BUILD A CHAT APP WITH LARAVEL

Chimezie Enyinnaya

November 7th, 2017

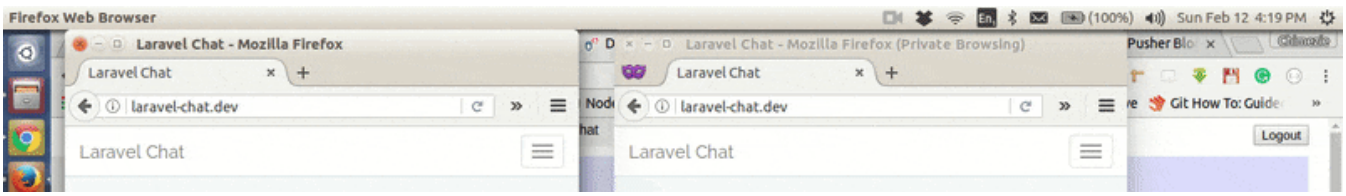
> A basic understanding of Laravel and Vue.js is needed to follow this tutorial.

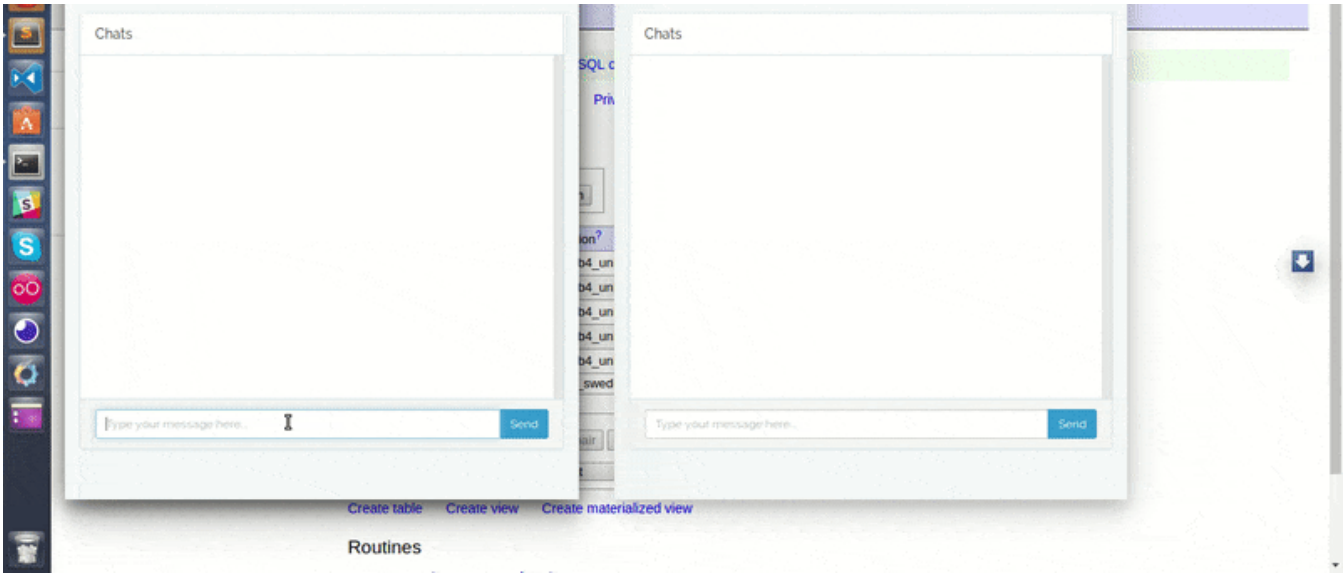
Laravel makes it easy to build modern applications with realtime interactions by providing an event broadcasting system which allows developers to share the same event names between the server-side code and the client-side JavaScript application.

