Laravel Event

Laravel Event là gì?

Một sự kiện trong máy tính là hành động hay một cái gì đó xảy ra tại một thời điểm xác định bởi một chương trình máy tính và có thể quản lý bằng các chương trình máy tính. Một vài ví dụ về sự kiện trong Laravel:

- Một người dùng đăng ký mới.
- Một bài viết được đăng.
- Một thành viên viết lời bình luận.
- Người dùng like một bức ảnh.
- Còn rất nhiều tình huống khác sử dụng Event

Tại sao sử dụng Laravel Event?

Có một số lý do cơ bản làm chúng ta thấy được Laravel Event có ý nghĩa như thế nào:

- Laravel Event giúp chúng ta tách biệt các thành phần trong ứng dụng, khi một sự kiện xảy ra, sự kiện không cần biết những gì khác ngoài việc thực thi công việc của nó.
- Không có Laravel Event sẽ phải đặt rất nhiều các code logic ở một chỗ
- Laravel Event giúp tăng tốc xử lý dữ liệu

Case Study sử dụng Laravel Event

Case Study 01

Một bài viết mới được đăng sẽ gửi mail đến tất cả các thành viên, việc gửi mail đến hàng triệu thành viên sẽ mất khá nhiều thời gian, với Laravel event gửi email sẽ là một trigger khi một bài viết mới được tạo ra.

Với cách viết code cũ trước đây, tất cả xử lý đưa vào tính năng đăng bài mới khiến việc đăng một bài mất rất nhiều thời gian

Case Study 02

Khi thanh toán đơn hàng, phải gửi email thông báo đến khách hàng, việc thanh toán đơn hàng là ưu tiên nhất và cần thời gian xử lý cực nhanh đảm bảo trải nghiệm tốt cho khách hàng, nếu email khách hàng nhận hiện đang có vấn đề, xử lý gửi mail sẽ vào một vòng lặp thử liên tục gửi, như vậy việc thanh toán sẽ rất lâu.

Với Laravel Event mọi việc được tách biệt rất rõ và tăng tốc trong xử lý dữ liệu ứng dụng

Làm việc với Laravel Event

- Laravel Event cho phép bạn đăng ký **Event** và tạo các **Listener** cho rất nhiều các sự kiện xảy ra trong ứng dụng
- Các class Event được lưu trữ trong `app/Events` và các listener được lưu trong `app/Listeners`
- Một Event có thể có rất nhiều Listerner được tạo ra và các Listener này độc lập với nhau

Bước 1: Định nghĩa Event

Event được tạo ra đơn giản bằng cách sử dụng câu lệnh artisan

```
php artisan make:event OrderPayment
```

Nó sẽ tạo ra một class Event nằm trong `app\Events` như sau:

```
<?php

namespace App\Events;

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 OrderPayment
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

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

Bước 2: Tạo Event Listener

Ngay khi một event xảy ra, ứng dụng cần biết để thực hiện các việc khác, đó chính là lý do cần Event Listerner. Để tạo Listerner đơn giản là sử dung lệnh artisan:

```
php artisan make:listener SendEmailAfterOrderPayment --event="OrderPayment"
```

Khi đó, sẽ tạo ra class `SendEmailAfterOrderPayment.php` trong `app\Listerners` với nội dung:

```
<?php

namespace App\Listeners;

use App\Events\OrderPayment;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendEmailAfterOrderPayment
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param  OrderPayment  $event
     * @return void
     */
    public function handle(OrderPayment $event)
    {
        //
    }
}
```

Phương thức `handle()` với tham số đầu vào là một instance của Event mà Listener được gán vào, phương thức này sẽ được dùng để thực hiện các công việc cần thiết để đáp lại Event. Instance đầu vào của phương thức cũng chứa cả các giá trị mà Event truyền sang

```
/**
 * Handle the event.
 *
 * @param  OrderPayment  $event
 * @return void
 */
public function handle(OrderPayment $event)
{
    //
    $orderAmout = $event->order->amount;
   ...
}
```

Đôi khi bạn muốn dừng quảng bá Event này đến các Listener khác, rất đơn giản bạn trả về giá trị false trong phương thức `handle()` của Listerner đang xử lý

Bước 3: Đăng ký Event

Trước khi đến các bước xử lý trong Event, chúng ta cần đăng ký Event, mở file `app/Providers/EventServiceProvider.php` và tìm đến thuộc tính `$listen`.

