Публикация на тему

Laravel Vue Socket чат

В статье рассматривается разработка чата на сокетах с бэкендом на Laravel, фронтендом на Vue. В частности, разбираем следующие темы: разработка бэкенд на Laravel, использование Vue для фронтенда, JWT-аутентификация, Laravel-echo сервер и база redis.

Анотация

-

-

Автор

Михалькевич Александр Викторович

Публикация

Наименование Laravel Vue Socket чат

Автор А.В.Михалькевич

Специальность В статье рассматривается разработка чата на сокетах с бэкендом на Laravel, фронтендом на Vue. В частности, разбираем следующие темы: разработка бэкенд на Laravel, использование Vue для фронтенда, JWT-аутентификация, Laravel-echo сервер и база redis.,

Анотация -

Anotation in English -

Ключевые слова laravel, echo-server, чат, vue, socket

Количество символов 17871

Содержание

Введение

1 Технические требования

Для понимания кода вам понадобятся следующие навыки:

• Основы PHP и Laravel;

• Умение создавать и работать с компаненами Vue;

• Понимание того, как работает Axios, отправлять и получаеть запросы по HTTP с помощью Axios HTTP client;

• Понимание принципов взаимодействия бэкенда и фронтенда.

• Понимание того, как работает JWT токен. • Умение запускать node-сервера.

• Умение подключаться к базе redis.

2 Бэкенд на Laravel

Сперва с помощью команды:

```
composer create-project laravel/laravel backend
```

Создадим новый проект. После - создадим базу данных chat и подключимся к ней в файле `.env`

В этом же файле пропишем необходимые настройки для подключения к базе данных Redis

```
BROADCAST_DRIVER=redis
QUEUE_CONNECTION=redis
QUEUE_DRIVER=sync
...
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

2 .1 Миграции для чатов и сообщений

Создадим нужные модели и миграции с помощью команд

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

и

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

Далее заполним миграции. Миграция CreateChatsTable:

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateChatsTable extends Migration
{
 public function up(){
  Schema::create('chats', function (Blueprint $table){
   $table->id();
   $table->string('name');
   $table->timestamp('created_at')->useCurrent();
   $table->timestamp('updated_at')->useCurrent();
  });
 }
 public function down(){
  Schema::dropIfExists('chats');
 }
}
```

И миграция CreateMessagesTable

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
class CreateMessagesTable extends Migration
{
 public function up(){
  Schema::create('messages', function (Blueprint$table) {
   $table->id();
   $table->unsignedBigInteger('user_id');
   $table->unsignedBigInteger('chat_id');
   $table->string('message');
   $table->timestamp('created_at')->useCurrent();
   $table->timestamp('updated_at')->useCurrent();
   $table->foreign('user_id')->references('id')->on('users');
   $table->foreign('chat_id')->references('id')->on('chats');
  });
 }
 public function down(){
  Schema::dropIfExists('messages');
 }
}
```

Далее запускаем команду

```
php artisan migrate
```

2 .2 Контроллер ChatController

Теперь можно приступить к созданию контроллеров приложения.

```
php artisan make:controller ChatController
```

Основная задача контроллера - взаимодействовать с моделью Chat, и выполнять основные операции по созданию, удалению, обновлению и выводу записей таблицы chats

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;
use App\Models\Chat;

class ChatController extends Controller
{
    public function get(Request $request)
    {
        return Chat::find($request->id);
    }

    public function getAll(Request $request)
    {
        return Chat::all();
    }

    public function create(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'name' => 'required|string|max:255',
        ]);

        if($validator->fails()){
            return response()->json($validator->errors()->toJson(), 400);
        }

        $chat = Chat::create([
            'name' => $request->get('name'),
        ]);

        return response()->json($chat, 201);
    }

    public function update(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'name' => 'required|string|max:255',
```

```
        ]);

        if($validator->fails()){
            return response()->json($validator->errors()->toJson(), 400);
        }

        $chat = Chat::find($request->id);
        $chat->name =  $request->get('name');
        $chat->save();

        return response()->json($chat);
    }

    public function delete(Request $request)
    {
        $chat = Chat::find($request->id);
        $chat->delete();
        return response(null, 200);
    }
}
```

2 .3 Контроллер MessageController

Создадим контроллер MessageController с помощью команды:

```
php artisan make:controller MessageController
```

Основная задача этого контроллера - создание и вывод сообщенй выбранного чата.

```
namespace App\Http\Controllers;

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

class MessageController extends Controller
{
    public function getAll(Request $request)
    {
        return Message::with(['user', 'chat'])
            ->where('chat_id', $request->chat_id)
            ->get();
    }

    public function create(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'message' => 'required|string|max:2500',
        ]);
```

```
        if($validator->fails()){
            return response()->json($validator->errors()->toJson(), 400);
        }

        $message = Message::create([
            'message' => $request->get('message'),
            'chat_id' => $request->get('chat_id'),
            'user_id' => $request->get('user_id')
        ]);

        $user = Auth::user();
        event(new MessageSent($user, $message));
        return response()->json($message, 201);
    }
}
```

Обратите внимание, что при создании сообщения (экшн create) вызывается событие MessageSent. Класс события необходимо реализовать, сделаем это чуть позже, после того, как создадим все нужные контроллеры и пропишем маршруты.