Pusher, on the other hand, is an easy and reliable platform for building scalable realtime applications. Laravel provides support for Pusher out of the box, which makes building realtime applications with Laravel and Pusher seamless. In fact, Pusher has emerged as one of the Laravel community's preferred tools to make apps realtime, thanks to the support of Taylor Otwell, Jeffrey Way, Matt Stauffer, and many more.

In this post, I will be showing you how to build a laravel chat application with Pusher. I will be using Vue.js as my JavaScript framework, although you can use the JavaScript framework of your choice or even jQuery and vanilla JavaScript.

Before we start, let's take a quick look at what we'll be building.





The code of the completed demo is available on GitHub.

Setting Up Laravel

We'll start by creating a new Laravel project. While there are different ways of creating a new Laravel project, I prefer using the Laravel installer. Open your terminal and run the code below:

```
laravel new laravel-chat
```

This will create a `laravel-chat` project within the directory where you ran the command above.

Before we start using Laravel event broadcasting, we first need to register the `App\Providers\BroadcastServiceProvider`. Open `config/app.php` and uncomment the following line in the `providers` array.

```
// App\Providers\BroadcastServiceProvider
```

We need to tell Laravel that we are using the Pusher driver in the `.env` file:

```
// .env

BROADCAST_DRIVER=pusher
```

Though Laravel supports Pusher out of the box, we still need to install the Pusher PHP SDK. We can do this using composer:

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

Once the installation is done, we need to configure our Pusher app credentials in `config/broadcasting.php`. To get our Pusher app credential, we need to have a Pusher account.

Setting Up Pusher

If you don't have one already, create a free Pusher account at https://pusher.com/signup then login to your dashboard and create an app.

Now, let's fill in our Pusher app credentials. If you open the `config/broadcasting.php`, you'll notice that Laravel is pulling some of Pusher credential from the `.env` file:

```
// Don't add your credentials here!
// config/broadcasting.php

'pusher' => [
  'driver' => 'pusher',
  'key' => env('PUSHER_APP_KEY'),
  'secret' => env('PUSHER_APP_SECRET'),
  'app_id' => env('PUSHER_APP_ID'),
  'options' => [],
],
```

We need to modify the source a little bit here to get this to work. Modify the source so that it looks like this:

```
  'pusher' => [
      'driver' => 'pusher',
      'key' => env('PUSHER_APP_KEY'),
      'secret' => env('PUSHER_APP_SECRET'),
      'app_id' => env('PUSHER_APP_ID'),
      'options' => [
          'cluster' => env('PUSHER_CLUSTER'),
          'encrypted' => true,
      ],
  ],
```

Then let's update the `.env` file to contain our Pusher app credentials (noting the added cluster credential, this won't be in your `.env` file as Laravel has not added this functionality yet:

```
// .env

PUSHER_APP_ID=xxxxxx
PUSHER_APP_KEY=xxxxxxxxxxxxxxxxxxxx
PUSHER_APP_SECRET=xxxxxxxxxxxxxxxxxxxx
PUSHER_CLUSTER=xx
```

Remember to replace the `x`s with your Pusher app credentials. You can find your app credentials under the **Keys** section on the **Overview** tab.

Now that we've set up the back-end of our project, let's move on to setting up the front-end. Laravel provides some front-end frameworks and libraries, including - `Bootstrap`, `Vuejs` and `Axios` which we'll be using in this tutorial.

We'll also be making use of Laravel Mix, which is a wrapper around Webpack that will help us compile our CSS and JavaScript.

But first, we need to install these dependencies through `NPM`:

```
npm install
```

To subscribe and listen to events, Laravel provides Laravel Echo, which is a JavaScript library that makes it painless to subscribe to channels and listen for events broadcast by Laravel. We'll need to install it along with the Pusher JavaScript library:

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

Once installed, we need to tell Laravel Echo to use Pusher. At the bottom of the `resources/assets/js/bootstrap.js` file, Laravel have stubbed Echo integration though it is commented out. Simply uncomment the Laravel Echo section and update the details with:

```
// resources/assets/js/bootstrap.js
```

```
import Echo from "laravel-echo"

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'xxxxxxxxxxxxxxxxxxxx',
    cluster: 'eu',
    encrypted: true
});
```

Remember to replace the `x`s with your Pusher app key. Also use the same `cluster` that you specified earlier in `config/broadcasting.php`.