```
protected $listen = [
    'App\Events\OrderPayment' => [
        'App\Listeners\SendEmailAfterOrderPayment',
    ],
];
```

Chúng ta đã khai báo Listerner này của Event nào, một Event có thể có nhiều các Listener khác nhau

```
protected $listen = [
    'App\Events\OrderPayment' => [
        'App\Listeners\SendEmailAfterOrderPayment',
        'App\Listeners\SendSMSAfterOrderPayment',
    ],
];
```

Ba bước trên có thể được thực hiện đơn giản hơn bằng cách gán Listener cho Event trong file `app/Providers/EventServiceProvider.php` sau đó chạy lệnh

```
php artisan event:generate
```

Khi đó, các class trong `app\Events` và `app\Listeners` sẽ được tự động sinh ra.

Bước 4: Tạo ra thông báo sự kiện xảy ra

Khi thực hiện một đoạn code nào đó, muốn thông báo đến hệ thống là sự kiện xảy ra để các listener có thể biết được chúng ta sử dụng phương thức `dispatch()`.

```
use App\Events\OrderPayment;

// Các xử lý thanh toán đơn hàng ở đây
$order = new Order;
$order->amount = 2000000;
$order->note = 'Tạo đơn hàng mẫu';
...
$order->save();

OrderPayment::dispatch($order)
```

Model Events

Trong quá trình hoạt động của mình, mỗi Eloquent Model có thể tạo ra nhiều sự kiện khác nhau, cho phép chúng ta thao tác với những thời điểm khác nhau trong chu kỳ hoạt động của model

đó. Các phương thức tương ứng với các sự kiện đó là: `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring` và `restored`.

Tên của các phương thức trên cho chúng ta biết sự kiện nào sẽ được thực hiện và được thực hiện tại thời điểm nào. Các sự kiện `creating` và `created` được thực hiện khi một model được lưu lại trong cơ sở dữ liệu lần đầu tiên, nếu model đã tồn tại và phương thức `save` được gọi trên model đó hai sự kiện `updating` và `updated` sẽ được thực hiện. Khi một model được xóa bằng phương pháp Soft Delete và được khôi phục lại qua phương thức `restore` hai event `restoring` và `restored` sẽ được thực hiện.

Bài toán thực tế

Bây giờ chúng ta muốn xóa toàn bộ những bài viết của một người dùng khi người dùng đó bị xóa khỏi hệ thống. Chúng ta sẽ thực hiện thao tác đó sử dụng sự kiện `deleting` của model `User`. Có rất nhiều cách để thực hiện thao tác trên này, cách đơn giản nhất là sử dụng phương thức `deleting` của model User trong một `ServiceProvider`

```
<?php

namespace App\Providers;

use App\Models\User;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        User::deleting(function ($user) {
            $user->posts()->delete();
        });
    }
}
```

Ngoài ra, chúng ta có thể sử dụng Event và Listener riêng để xử lý bằng cách khai báo thuộc tính `$dispatchesEvents` trong Model

```
protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
```

Model Observer

Để tạo Model Observer, bạn sử dụng câu lệnh sau:

```
php artisan make:observer UserObserver --model=User
```

```
<?php
```

```
namespace App\Models\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Hook into user deleting event.
     *
     * @param User $user
     * @return void
     */
    public function deleting(User $user)
    {
        $user->posts()->delete();
    }

    public function created(User $user)
    {
        //
    }

    public function restoring(User $user)
    {
        //
    }
}
```

Để đăng ký **Observer Model**, bạn mở file `EventServiceProvider.php`

```
<?php

namespace App\Providers;

use App\Models\User;
use App\Models\Observers\UserObserver;
use Illuminate\Contracts\Events\Dispatcher as DispatcherContract;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;

class EventServiceProvider extends ServiceProvider
{
    /**
     * Register any other events for your application.
     *
     * @param  \Illuminate\Contracts\Events\Dispatcher $events
     * @return void
     */
    public function boot(DispatcherContract $events)
    {
        User::observe(new UserObserver);
    }
}
```