2 .4 JWT auth, настройка авторизации с помощью токенов

Сперва необходимо установить модуль JWT-auth

```
composer require tymon/jwt-auth
```

Далее опубликовать необходимые файлы:

```
php artisan vendor:publish --
provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

Далее создать секретный ключ для JWT

```
php artisan jwt:secret
```

Далее настроить конфигурацию для работы с JWT-токенон. Убедитесь в том, что в файле config/auth.php имеются следующие настройки:

```
return [
 'defaults' => [
  'guard' => 'api',
  'passwords' => 'users',
 ],
 'guards' => [
  'web' => [
   'driver' => 'session',
   'provider' => 'users',
  ],
  'api' => [
   'driver' => 'jwt',
```

```
   'provider' => 'users',
   ],
  ],
 'providers' => [
  'users' => [
   'driver' => 'eloquent',
   'model' => App\Models\User::class,
  ],
 ],
 'passwords' => [
  'users' => [
   'provider' => 'users',
   'table' => 'password_resets',
   'expire' => 60,
   'throttle' => 60,
  ],
 ],
 'password_timeout' => 10800,
];
```

2 .5 Кросс-доменные запросы и middleware для jwt-токенов

Приложения фронт и бэк будут находиться на разных доменах. Поэтому необходимо подключить модуль кроссдоменных сообщений, иначе браузер будет выдавать ошибку cors.

```
composer require fruitcake/laravel-cors
```

Далее, внесём неоходимые изменения в файл `/App/Http/Kernel.php`. В частности, в свойство `$middleware`:

```
    protected $middleware = [
        // \App\Http\Middleware\TrustHosts::class,
        \App\Http\Middleware\TrustProxies::class,
        \Fruitcake\Cors\HandleCors::class,
        \App\Http\Middleware\PreventRequestsDuringMaintenance::class,
        \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
        \App\Http\Middleware\TrimStrings::class,
\Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
        \Fruitcake\Cors\HandleCors::class,
    ];
```

и в свойство `$routeMiddleware`:

```
protected $routeMiddleware = [
        'auth' => \App\Http\Middleware\Authenticate::class,
        'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
        'cache.headers' =>
\Illuminate\Http\Middleware\SetCacheHeaders::class,
```

```
        'can' => \Illuminate\Auth\Middleware\Authorize::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'password.confirm' =>
\Illuminate\Auth\Middleware\RequirePassword::class,
        'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
        'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
        'verified' =>
\Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
        'jwt.verify' => \App\Http\Middleware\JwtMiddleware::class,
    ];
```

Теперь мы можем использовать `'jwt.verify'` или `JwtMiddleware` для защиты маршрутов.

2 .6 Модуль predis и дополнительные настройки для redis

Ещё необходимо установить модуль predis

```
composer require predis/predis
```

А в файле `config/database.php` должны быть прописаны следующие настройки для redis:

```
    'redis' => [

        'client' => env('REDIS_CLIENT', 'predis'),

        'options' => [
            'cluster' => env('REDIS_CLUSTER', 'redis'),
            'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME',
'laravel'), '_').'_database_'),
        ],

        'default' => [
            'url' => env('REDIS_URL'),
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_PORT', '6379'),
            'database' => env('REDIS_DB', '0'),
        ],

        'cache' => [
            'url' => env('REDIS_URL'),
            'host' => env('REDIS_HOST', '127.0.0.1'),
            'password' => env('REDIS_PASSWORD', null),
            'port' => env('REDIS_PORT', '6379'),
            'database' => env('REDIS_CACHE_DB', '1'),
        ],

    ],
```

2 .7 События real-time

Теперь, когда мы закончили с добавлением маршрутов, аутентификации и конфигурации базы данных и создали необходимые контроллеры, мы можем добавить код, позволяющий общаться между
фронтенд и бэкэнд в режиме реального времени. Во-первых, нам нужно создать класс события, чтобы мы могли вызывать функцию события для трансляции события, которое вызывается в MessageController.

Для этого мы запускаем команду:

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

Далее находим файл `Events/MessageSent.php` и заменяем его содержимое следующим кодом:

```
namespace App\Events;

use App\Models\User;
use App\Models\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 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 Channel('chat');
    }

    public function broadcastAs()
    {
        return 'MessageSent';
    }
}
```

2 .8 Маршрутизатор chanels.php

Для реализации broadcast маршрутов в файле `routes/chanels.php` пропишем следующее:

```
use Illuminate\Support\Facades\Broadcast;
Broadcast::channel('chat', function () {
 return true;
});
```

2 .9 Остальные маршруты приложения

Остальные и основные маршруты приложения пропишем в файле `routes/api.php`

