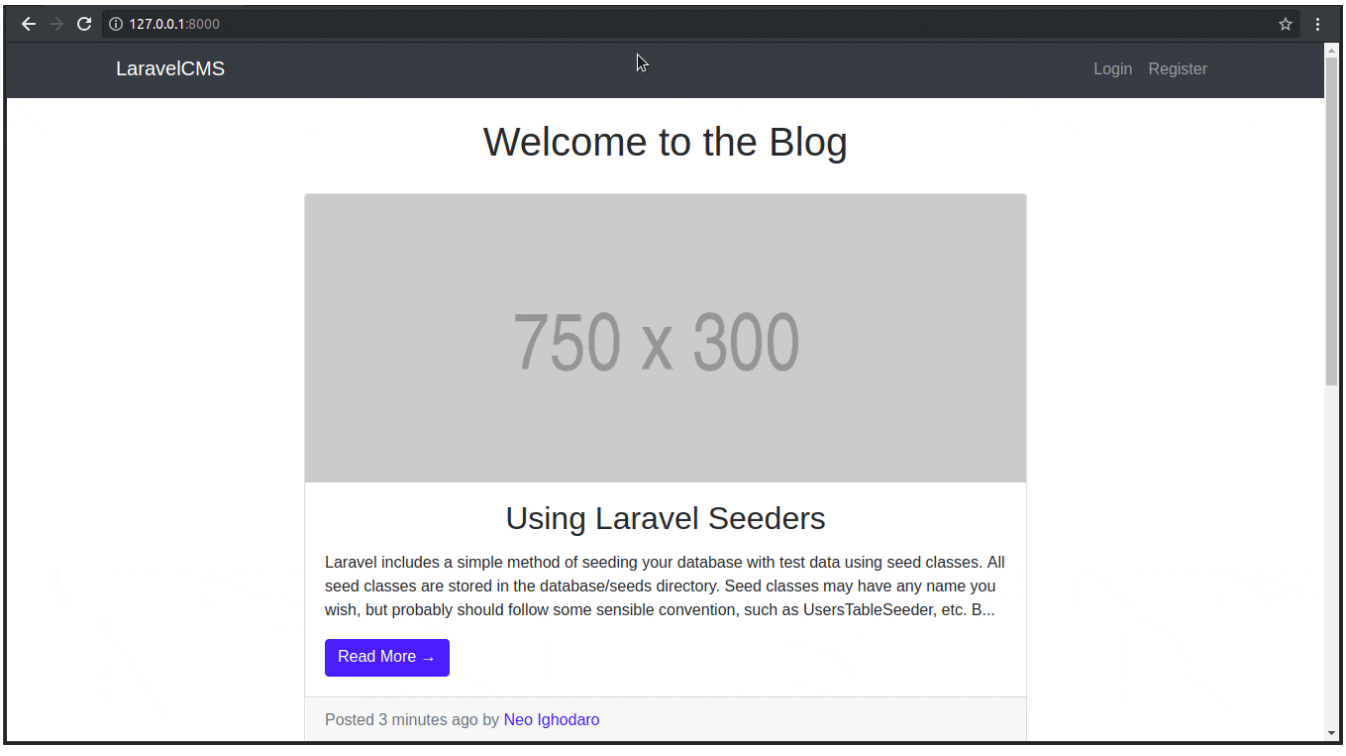
Neo Ighodaro

October 10th, 2018

> Basic knowledge of Laravel and Vue will be helpful.

