Application de chat utilisant Laravel vue.js et pusher

Pour créer une application de chat dans Laravel avec Vue.js, utilisez le service de messagerie en temps réel de Pusher et, côté client, utilisez la bibliothèque Laravel Echo et Pusher.js pour mettre à jour notre interface utilisateur en temps réel.

1 - Créer un projet Laravel

```
# composer create-project --prefer-dist laravel/laravel laravel-chat
"8.*"
```

Allez dans le dossier du projet.

```
# cd laravel-chat
```

Installer toutes les dépendances frontales à l'aide de la commande suivante.

```
# npm install
```

Créez un échafaudage d'authentification fourni par Laravel.

```
# composer require laravel/ui
# php artisan ui vue --auth
```

https://laravel.com/docs/8.x/authentication

Installer Bootstrap

```
# php artisan ui bootstrap
# npm install
# npm run dev
```

Installer Vue.JS

```
# php artisan ui vue
# npm install
```

Installer loader

```
# npm install vue-loader@^15.9.7 --save-dev --legacy-peer-deps
# npm update vue-loader
# npm install
# npm run dev
```

Ouvrez l'éditeur MySQL et créez la base de données

```
# create database laravel_chat;
```

Configurez la base de données dans le fichier .env.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_chat
DB_USERNAME=root
DB_PASSWORD=
```

Accédez au terminal et créez les tables à partir des migrations.

ATTENTION

Il y a un bogue dans Laravel> 5.4 si vous utilisez une version de MySQL antérieure à 5.7.7 ou MariaDB antérieure à 10.2.2. Cela peut être corrigé en remplaçant le boot() de :

app\Providers\AppServiceProvider.php

```php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
```

```
    {
        Schema::defaultStringLength(191);
    }
}
```

Exécutez la commande après le changement, si nécessaire

```
# php artisan migrate
```

Exécutez le projet laravel :

```
# php artisan serve
```

Maintenant, enregistrez 2 utilisateurs sur http://localhost:8000/register.

2 - Modèle de message et migration

Créez un modèle de message avec le fichier de migration en exécutant la commande :

```
# php artisan make:model Message -m
```

Ouvrez le modèle Message et ajoutez-y le code ci-dessous :

app\Models\Message.php

```
class Message extends Model
{
    use HasFactory;

    protected $fillable = ['message'];
}
```

Dans le répertoire databases/migrations, ouvrez la carte des messages de migration qui a été créée lorsque nous avons exécuté la commande ci-dessus et mettez à jour la méthode up avec:

database\migrations\2022_05_17_173811_create_messages_table.php

```
    public function up()
    {
        Schema::create('messages', function (Blueprint $table) {
            $table->id();
            $table->integer('user_id')->unsigned();
            $table->text('message');
```

```
            $table->timestamps();
            $table->foreign('user_id')->references('id')->on('users')-
>onDelete('cascade');
        });
    }
```

Le message aura cinq colonnes : un identifiant d'incrémentation automatique, user_id, message, created_at et updated_at. La colonne user_id contiendra l'ID de l'utilisateur qui a envoyé un message et la colonne message contiendra le message réel qui a été envoyé.

Exécutez la migration :

```
# php artisan migrate
```

3 - Relation entre l'utilisateur et le message

Nous devons configurer la relation entre un utilisateur et un message. Un utilisateur peut envoyer plusieurs messages alors qu'un message particulier a été envoyé par un utilisateur. Ainsi, la relation entre l'utilisateur et le message est une relation un à plusieurs. Pour définir cette relation, ajoutez le code ci-dessous au modèle utilisateur :

app\Models\User.php

```
    public function messages()
    {
    return $this->hasMany(Message::class);
    }
```

Ensuite, nous devons définir la relation inverse en ajoutant le code ci-dessous au modèle de message :

app\Models\Message.php

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

4 - ChatsController

Créez le contrôleur qui gérera la logique de notre application de chat. Créez un ChatsController avec la commande ci-dessous :

```
# php artisan make:controller ChatsController
```

Ouvrez le nouveau fichier create app/Http/Controllers/ChatsController.php et ajoutez-y le code suivant :

app\Http\Controllers\ChatsController.php

```
namespace App\Http\Controllers;

use App\Models\Message;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;


class ChatsController extends Controller
{
        public function __construct()
    {
    $this->middleware('auth');
    }

    /**
     * Show chats
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
    return view('chat');
    }

    /**
     * Fetch all messages
     *
     * @return Message
     */
    public function fetchMessages()
    {
    return Message::with('user')->get();
    }

    /**
     * Persist message to database
     *
     * @param  Request $request
     * @return Response
     */
```

```
    public function sendMessage(Request $request)
    {
    $user = Auth::user();