Now that we are done with setting up Laravel and Pusher and other dependencies, it time to start building our chat application.

Authenticating Users

Our chat app will require users to be logged in before they can begin to chat. So, we need an authentication system, which with Laravel is as simple as running an `artisan` command in the terminal:

```
php artisan make:auth
```

This will create the necessary routes, views and controllers needed for an authentication system.

Before we go on to create users, we need to run the `users` migration that comes with a fresh installation of Laravel. But to do this, we first need to setup our database. Open the `.env` file and enter your database details:

```
// .env

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel-chat
DB_USERNAME=root
DB_PASSWORD=root
```

Update with your own database details. Now, we can run our migration:

```
php artisan migrate
```

There's a bug in Laravel 5.4 if you're running a version of MySQL older than 5.7.7 or MariaDB older than 10.2.2. More info here. This can be fixed by replacing the `boot()` of `app/Providers/AppServiceProvider.php` with:

```
// app/Providers/AppServiceProvider.php

// remember to use
Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Schema::defaultStringLength(191);
}
```

Message Model and Migration

Create a `Message` model along with the migration file by running the command:

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

Open the `Message` model and add the code below to it:

```
// app/Message.php

/**
 * Fields that are mass assignable
 *
 * @var array
 */
protected $fillable = ['message'];
```

Within the `databases/migrations` directory, open the `messages` table migration that was created when we ran the command above and update the `up` method with:

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

The `message` will have five columns: an auto increment `id`, `user_id`, `message`, `created_at` and `updated_at`. The `user_id` column will hold the ID of the user that sent a message and the `message` column will hold the actual message that was sent. Run the migration:

```
php artisan migrate
```

User To Message Relationship

We need to setup the relationship between a user and a message. A user can send many messages while a particular message was sent by a user. So, the relationship between the user and message is a one to many relationship. To define this relationship, add the code below to `User` model:

```
// app/User.php

/**
 * A user can have many messages
 *
 * @return \Illuminate\Database\Eloquent\Relations\HasMany
 */
public function messages()
{
  return $this->hasMany(Message::class);
}
```

Next, we need to define the inverse relationship by adding the code below to `Message` model:

```
// app/Message.php

/**
 * A message belong to a user
```

```
     *
     * @return \Illuminate\Database\Eloquent\Relations\BelongsTo
     */
    public function user()
    {
      return $this->belongsTo(User::class);
    }
```

Defining App Routes

Let's create the routes our chat app will need. Open `routes/web.php` and replace the routes with the code below to define three simple routes:

```
// routes/web.php

Auth::routes();

Route::get('/', 'ChatsController@index');
Route::get('messages', 'ChatsController@fetchMessages');
Route::post('messages', 'ChatsController@sendMessage');
```

The homepage will display chat messages and an input field to type new messages. A `GET messages` route will fetch all chat messages and a `POST messages` route will be used for sending new messages.

NOTE: Since we have removed the `/home` route, you might want to update the `redirectTo` property of both `app/Http/Controllers/Auth/LoginController.php` and `app/Http/Controllers/Auth/RegisterController.php` to:

```
protected $redirectTo = '/';
```

ChatsController

Now let's create the controller which will handle the logic of our chat app. Create a `ChatsController` with the command below:

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

Open the new create `app/Http/Controllers/ChatsController.php` file and add the

Open the new created `app/Http/Controllers/ChatsController.php` file and add the following code to it:

```
// app/Http/Controllers/ChatsController.php

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

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!'];
}
```

```
    }
```

Using the `auth` middleware in `ChatsController`'s `__contruct()` indicates that all the methods with the controller will only be accessible to authorized users. Then the `index()` will simply return a view file which we will create shortly. The `fetchMessages()` return a `JSON` of all messages along the their users. Lastly, the `sendMessage()` will persist the message into the database and return a status message.

