Exploring Laravel Events

### Introduction

**[Introduction Slide]**

Hello my name is Lee Crosdale,

A bit on my background:

At the start of my career I was a Microsoft Dynamics AX Developer, building SSRS reports and making various modifications to the AX codebase, I also built an Android app (which as far as I’m aware is still in use).

I then moved into Web Development for a company named ParcelBroker (Magpie Solutions), where I was the IT Manager / Developer on the project, I spent a lot of time refactoring a legacy PHP site into Laravel, as well as building some internal tools, both with Laravel and some other languages, Node, Python etc.

I’m currently the Lead Laravel and PHP Developer for Enovate (Point to boss if he is in the room), where I work on various Laravel and CraftCMS projects.

You can find me on twitter **@leecrosdale**.

I also stream on twitch: **twitch.tv/crosdale**

**[Introductions]**

Do intros

**[Intro Talk Slide]**

My talk is about exploring Laravel Events.

**[Agenda]**

(Just read this off the slide)

My plan is to go through some basics, and then onto some live coding (argh) a small Laravel Events project, so that you can all see how easy they are to use, everything will likely break, so on this other tab I also have Google ready to go.

I’ve also got a cool surprise at the end (LaravelBR)

**[Let’s get started]**

(Take time for gif)

**[What Laravel docs define as an ‘Event’]**

The Laravel docs define events as the following:

*“Laravel's events provide a simple observer implementation, allowing you to subscribe and listen for various events that occur in your application. Event classes are typically stored in the app/Events directory, while their listeners are stored in app/Listeners. Don't worry if you don't see these directories in your application, since they will be created for you as you generate events and listeners using Artisan console commands.*

*Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other. For example, you may wish to send a Slack notification to your user each time an order has shipped. Instead of coupling your order processing code to your Slack notification code, you can raise an OrderShipped event, which a listener can receive and transform into a Slack notification.”*

Here is an example that would sit inside the EventServiceProvider (which is essentially where Laravel will look for and load each event / listener)

**[Event Basics]**

The way that you create a new event is Laravel is super simple, using the artisan command simple enter php artisan make:event {eventName} in this case, OrderShipped

You can also automatically create events and listeners (more listeners that in a second) by specifying them in the EventServiceProvider, but to be honest, I’ve found it a bit weird since I use PHPStorm and there is nothing to ‘auto import’

You can also wildcard events into the EventServiceProvider, but this isn’t something I’ve used before.

I usually just keep it simple and build piece by piece, rather than do everything at once.

**[An example event]**

This event simply takes an eloquent model, and then serializes it for later use.

Events can also be broadcasted to a websocket server, which is something I’ll cover later on.

**[Listeners]**

Listeners do exactly what they sound like they do, they listen, specifically for events.

We can also have multiple listeners per event, for example {read from slide}

Listeners are queueable, so rather than execute it during the request, we can put it in a queue, and the server will process it.

E.g We want to send an email to the user, but connecting to our external mail service is slow, instead of making the user wait, we will queue this listener, and run it on our server instead.

**[Example Listener]**

Here you can see that inside the listener, we have access to the event, and subsequently the ‘order’ eloquent model that we passed in earlier.

Something else you can do with events are ‘delay’ them, for example “run this listener 60 seconds after the event fires”

And maybe you also only want to reward a customer if they spend over a certain amount of money, so you would only queue that listener to run based on the logic below.

**[Auto Discovery via PHP Reflection]**

In Laravel 5.8.9 onwards, automatic discovery exists, but you have to enable it in the service provider

Basically Laravel will match the listener for the ‘name’ of your event using reflection, again, I like specifying these myself so that *something* is written out rather than relying on ‘magic’ for it to be picked up.

**[Broadcasting]**

In many modern web applications, WebSockets are used to implement realtime, live-updating user interfaces. When some data is updated on the server, a message is usually sent over a WebSocket connection to be handled by the client. This is better than periodically polling, or long-polling requests.