    $message = $user->messages()->create([
        'message' => $request->input('message')
    ]);

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

}
```

L'utilisation du middleware auth dans __contruct() de ChatsController indique que toutes les méthodes avec le contrôleur ne seront accessibles qu'aux utilisateurs autorisés. Ensuite, l'index () renverra simplement un fichier de vue. Le fetchMessages() renvoie un JSON de tous les messages avec leurs utilisateurs. Enfin, sendMessage() conservera le message dans la base de données et renverra un message d'état.

5 - Définition des itinéraires (routes) d'application

Créer les itinéraires (routes) dont notre application de chat aura besoin. Ouvrez routes/web.php et remplacez les routes par le code ci-dessous pour définir trois routes simples :

routes\web.php

```
use App\Http\Controllers\ChatsController;
```

```
Route::get('/chat', [ChatsController::class, 'index'])->name('chat');
Route::get('messages', [ChatsController::class, 'fetchMessages'])-
>name('messages');
Route::post('/messages', [ChatsController::class, 'sendMessage'])-
>name('send.messages');
```

La page d'accueil affichera des messages de chat et un champ de saisie pour saisir de nouveaux messages. Une route de messages GET récupérera tous les messages de discussion et une route de messages POST sera utilisée pour envoyer de nouveaux messages.

6 - Création de la vue de l'application de chat

Nous allons mettre les messages de chat dans une card Bootstrap pour lui donner une belle apparence.

Créez un nouveau fichier resources/views/chat.blade.php et collez-y :

resources\views\chat.blade.php

```
@extends('layouts.app')
@section('content')
<div class="container">
    <div class="card">
        <div class="card-header">Chats</div>
        <div class="card-body">
            <chat-messages :messages="messages"></chat-messages>
        </div>
        <div class="card-footer">
            <chat-form v-on:messagesent="addMessage" :user="{{ Auth::user()
}}"></chat-form>
        </div>
    </div>
</div>
@endsection
```

Notez que nous avons deux balises personnalisées : chat-messages et chat-form, ce sont des composants Vue que nous créerons bientôt. Le composant de messages de chat affichera nos messages de chat et le formulaire de chat fournira un champ de saisie et un bouton pour envoyer les messages. Remarquez v-on:messagesent="addMessage" :user="{{ Auth::user() }}": plus tard, cela recevra un événement messagesent du composant chat-form et transmettra la propriété user au chat-form .

Sur la page d'accueil, créez un bouton pour accéder au chat, après le corps de la carte div.

resources\views\home.blade.php

```
 <div class="card-footer">
     <a href="{{route('chat')}}" class="btn btn-secondary btn-ouline">Chat</a>
 </div>
```

Créez un nouveau fichier ChatMessages.vue dans resources/js/components et collez-y le code ci-dessous :

resources\js\components\ChatMessages.vue

```
<template>
  <ul class="chat">
```

```
    <li class="left clearfix" v-for="message in messages" :key="message.id">
      <div class="clearfix">
        <div class="header">
          <strong>
            {{ message.user.name }}
          </strong>
        </div>
        <p>
          {{ message.message }}
        </p>
      </div>
    </li>
  </ul>
</template>

<script>
export default {
  props: ["messages"],
};
</script>
```

Ce composant accepte un tableau de messages en tant que "props", les parcourt en boucle et affiche le nom de l'utilisateur qui a envoyé le message et le corps du message.

Créez un nouveau fichier ChatForm.vue dans resources/js/components et collez-y le code ci-dessous.

resources\js\components\ChatForm.vue

```
<template>
  <div class="input-group">
    <input
      id="btn-input"
      type="text"
      name="message"
      class="form-control input-sm"
      placeholder="Type your message here..."
      v-model="newMessage"
      @keyup.enter="sendMessage"
    />
    <span class="input-group-btn">
      <button class="btn btn-primary btn-sm" id="btn-chat" @click="sendMessage">
        Send
      </button>
    </span>
```

```
  </div>
</template>

<script>
export default {
  props: ["user"],
  data() {
    return {
      newMessage: "",
    };
  },
  methods: {
    sendMessage() {
      this.$emit("messagesent", {
        user: this.user,
        message: this.newMessage,
      });
      this.newMessage = "";
    },
  },
};
</script>
```

Si vous jetez un oeil à resources/views/chat.blade.php, vous remarquerez dans la balise <chat-form> qu'il y a un v-on:messagesent="addMessage". Cela signifie que lorsqu'un événement de message est émis à partir du code ci-dessus, il sera géré par la méthode addMessage() dans l'instance Vue racine resources/js/app.js.

Ouvrez le fichier resources/assets/js/app.js et mettez à jour avec le code ci-dessous :

resources\js\app.js