Creating The Chat App View

For the chat app view, we'll be making use of Bootsnipp chat snippet with some few modifications.

Create a new `resources/views/chat.blade.php` file and paste into it:

```
<!-- resources/views/chat.blade.php -->

@extends('layouts.app')

@section('content')

<div class="container">
    <div class="row">
        <div class="col-md-8 col-md-offset-2">
            <div class="panel panel-default">
                <div class="panel-heading">Chats</div>

                <div class="panel-body">
                    <chat-messages :messages="messages"></chat-messages>
                </div>
                <div class="panel-footer">
                    <chat-form
                        v-on:messagesent="addMessage"
                        :user="{{ Auth::user() }}"
                    ></chat-form>
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Notice we have some custom tags with the `chat` view. these are `Vue` components

Notice we have some custom tags with the `chat` view, these are Vue components which we'll create soon. The `chat-messages` component will display our chat messages and the `chat-form` will provide an input field and a button to send the messages.

Before we go to create our `Vue` component, let's add the styles for the `chat` view. Open `resources/views/layouts/app.blade.php` (which was created when we ran `make:auth`) and add the code below just after the styles link:

```
<!-- resources/views/layouts/app.blade.php -->

<style>
  .chat {
    list-style: none;
    margin: 0;
    padding: 0;
  }

  .chat li {
    margin-bottom: 10px;
    padding-bottom: 5px;
    border-bottom: 1px dotted #B3A9A9;
  }

  .chat li .chat-body p {
    margin: 0;
    color: #777777;
  }

  .panel-body {
    overflow-y: scroll;
    height: 350px;
  }

  ::-webkit-scrollbar-track {
    -webkit-box-shadow: inset 0 0 6px rgba(0,0,0,0.3);
    background-color: #F5F5F5;
  }

  ::-webkit-scrollbar {
    width: 12px;
    background-color: #F5F5F5;
  }

  ::-webkit-scrollbar-thumb {
    -webkit-box-shadow: inset 0 0 6px rgba(0,0,0,.3);
    background-color: #555;
  }
```

```
</style>
```

Looking at the `resources/assets/js/bootstrap.js`, you will notice that Laravel has set up some of the front-end dependencies (jQuery, Bootstrap, Lodash, Vue, Axios, Echo) that are included out of the box. We can start using `Vue` without any further setup.

Create a new `ChatMessages.vue` file within `resources/assets/js/components` and paste the code below into it:

```
// resources/assets/js/components/ChatMessages.vue

<template>
    <ul class="chat">
        <li class="left clearfix" v-for="message in messages">
            <div class="chat-body clearfix">
                <div class="header">
                    <strong class="primary-font">
                        {{ message.user.name }}
                    </strong>
                </div>
                <p>
                    {{ message.message }}
                </p>
            </div>
        </li>
    </ul>
</template>

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

This component accepts an array of messages as `props`, loops through them and displays the name of the user who sent the message and the message body.

Next, create a new `ChatForm.vue` file within `resources/assets/js/components` and paste the code below into it:

```
// resources/assets/js/components/ChatForm.vue

<template>
```

```
<template>
    <div class="input-group">
        <input id="btn-input" type="text" name="message" class="form-control i

        <span class="input-group-btn">
            <button class="btn btn-primary btn-sm" id="btn-chat" @click="sendM
                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>
```

The `ChatForm` component displays an input field and a send button. It accepts the authenticated user as `props`. It also contains `newMessage` data which is bound to the input field. When the send button is clicked or the enter key is pressed on the input field, a `sendMessage()` is called. The `sendMessage()` simply triggers a `messagesent` event which passes along the message that was sent by the user to the root `Vue` instance (which will handle the actual sending of the message) and finally clear the input filed.