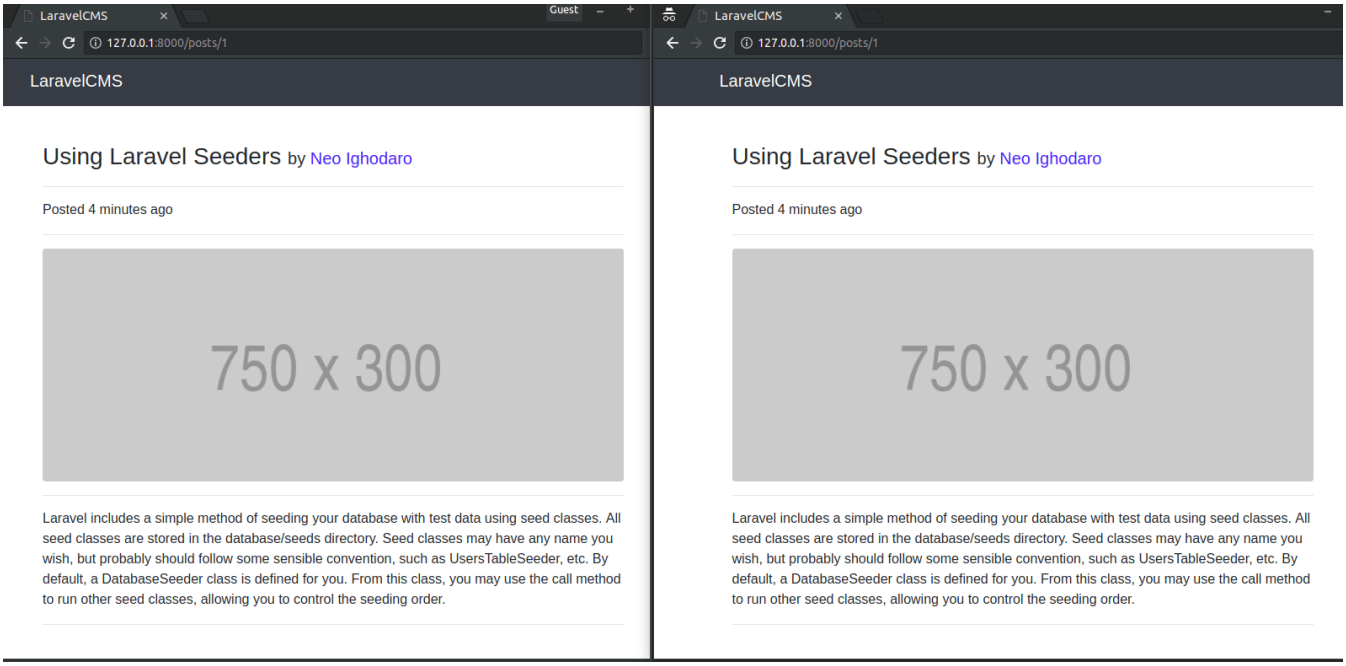
In the previous part of this series, we finished building the backend of the application using Vue. We were able to add the create and update component, which is used for creating a new post and updating an existing post.

Here's a screen recording of what we have been able to achieve:



In this final part of the series, we will be adding support for comments. We will also ensure that the comments on each post are updated in realtime, so a user doesn't have to refresh the page to see new comments.

When we are done, our application will have new features and will work like this:



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

Prerequisites

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

- A Pusher account. Sign up here.
- Basic knowledge of PHP.
- Basic knowledge of the Laravel framework.
- Basic knowledge of JavaScript (ES6 syntax).
- Basic knowledge of Vue.

Adding comments to the backend

When we were creating the API, we did not add the support for comments to the post resource, so we will have to do so now. Open the API project in your text editor as we will be modifying the project a little.

The first thing we want to do is create a model, controller, and a migration for the

comment resource. To do this, open your terminal and `cd` to the project directory and run the following command:

```
$ php artisan make:model Comment -mc
```

The command above will create a model called `Comment`, a controller called `CommentController`, and a migration file in the `database/migrations` directory.

Updating the comments migration file

To update the comments migration navigate to the `database/migrations` folder and find the newly created migration file for the `Comment` model. Let's update the `up()` method in the file:

```
// File: ./database/migrations/*_create_comments_table.php
public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->increments('id');
        $table->timestamps();
        $table->integer('user_id')->unsigned();
        $table->integer('post_id')->unsigned();
        $table->text('body');
    });
}
```

We included `user_id` and `post_id` fields because we intend to create a link between the comments, users, and posts. The `body` field will contain the actual comment.

Defining the relationships among the Comment, User, and Post models

In this application, a comment will belong to a user and a post because a user can make a comment on a specific post, so we need to define the relationship that ties everything up.