The package that we use for handling websockets on the client side is called ‘Laravel Echo’

There are also a few services that provide servers, which are

* Pusher,
* Redis,
* SocketIO,
* Laravel-Websockets

The one I will use in my example is Laravel Websockets, which essentially shares the same driver as Pusher

**[Example Broadcasting Code]**

An event implements ‘Should Broadcast’

This will send the event and it’s data off to the websockets server when we fire the event

**[Laravel Websockets Setup]**

pa websockets:serve

Goto laravel-websockets url

**[Quick Recap]**

So we have covered the basics of events, listeners and broadcasting

The TLDR; is

An event will fire (and also broadcast if that is set), a listener will listen for that event, and then perform {x} action

**[A food ordering Example]**

**[Client Brief]**

A client owns a food - takeaway ordering website, they would like to inform customers about the status of their order, from Receiving the order until it is out for delivery.

When a customer places an order, they would also like to send them a 20% discount on their next order, if they spend over £10.

//And finally, 20 minutes after delivery, they would like to ask the customer to rate the food.

**[Switch to PHPStorm]**

So currently the client has a very static site, the customer can goto the (very poor) shop and order some food at the click of a button.

When the status changes, they have to refresh the page, and there are no further notifications.

We will first set up the websockets server.

**composer require beyondcode/laravel-websockets**

Then we will set the env file BROADCAST\_DRIVER to use ‘PUSHER’ since laravel-websockets works with this driver.

For the PUSHER\_APP\_ID etc, use whatever you like for local dev.

We also need to edit the config/broadcasting.php file

'pusher' => [

 'driver' => 'pusher',

 'key' => env('PUSHER\_APP\_KEY'),

 'secret' => env('PUSHER\_APP\_SECRET'),

 'app\_id' => env('PUSHER\_APP\_ID'),

 'options' => [

 'cluster' => env('PUSHER\_APP\_CLUSTER'),

 'encrypted' => *true*,

 'host' => '127.0.0.1',

 'port' => 6001,

 'scheme' => 'http'

 ],

],

Then we can start the websockets server.

**php artisan websockets:serve**

**Goto** [**http://127.0.0.1:8000/laravel-websockets**](http://127.0.0.1:8000/laravel-websockets)

We can then goto the websockets dashboard which will give us a nice ui for all the events that come through, and you can see some already have happened, this is just basic stuff to do with the laravel-websockets package.

Now that is set up, we have to get the client to join the websockets server.

**npm install --save laravel-echo pusher-js** and we also have to install laravel-echo

**Boostrap.js - uncomment bottom of file**

And in order to use laravel echo with laravel websockets, we have to edit a few things

**window.Echo = *new* Echo({**

 **broadcaster: 'pusher',**

 **key: process.env.MIX\_PUSHER\_APP\_KEY,**

 **wsHost: window.location.hostname,**

 **wsPort: 6001,**

 **disableStats: *true***

**});**

It’s important that after this step we restart npm run watch

If I refresh the page now, you will see the client connecting to the server. (show websockets dashboard or terminal).

Now we can create the OrderStatus event

Pa make:event OrderStatus

We can then go into the event and make it broadcastable.

Implements shouldbroadcast

We should also pass the order model to the event

We can also specify the channel that it should broadcast to, usually you would broadcast these to private channels, I’ll go over that if we get time, but for now we will just use a normal channel.

We will broadcast to order.orderid

I will now quickly code the ‘Change order status’ part of the admin panel

<form *method*="post" *action*="{{ route('update', $order) }}">

 {{ csrf\_field() }}

<select *name*="status">

 <option>Accepted</option>

 <option>Cooking</option>

 <option>Out for delivery</option>

 <option>Delivered</option>

</select>

 <button *type*="submit">Update</button>

</form>

Add the route

Route::*post*('order/{order}', 'OrderController@update')->name('update')->middleware('auth');

Add the update function to the OrderController

*public function* update(Request $request, Order $order)

{

 $order->status = $request->status;

 $order->save();

 event(*new* OrderStatus($order));

}

Cool.