Next, we need to register our component in the root `Vue` instance. Open the `resources/assets/js/app.js` and update with code below:

```
// resources/assets/js/app.js

require('./bootstrap');

Vue.component('chat-messages', require('./components/ChatMessages.vue'));
Vue.component('chat-form', require('./components/ChatForm.vue'));

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.data);
            });
        }
    }
});
```

Once the `Vue` instance is created, using `Axios`, we make a `GET` request to the `messages` route and fetch all the messages then pass it to the messages array that will be displayed on the `chat` view. The `addMessage()` receives the message that was emitted from the `ChatForm` component, pushes it to the messages array and makes a `POST` request to the `messages` route with the message.

Broadcasting Message Sent Event

To add the realtime interactions to our chat app, we need to broadcast some kind of events based on some activities. In our case, we'll fire a `MessageSent` when a user

sends a message. First, we need to create an event, we'll call it `MessageSent`:

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

This will create a new `MessageSent` event class within the `app/Events` directory. This class must implement the `ShouldBroadcast` interface. The class should look like:

```
// app/Events/MessageSent.php

use App\User;
use App\Message;
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 MessageSent implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * User that sent the message
     *
     * @var User
     */
    public $user;

    /**
     * Message details
     *
     * @var Message
     */
    public $message;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct(User $user, Message $message)
    {
        $this->user = $user;
        $this->message = $message;
    }
```

```
    }

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

We defined two public properties that will be the data that will be passed along to the channel we are broadcasting to.

NOTE: These properties must be **public** for it to be passed along to the channel.

Since our chat app is an authenticated-only app, we create a private channel called `Chat`, which only authenticated users will be able to connect to. Using the `PrivateChannel` class, Laravel is smart enough to know that we are creating a private channel, so don't prefix the channel name with `private-` (as specified by Pusher), Laravel will add the `private-` prefix under the hood.

Next, we need to update the `sendMessage()` of `ChatsController` to broadcast the `MessageSent` event:

```
// app/Http/Controllers/ChatsController.php

//remember to use
use App\Events\MessageSent;

/**
 * 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')
  ]);
```

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

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

Since we created a private channel, only authenticated users will be able to listen on the `chat` channel. So, we need a way to authorize that the currently authenticated user can actually listen on the channel. This can be done by in the `routes/channels.php` file:

```
// routes/channels.php

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

We pass to the `channel()`, the name of our channel and a callback function that will either return `true` or `false` depending on whether the current user is authenticated.

Now when a message is sent, the `MessageSent` event will be broadcast to Pusher. We are using the `toOthers()` which allows us to exclude the current user from the broadcast's recipients.

Listening For Message Sent Event

Once the `MessageSent` event is broadcast, we need to listen for this event so we can update the chat messages with the newly sent message. We can do so by adding the code snippet below to `created()` of `resources/assets/js/app.js` just after `this.fetchMessages()`:

```
// resources/assets/js/app.js

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

We subscribe to the `chat` channel using Echo's `private()` since the channel is a private channel. Once subscribed, we listen for the `MessageSent` and based on this, update the chat messages array with the newly sent message.

Before testing out our chat app, we need to compile the JavaScript files using Laravel Mix using:

```
npm run dev
```

Now we can start our chat app by running:

```
php artisan serve
```

Our chat app is done as we can now send and receive messages in realtime.

Conclusion

You can see how straightforward it is to build a realtime app with Laravel and Pusher. With Pusher, you are not limited to chat apps, you can build any application that requires realtime interactivity. So, go create a free Pusher account and start building great applications!

PHP LARAVEL VUE.JS

CHANNELS

COMPANY

Careers

Blog

Press

OTHER

Contact sales

Terms & conditions

Security

CONNECT

Twitter

Medium

YouTube

LinkedIn

Github