```
/**
 * First we will load all of this project's JavaScript dependencies which
 * includes Vue and other libraries. It is a great starting point when
 * building robust, powerful web applications using Vue and Laravel.
 */
const { default: Echo } = require('laravel-echo');

require('./bootstrap');

window.Vue = require('vue').default;

/**
```

```
 * The following block of code may be used to automatically register your
 * Vue components. It will recursively scan this directory for the Vue
 * components and automatically register them with their "basename".
 *
 * Eg. ./components/ExampleComponent.vue -> <example-component></example-
component>
 */

// const files = require.context('./', true, /\.vue$/i)
// files.keys().map(key => Vue.component(key.split('/').pop().split('.')[0],
files(key).default))
Vue.component('example-component',
require('./components/ExampleComponent.vue').default);

Vue.component('chat-messages', require('./components/ChatMessages.vue').default);
Vue.component('chat-form', require('./components/ChatForm.vue').default);
/**
 * Next, we will create a fresh Vue application instance and attach it to
 * the page. Then, you may begin adding components to this application
 * or customize the JavaScript scaffolding to fit your unique needs.
 */

 const app = new Vue({
    el: '#app',
    data: {
        messages: []
    },
    created() {
        this.fetchMessages();
    },
    methods: {
        fetchMessages() {
            axios.get('/messages').then(response => {
                this.messages = response.data;
            });
        },
        addMessage(message) {

            this.messages.push(message);
            axios.post('/messages', message)  .then(response => {
                console.log(response)
             })
             .catch(error => {
               console.log(error.response)
             });
```

```
        }
    }
});
```

Une fois l'instance Vue créée, à l'aide d'Axios, nous effectuons une requête GET sur la route des messages et récupérons tous les messages, puis la transmettons au tableau de messages qui sera affiché sur la vue de discussion. addMessage() reçoit le message qui a été émis par le composant ChatForm, le pousse vers le tableau de messages et effectue une requête POST sur la route des messages avec le message.

7 - Réglage du pusher

Avant de commencer à utiliser la diffusion d'événements (Broadcasting) Laravel, nous devons d'abord enregistrer le App\Providers\BroadcastServiceProvider. Ouvrez config/app.php et décommentez la ligne suivante dans le tableau providers.

config\app.php

```
        App\Providers\BroadcastServiceProvider::class,
```

Nous devons dire à Laravel que nous utilisons le pilote Pusher dans le fichier .env :

.env

```
BROADCAST_DRIVER=pusher
```

Bien que Laravel supporte Pusher prêt à l'emploi, nous devons encore installer le SDK PHP Pusher. Nous pouvons le faire en utilisant composer:

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

Une fois l'installation terminée, nous devons configurer les informations d'identification de notre application Pusher dans config/broadcasting.php. Pour obtenir nos informations d'identification pour l'application Pusher, nous devons avoir un compte Pusher.

L'étape suivante consiste à créer une application pusher.

Accédez à cette URL : https://pusher.com/

Maintenant, vous allez rediriger vers cette page : https://dashboard.pusher.com

Créez une application et accédez à la section Clés d'application. Vérifiez également votre cluster. Maintenant, récupérez toutes les données et placez-les dans le fichier .env.

```
PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
PUSHER_APP_CLUSTER=mt1
```

Pour vous abonner et écouter des événements, Laravel fournit Laravel Echo, qui est une bibliothèque JavaScript qui facilite l'abonnement aux chaînes et l'écoute des événements diffusés par Laravel. Nous devrons l'installer avec la bibliothèque JavaScript Pusher :

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

Une fois installé, nous devons dire à Laravel Echo d'utiliser Pusher. Au bas du fichier resources/assets/js/bootstrap.js, Laravel a supprimé l'intégration d'Echo bien qu'elle soit commentée. Décommentez simplement la section Laravel Echo et mettez à jour les détails avec:

resources\js\bootstrap.js

```
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,
    forceTLS: true
});
```

8 - Événement de diffusion de message envoyé

Pour ajouter les interactions en temps réel à notre application de chat, nous devons diffuser une sorte d'événements basés sur certaines activités. Dans notre cas, nous déclencherons un MessageSent lorsqu'un utilisateur envoie un message. Tout d'abord, nous devons créer un événement, nous l'appellerons MessageSent :

```
# php artisan make:event MessageSent
```

Cela créera une nouvelle classe d'événements MessageSent dans le répertoire app/Events. Cette classe doit implémenter l'interface ShouldBroadcast. La classe devrait ressembler à :

app\Events\MessageSent.php

```
<?php
```

```php
namespace App\Events;

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

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

    public $user;
    public $message;

    public function __construct(User $user, Message $message)
    {
        $this->user = $user;
        $this->message = $message;
    }