So if I know update an order, we should see the event being sent through to the websockets server.

And there we go.

Now we need to get the client to join the correct room, so that their front end will update.

We will move the order page into vuejs

<order-component *:order*="{{$order}}"></order-component>

In order.blade.php

We will create the component, and then turn on auto discovery for vue components.

We will then ask Echo to join the order channel.

Echo.channel(`order.${*this*.order.id}`)

 .listen('OrderStatus', (e) => {

 console.log(e.order);

 });

And we will display the same data in the component as the blade had before.

<order-component *:new*="{{$order}}"></order-component>

**<script>**

 ***export default* {**

 **mounted() {**

 **Echo.channel(`order.${*this*.order.id}`)**

 **.listen('OrderStatus', (e) => {**

 **console.log(e.order);**

 ***this*.order = e.order;**

 **});**

 **},**

 **data() {**

 ***return* {**

 **order: { ...*this*.new}**

 **}**

 **},**

 **props: ['new'],**

 **}**

**</script>**

**And now, if I seperate the screen, you will see that the order status will update on the page, as we change it in the admin panel.**

**--------------------OLD NOT IN TALK---------------------------------**

**[Customer Case]**

Customer Brief:

A customer has an existing Tic-Tac-Toe game, this is currently limited to a single, 2 player game. (One user is X the other is O, and only 1 game can be played at a time).

The customer would like to add more team based elements, the main one being a ‘Voting’ system, that will allow 1 or more people in each team, to ‘Vote’ for the next Square that is filled out.

**Switch to PHPStorm**

Ok, I’ve already created a Laravel 6 project, connected the database etc.

I’ve also entered my Pusher/Websocket details (more on that later) into my .env file.

*Top tip: When you change .env, the mix\_ will pass those along to laravel mix (npm) so you can access them in javascript, but make sure you re-run npm run watch or they won’t get picked up!*

Some of the project is pre-built and set up.

* Go over the existing project
* 3x3 square generated with Canvas
* Show new game event resets screen
* Show webhook that will place an X in the specified location
* Show team ‘X’ wins screen

The event that we will need is a ‘Vote’ event, to tell everyone else when someone has voted.

I can create those quickly here:

Create new event: `pa make:event Vote`

I also will need an endpoint for the votes to go

Create new endpoint /play/vote - POST VoteController@vote

We also need to store those votes, with a Model

Make:model Vote - make this guarded, copy the create\_picks into create\_votes as they will be the same.

A game now also hasMany votes

*public function votes()*

*{*

*return $this->hasMany(Vote::class)->where('turn', $this->turn\_number);*

*}*

*protected* $with = ['picks','votes'];

We also need to make sure that after a certain time frame, the game will automatically ‘tick’ the game over to the next turn.

Create Console Command, Create listener for Vote event

// Code the part that will take the vote, and then broadcast it out.

We will take the first bit from the existing game, the Pick::create function will become vote as we want to store the vote that the user made.

And then we want to fire our new Vote event, and pass in the location of the vote so we can update the users screens.

event(*new* Vote([$request->col, $request->row, $request->team]));

We will also create a ‘next turn listener’ that will check the game status after the next turn is fired,

Make:listener CheckGame

And then we will register that in the event service provider.