Open the `User` model and include this method:

```
// File: ./app/User.php
public function comments()
```

```
    {
        return $this->hasMany(Comment::class);
    }
```

This is a relationship that simply says that a user can have many comments. Now let's define the same relationship on the `Post` model. Open the `Post.php` file and include this method:

```
    // File: ./app/Post.php
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
```

Finally, we will include two methods in the `Comment` model to complete the second half of the relationships we defined in the `User` and `Post` models.

Open the `app/Comment.php` file and include these methods:

```
    // File: ./app/Comment.php
    public function user()
    {
        return $this->belongsTo(User::class);
    }

    public function post()
    {
        return $this->belongsTo(Post::class);
    }
```

Since we want to be able to mass assign data to specific fields of a comment instance during comment creation, we will include this array of permitted assignments in the `app/Comment.php` file:

```
    protected $fillable = ['user_id', 'post_id', 'body'];
```

We can now run our database migration for our comments:

```
    $ php artisan migrate
```

Configuring Laravel to broadcast events using Pusher

We already said that the comments will have a realtime functionality and we will be building this using Pusher, so we need to enable Laravel's event broadcasting feature.

Open the `config/app.php` file and uncomment the following line in the `providers` array:

```
App\Providers\BroadcastServiceProvider
```

Next, we need to configure the broadcast driver in the `.env` file:

```
BROADCAST_DRIVER=pusher
```

Let's pull in the Pusher PHP SDK using composer:

```
$ composer require pusher/pusher-php-server
```

Configuring Pusher

For us to use Pusher in this application, it is a prerequisite that you have a Pusher account. You can create a free Pusher account here then login to your dashboard and create an app.

Once you have created an app, we will use the app details to configure pusher in the `.env` file:

```
PUSHER_APP_ID=xxxxxx
PUSHER_APP_KEY=xxxxxxxxxxxxxxxxxxx
PUSHER_APP_SECRET=xxxxxxxxxxxxxxxxxxx
PUSHER_APP_CLUSTER=xx
```

Update the Pusher keys with the app credentials provided for you under the **Keys** section on the **Overview** tab on the Pusher dashboard.

Broadcasting an event for when a new comment is sent

To make the comment update realtime, we have to broadcast an event based on the comment creation activity. We will create a new event and call it `CommentSent`. It is to be fired when there is a successful creation of a new comment.

Run command in your terminal:

```
php artisan make:event CommentSent
```

There will be a newly created file in the `app\Events` directory, open the `CommentSent.php` file and ensure that it implements the `ShouldBroadcast` interface.

Open and replace the file with the following code:

```
// File: ./app/Events/CommentSent.php
<?php

namespace App\Events;

use App\Comment;
use App\User;
use Illuminate\Broadcasting\Channel;
use Illuminate\Queue\SerializesModels;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class CommentSent implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $user;

    public $comment;

    public function __construct(User $user, Comment $comment)
    {
        $this->user = $user;

        $this->comment = $comment;
    }

    public function broadcastOn()
```

```
        {
            return new PrivateChannel('comment');
        }
    }
```

In the code above, we created two public properties, `user` and `comment`, to hold the data that will be passed to the channel we are broadcasting on. We also created a private channel called `comment`. We are using a private channel so that only authenticated clients can subscribe to the channel.

Defining the routes for handling operations on a comment

We created a controller for the comment model earlier but we haven't defined the web routes that will redirect requests to be handled by that controller.

Open the `routes/web.php` file and include the code below:

```
// File: ./routes/web.php
Route::get('/{post}/comments', 'CommentController@index');
Route::post('/{post}/comments', 'CommentController@store');
```

Setting up the action methods in the CommentController

We need to include two methods in the `CommentController.php` file, these methods will be responsible for storing and retrieving methods. In the `store()` method, we will also be broadcasting an event when a new comment is created.