```
Route::post('register', [UserController::class, 'register']);

Route::group([
    'middleware' => 'api',
    'prefix' => 'auth'
], function () {
    Route::post('login', [AuthController::class, 'login']);
    Route::post('logout', [AuthController::class, 'logout']);
    Route::post('refresh', [AuthController::class, 'refresh']);
    Route::post('me', [AuthController::class, 'me']);
});

Route::group([
    'middleware' => ['api', 'jwt.verify'],
```

```
    'prefix' => 'chat'
], function () {
    Route::get('{id}', [ChatController::class, 'get']);
    Route::get('', [ChatController::class, 'getAll']);
    Route::post('create', [ChatController::class, 'create']);
    Route::put('update/{id}', [ChatController::class, 'update']);
    Route::delete('delete/{id}', [ChatController::class, 'delete']);
});

Route::group([
    'middleware' => ['api', 'jwt.verify'],
    'prefix' => 'message'
], function () {
    Route::get('{chat_id}', [MessageController::class, 'getAll']);
    Route::post('create', [MessageController::class, 'create']);
});
```

2 .10 Запуск сервера бэкенда

Сперва необходимо перейти в папку backend.

И из этой папки запускаем laravel-сервер

```
php artisan serve
```

3 Laravel echo server

Сперва создадим папку laravel-echo-server на уровне с папкой backend.

В этой папке будет находиться нужный конфиг, и запускать laravel-echo-server будет отсюда.

3 .1 Установка Redis и laravel-echo-server

Теперь нам пригодится последняя актуальная версия базы Redis.

Если у вас уже установлен Redis, то сразу можете переходить к настройке файла laravel-echo-server.json

```
sudo apt update
sudo apt install redis-server
```

После этого можно установить laravel-echo-server. Делать это лучше глобально:

```
npm i laravel-echo-server
```

3 .2 laravel-echo-server.json

В папке laravel-echo-server необходимо создать файл laravel-echo-server.json со следующим содержимым:

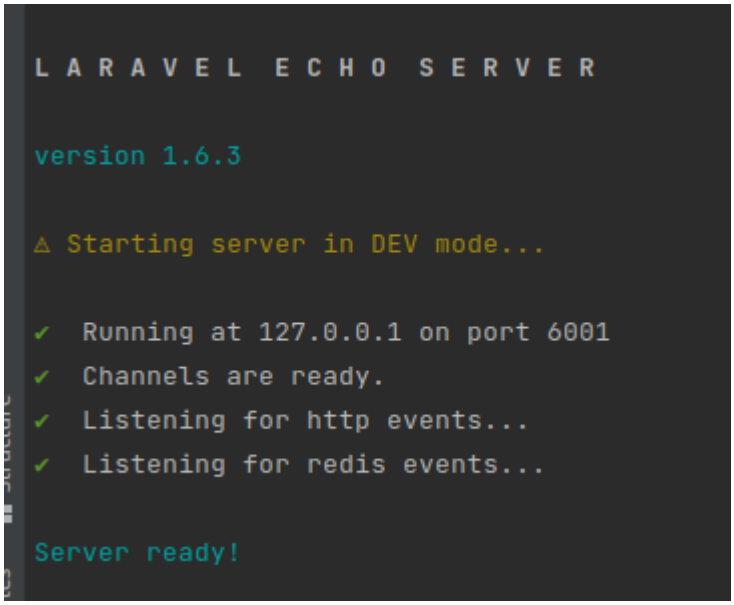
```
{
        "authHost": "http://localhost:8000",
        "authEndpoint": "/broadcasting/auth",
        "clients": [
                {
                        "appId": "APP_ID",
                        "key": "c84077a4dabd8ab2a60e51b051c9d0ea"
                }
        ],
        "database": "redis",
        "databaseConfig": {
                "redis": {
                        "port": "6379",
                        "host": "localhost"
                },
                "sqlite": {
                        "databasePath": "/database/laravel-echo-
server.sqlite"
                },
                "publishPresence": true
        },
        "devMode": true,
        "host": "127.0.0.1",
        "port": "6001",
        "protocol": "http",
        "socketio": {},
        "secureOptions": 67108864,
        "sslCertPath": "",
        "sslKeyPath": "",
        "sslCertChainPath": "",
        "sslPassphrase": "",
        "subscribers": {
                "http": true,
                "redis": true
        },
        "apiOriginAllow": {
                "allowCors": true,
                "allowOrigin": "*",
                "allowMethods": "GET, POST",
                "allowHeaders": "Origin, Content-Type, X-Auth-Token, X-
Requested-With, Accept, Authorization, X-CSRF-TOKEN, X-Socket-Id"
        }
}
```

3 .3 Запуск laravel-echo-server

Для запуска можно воспользоваться командой:

```
laravel-echo-server start
```

Такой ответ консоли свидетельствует об успешном запуске сервера



4 Фронтенд на Vue

ыв

Заключение

Список использованных источников

Приложения