Next we will take the existing game logic and place that into the ‘next turn listener’

And then add the decide vote function

$votes = $game->votes()->where('team', $game->current\_team)->where('turn\_number', $game->turn\_number)->get();

public function handle(NextTurn $event)

 {

 $game = $event->game;

 // Decide on vote

 $this->decideVote($game);

 $game = Game::first();

 $status = $this->checkGame($game);

 dump($status);

 ++$game->turn\_number;

 if ($status === 'win') {

 event(new Win());

 } else if ($status === 'draw') {

 event(new Draw());

 }

 return response(['status' => $game->save()]);

 }

 private function decideVote(Game $game)

 {

 $votes = DB::table('votes')->select(['row','col',DB::raw('COUNT(id) as vote\_count')])->where('game\_id', $game->id)->where('turn', $game->turn\_number)->groupBy(['row','col'])->get();

 $highest\_vote = 0;

 $winning\_vote = null;

 foreach($votes as $vote) {

 if ($vote->vote\_count > $highest\_vote) {

 $highest\_vote = $vote->vote\_count;

 $winning\_vote = $vote;

 }

 }

 if ($winning\_vote) {

 Pick::create([

 'game\_id' => $game->id,

 'team' => $game->current\_team,

 'col' => $winning\_vote->col,

 'row' => $winning\_vote->row,

 'turn' => $game->turn\_number

 ]);

 }

 }

 private function checkGame(Game $game)

 {

 $picks = $game->picks;

 // Build picks lines

 $picksLines = [];

 foreach ($picks as $pick) {

 $picksLines[$pick->team][$pick->row][$pick->col] = $pick->team;

 }

 dump($picksLines);

 dump($game->current\_team);

 return $this->checkStatus($picksLines, $game->current\_team);

 }

 private function checkStatus($currentLines, $team)

 {

 if (isset($currentLines[$team])) {

 // Horizontal

 if (isset($currentLines[$team][0][0]) && isset($currentLines[$team][0][1]) && isset($currentLines[$team][0][2])) return 'win';

 if (isset($currentLines[$team][1][0]) && isset($currentLines[$team][1][1]) && isset($currentLines[$team][1][2])) return 'win';

 if (isset($currentLines[$team][2][0]) && isset($currentLines[$team][2][1]) && isset($currentLines[$team][2][2])) return 'win';

 // Vertical

 if (isset($currentLines[$team][0][0]) && isset($currentLines[$team][1][0]) && isset($currentLines[$team][2][0])) return 'win';

 if (isset($currentLines[$team][0][1]) && isset($currentLines[$team][1][1]) && isset($currentLines[$team][2][1])) return 'win';

 if (isset($currentLines[$team][0][2]) && isset($currentLines[$team][1][2]) && isset($currentLines[$team][2][2])) return 'win';

 // Diagonal

 if (isset($currentLines[$team][0][0]) && isset($currentLines[$team][1][1]) && isset($currentLines[$team][2][2])) return 'win';

 if (isset($currentLines[$team][0][2]) && isset($currentLines[$team][1][1]) && isset($currentLines[$team][2][0])) return 'win';

 }

 if (Pick::all()->count() === 9) {

 return 'draw';

 }

 return 'continue';

 }

I have to update the Game update function so we can show the votes to the users

*public function* update()

{

 $game = Game::*first*();

 *return* response(['game' => $game, 'votes' => $game->votes()->where('turn', $game->turn\_number)->get()]);

}

We can get rid of the pick function on the GameController now.

Now we can change the api endpoint on the GameComponent from pick to vote

axios.get('update').then((response) => {

 *this*.game = response.data.game;

 *this*.votes = response.data.votes;

 *this*.render();

});

I’ve also got some JS ready as I didn’t want to live code this part, since it’s more to do with Canvas than events / broadcasting

*let* count = 0;

*for* (*let* k *in this*.votes) {

 *let* vote = *this*.votes[k];

 *if* (vote.col === x && vote.row === y) {

 count++;

 }

}

ctx.fillText(count, x \* *this*.provider.tileW + (*this*.provider.tileW / 2) , y \* *this*.provider.tileH + (*this*.provider.tileH / 2));

We can now update the kernel and run the console ForceTurn command.

*protected* $commands = [

 ForceTurn::*class*

];

**[Go back to slide]**

{read slide}

**[End of speech - Here is what I’ve done with that knowledge - EventsBR]**