    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('chat');
    }

}
```

Nous avons défini deux propriétés publiques qui seront les données qui seront transmises à la chaîne sur laquelle nous diffusons.

REMARQUE : ces propriétés doivent être publiques pour être transmises au canal.

Étant donné que notre application de chat est une application authentifiée uniquement, nous créons un canal privé appelé Chat, auquel seuls les utilisateurs authentifiés pourront se connecter. En utilisant la classe PrivateChannel, Laravel est assez intelligent pour savoir que nous créons un canal privé, donc ne préfixez pas le nom du canal avec private- (comme spécifié par Pusher), Laravel ajoutera le préfixe private.

Ensuite, nous devons mettre à jour le sendMessage() de ChatsController pour diffuser l'événement MessageSent :*app\Http\Controllers\ChatsController.php*

```
use App\Events\MessageSent;
```

```
    public function sendMessage(Request $request)
    {
        $user = Auth::user();

        $message = $user->messages()->create([
          'message' => $request->input('message')
        ]);

        broadcast(new MessageSent($user, $message))->toOthers();

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

Depuis que nous avons créé un canal privé, seuls les utilisateurs authentifiés pourront écouter sur le canal de chat. Nous avons donc besoin d'un moyen d'autoriser que l'utilisateur actuellement authentifié puisse réellement écouter sur le canal. Cela peut être fait par dans le fichier routes/channels.php :

routes\channels.php

```
Broadcast::channel('App.Models.User.{id}', function ($user, $id) {
    return (int) $user->id === (int) $id;
});

Broadcast::channel('chat', function ($user) {
    return Auth::check();
});
```

Nous passons au channel(), le nom de notre channel et une fonction callback qui renverra true ou false selon que l'utilisateur courant est authentifié ou non.

Désormais, lorsqu'un message est envoyé, l'événement MessageSent sera diffusé à Pusher. Nous utilisons toOthers() qui nous permet d'exclure l'utilisateur actuel des destinataires de la diffusion.

9 - Événement d'écoute des messages envoyés

Une fois l'événement MessageSent diffusé, nous devons écouter cet événement afin de pouvoir mettre à jour les messages de chat avec le message nouvellement envoyé. Nous pouvons le faire en ajoutant l'extrait de code ci-dessous à created() of resources/assets/js/app.js juste après this.fetchMessages() :

resources\js\app.js

```
    created() {
        this.fetchMessages();
        window.Echo.private('chat')
        .listen('MessageSent', (e) => {
            this.messages.push({
            message: e.message.message,
            user: e.user
            });
        });
    },
```

Nous nous abonnons au canal de chat en utilisant le private() d'Echo puisque le canal est un canal privé. Une fois abonné, nous écoutons le MessageSent et sur cette base, mettons à jour le tableau des messages de chat avec le message nouvellement envoyé.

Avant de tester notre application de chat, nous devons compiler les fichiers JavaScript à l'aide de Laravel Mix en utilisant :

```
# npm run dev
```

Nous pouvons maintenant démarrer notre application de chat en exécutant :

```
# php artisan serve
```

Notre application de chat est terminée car nous pouvons désormais envoyer et recevoir des messages en temps réel.

Ouvrez un navigateur de section privée avec un compte et un deuxième navigateur de section privée avec un autre, et vérifiez si les messages peuvent être envoyés de l'un à l'autre.

Remarque : Si les messages ne s'envoient pas, ouvrez la console de votre navigateur et vérifiez s'il y a une erreur 500 cURL error 60. Si c'est le cas, installez le fichier carcet.perm dans WAMP/Xampp et configurez le php. Ini

1. Téléchargez le dernier fichier cacert.pem. (https://curl.haxx.se/docs/caextract.html)

2. Déplacez le fichier vers l'emplacement souhaité. Je le garde dans le dossier wamp64. Le chemin d'accès au fichier est D:/wamp64/cacert.pem

3. Nous devons maintenant ajouter ce chemin dans les fichiers php.ini situés dans différents dossiers.

4. Cliquez avec le bouton gauche sur l'icône wampserver dans la barre d'état système. Sélectionnez PHP > php.ini. (Ce fichier se trouve dans le dossier Apache.) Et ajoutez le chemin d'accès au paramètre curl.cainfo comme ci-dessous.

```
; A default value for the CURLOPT_CAINFO option.
; This is required to be an absolute path.
curl.cainfo="D:/wamp64/cacert.pem"
```

https://insidert.medium.com/fix-curl-error-60-in-wamp-server-a0ffff8dbb29

Références:

https://appdividend.com/2022/03/01/laravel-vue-chat-application/amp/

https://pusher.com/tutorials/chat-laravel/

https://pusher.com/tutorials/how-to-build-a-chat-app-with-vue-js-and-laravel/

https://github.com/markitosanches/laravel-vue-pusher-chat