Open the `CommentController.php` file and replace its contents with the code below:

```
// File: ./app/Http/Controllers/CommentController.php
<?php

namespace App\Http\Controllers;

use App\Comment;
use App\Events\CommentSent;
use App\Post;
use Illuminate\Http\Request;

class CommentController extends Controller
{
    public function store(Post $post)
    {
```

```
    {
        $this->validate(request(), [
            'body' => 'required',
        ]);

        $user = auth()->user();

        $comment = Comment::create([
            'user_id' => $user->id,
            'post_id' => $post->id,
            'body' => request('body'),
        ]);

        broadcast(new CommentSent($user, $comment))->toOthers();

        return ['status' => 'Message Sent!'];
    }

    public function index(Post $post)
    {
        return $post->comments()->with('user')->get();
    }
}
```

In the `store` method above, we are validating then creating a new post comment. After the comment has been created, we broadcast the `CommentSent` event to other clients so they can update their comments list in realtime.

In the `index` method we just return the comments belonging to a post along with the user that made the comment.

Adding a layer of authentication

Let's add a layer of authentication that ensures that only authenticated users can listen on the private `comment` channel we created.

Add the following code to the `routes/channels.php` file:

```
// File: ./routes/channels.php
Broadcast::channel('comment', function ($user) {
    return auth()->check();
});
```

Adding comments to the frontend

In the second article of this series, we created the view for the single post landing page in the `single.blade.php` file, but we didn't add the comments functionality. We are going to add it now. We will be using Vue to build the comments for this application so the first thing we will do is include Vue in the frontend of our application.

Open the master layout template and include Vue to its `<head>` tag. Just before the `<title>` tag appears in the `master.blade.php` file, include this snippet:

```
<!-- File: ./resources/views/layouts/master.blade.php -->
<meta name="csrf-token" content="{{ csrf_token() }}">
<script src="{{ asset('js/app.js') }}" defer></script>
```

The `csrf_token()` is there so that users cannot forge requests in our application. All our requests will pick the randomly generated `csrf-token` and use that to make requests.

Related: CSRF in Laravel: how VerifyCsrfToken works and how to prevent attacks

Now the next thing we want to do is update the `resources/assets/js/app.js` file so that it includes a template for the comments view.

Open the file and replace its 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'
import Comments from './components/Comments'

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',
        components: { Homepage, Comments },
        router,
    });
```

Above we imported the `Comment` component and then we added it to the list of components in the applications Vue instance.

Now create a `Comments.vue` file in the `resources/assets/js/components` directory. This is where all the code for our comment view will go. We will populate this file later on.

Installing Pusher and Laravel Echo

For us to be able to use Pusher and subscribe to events on the frontend, we need to pull in both Pusher and Laravel Echo. We will do so by running this command:

```
$ npm install --save laravel-echo pusher-js
```

> Laravel Echo is a JavaScript library that makes it easy to subscribe to channels and listen for events broadcast by Laravel.

Now let's configure Laravel Echo to work in our application. In the `resources/assets/js/bootstrap.js` file, find and uncomment this snippet of code:

```
import Echo from 'laravel-echo'

window.Pusher = require('pusher-js');

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: process.env.MIX_PUSHER_APP_KEY,
    cluster: process.env.MIX_PUSHER_APP_CLUSTER,
    encrypted: true
});
```

> The `key` and `cluster` will pull the keys from your `.env` file so no need to enter them manually again.

Now let's import the `Comments` component into the `single.blade.php` file and pass along the required the props.

Open the `single.blade.php` file and replace its contents with the code below:

```
{{-- File: ./resources/views/single.blade.php --}}
@extends('layouts.master')

@section('content')
<div class="container">
  <div class="row">
    <div class="col-lg-10 mx-auto">
      <br>
      <h3 class="mt-4">
        {{ $post->title }}
        <span class="lead">by <a href="#">{{ $post->user->name }}</a></spa
      </h3>
      <hr>
      <p>Posted {{ $post->created_at->diffForHumans() }}</p>
      <hr>
      <img class="img-fluid rounded" src="{!! !empty($post->image) ? '/upl
      <hr>
      <div>
        <p>{{ $post->body }}</p>
        <hr>
        <br>
```

```
            </div>

            @auth
            <Comments
                :post-id='@json($post->id)'
                :user-name='@json(auth()->user()->name)'>
            </Comments>
            @endauth
          </div>
        </div>
      </div>
      @endsection
```

Building the comments view

Open the `Comments.vue` file and add the following markup template below:

```
<template>
  <div class="card my-4">
    <h5 class="card-header">Leave a Comment:</h5>
    <div class="card-body">
      <form>
        <div class="form-group">
          <textarea ref="body" class="form-control" rows="3"></textarea>
        </div>
        <button type="submit" @click.prevent="addComment" class="btn btn-p
          Submit
        </button>
      </form>
    </div>
    <p class="border p-3" v-for="comment in comments">
       <strong>{{ comment.user.name }}</strong>:
       <span>{{ comment.body }}</span>
    </p>
  </div>
</template>
```

Now, we'll add a script that defines two methods:

1. `fetchComments()` - this will fetch all the existing comments when the component is created.
2. `addComment()` - this will add a new comment by hitting the backend server. It will also trigger a new event that will be broadcast so all clients receive them in realtime.

In the same file, add the following below the closing `template` tag:

```
<script>
export default {
  props: {
    userName: {
      type: String,
      required: true
    },
    postId: {
      type: Number,
      required: true
    }
  },
  data() {
    return {
      comments: []
    };
  },

  created() {
    this.fetchComments();

    Echo.private("comment").listen("CommentSent", e => {
        this.comments.push({
          user: {name: e.user.name},
          body: e.comment.body,
        });
    });
  },

  methods: {
    fetchComments() {
      axios.get("/" + this.postId + "/comments").then(response => {
        this.comments = response.data;
      });
    },

    addComment() {
      let body = this.$refs.body.value;
      axios.post("/" + this.postId + "/comments", { body }).then(response
        this.comments.push({
          user: {name: this.userName},
          body: this.$refs.body.value
        });
        this.$refs.body.value = "";
      });
    }
```

```
    }
};
</script>
```

In the `created()` method above, we first made a call to the `fetchComments()` method, then we created a listener to the private `comment` channel using Laravel Echo. Once this listener is triggered, the `comments` property is updated.

Testing the application

Now let's test the application to see if it is working as intended. Before running the application, we need to refresh our database so as to revert any changes. To do this, run the command below in your terminal:

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

Next, let's build the application so that all the changes will be compiled and included as a part of the JavaScript file. To do this, run the following command on your terminal:

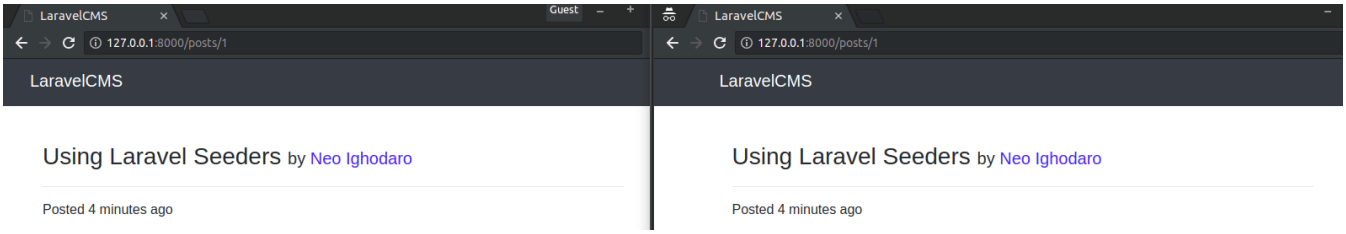
```
$ npm run dev
```

Finally, let's serve the application using this command:

```
$ php artisan serve
```

To test that our application works visit the application URL http://localhost:8000 on two separate browser windows, we will log in to our application on each of the windows as a different user.

We will finally make a comment on the same post on each of the browser windows and check that it updates in realtime on the other window:



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. By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order.

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. By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order.

Conclusion

In this final tutorial of this series, we created the comments feature of the CMS and also made it realtime. We were able to accomplish the realtime functionality using Pusher.

In this entire series, we learned how to build a CMS using Laravel and Vue.

The source code for this article series is available here on Github.

JAVASCRIPT LARAVEL PHP VUE.JS CHANNELS