# Events

## # Introduction

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.

## # Registering Events & Listeners

The `EventServiceProvider` included with your Laravel application provides a convenient place to register all of your application's event listeners. The `listen` property contains an array of all events (keys) and their listeners (values). You may add as many events to this array as your application requires. For example, let's add a `OrderShipped` event:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'App\Events\OrderShipped' => [
        'App\Listeners\SendShipmentNotification',
    ],
];
```

### # Generating Events & Listeners

Of course, manually creating the files for each event and listener is cumbersome. Instead, add listeners and events to your `EventServiceProvider` and use the `event:generate` command. This command will generate any events or listeners that are listed in your `EventServiceProvider`. Events and listeners that already exist will be left untouched:

```
php artisan event:generate
```

### # Manually Registering Events

Typically, events should be registered via the `EventServiceProvider` `$listen` array; however, you may also register Closure based events manually in the `boot` method of your `EventServiceProvider`:

```
/**
 * Register any other events for your application.
 *
 * @return void
 */
```

```
public function boot()
{
    parent::boot();

    Event::listen('event.name', function ($foo, $bar) {
        //
    });
}
```

### Wildcard Event Listeners

You may even register listeners using the `*` as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the event name as their first argument, and the entire event data array as their second argument:

```
Event::listen('event.*', function ($eventName, array $data) {
    //
});
```

## # Event Discovery

! Event Discovery is available for Laravel 5.8.9 or later.

Instead of registering events and listeners manually in the `$listen` array of the `EventServiceProvider`, you can enable automatic event discovery. When event discovery is enabled, Laravel will automatically find and register your events and listeners by scanning your application's `Listeners` directory. In addition, any explicitly defined events listed in the `EventServiceProvider` will still be registered.

Laravel finds event listeners by scanning the listener classes using reflection. When Laravel finds any listener class method that begins with `handle`, Laravel will register those methods as event listeners for the event that is type-hinted in the method's signature:

```
use App\Events\PodcastProcessed;

class SendPodcastProcessedNotification
{
    /**
     * Handle the given event.
     *
     * @param  \App\Events\PodcastProcessed
     * @return void
     */
    public function handle(PodcastProcessed $event)
    {
        //
    }
}
```

Event discovery is disabled by default, but you can enable it by overriding the `shouldDiscoverEvents` method of your application's `EventServiceProvider`:

```
/**
 * Determine if events and listeners should be automatically discovered.
 *
 * @return bool
 */
public function shouldDiscoverEvents()
{
    return true;
}
```

By default, all listeners within your application's Listeners directory will be scanned. If you would like to define additional directories to scan, you may override the `discoverEventsWithin` method in your `EventServiceProvider`:

```
/**
 * Get the listener directories that should be used to discover events.
 *
 * @return array
 */
protected function discoverEventsWithin()
{
    return [
        $this->app->path('Listeners'),
    ];
}
```

In production, you likely do not want the framework to scan all of your listeners on every request. Therefore, during your deployment process, you should run the `event:cache` Artisan command to cache a manifest of all of your application's events and listeners. This manifest will be used by the framework to speed up the event registration process. The `event:clear` command may be used to destroy the cache.

The `event:list` command may be used to display a list of all events and listeners registered by your application.

## # Defining Events

An event class is a data container which holds the information related to the event. For example, let's assume our generated `OrderShipped` event receives an Eloquent ORM object:

```php
<?php

namespace App\Events;

use App\Order;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use SerializesModels;

    public $order;

    /**
     * Create a new event instance.
     *
     * @param  \App\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
}
```

As you can see, this event class contains no logic. It is a container for the `Order` instance that was purchased. The `SerializesModels` trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's `serialize` function.

## # Defining Listeners

Next, let's take a look at the listener for our example event. Event listeners receive the event instance in their `handle` method. The `event:generate` command will automatically import the proper event class and type-hint the event on the `handle` method. Within the `handle` method, you may perform any actions necessary to respond to the event:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

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

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        // Access the order using $event->order...
    }
}
```

Your event listeners may also type-hint any dependencies they need on their constructors. All event listeners are resolved via the Laravel service container, so dependencies will be injected automatically.

#### Stopping The Propagation Of An Event

Sometimes, you may wish to stop the propagation of an event to other listeners. You may do so by returning `false` from your listener's `handle` method.

## # Queued Event Listeners

Queueing listeners can be beneficial if your listener is going to perform a slow task such as sending an e-mail or making an HTTP request. Before getting started with queued listeners, make sure to configure your queue and start a queue listener on your server or local development environment.

To specify that a listener should be queued, add the `ShouldQueue` interface to the listener class. Listeners generated by the `event:generate` Artisan command already have this interface imported into the current namespace, so you can use it immediately:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    //
}
```

That's it! Now, when this listener is called for an event, it will be automatically queued by the event dispatcher using Laravel's queue system. If no exceptions are thrown when the listener is executed by the queue, the queued job will automatically be deleted after it has finished processing.

#### Customizing The Queue Connection & Queue Name

If you would like to customize the queue connection, queue name, or queue delay time of an event listener, you may define the `$connection`, `$queue`, or `$delay` properties on your listener class:

```php
<?php

namespace App\Listeners;

use App\Events\OrderShipped;
use Illuminate\Contracts\Queue\ShouldQueue;

class SendShipmentNotification implements ShouldQueue
{
    /**
     * The name of the connection the job should be sent to.
     *
     * @var string|null
     */
    public $connection = 'sqs';

    /**
     * The name of the queue the job should be sent to.
     *
     * @var string|null
     */
    public $queue = 'listeners';

    /**
     * The time (seconds) before the job should be processed.
     *
     * @var int
     */
    public $delay = 60;
}
```

#### Conditionally Queueing Listeners

Sometimes, you may need to determine whether a listener should be queued based on some data that's only available at runtime. To accomplish this, a `shouldQueue` method may be added to a listener to determine whether the listener should be queued and executed synchronously:

```php
<?php

namespace App\Listeners;

use App\Events\OrderPlaced;
use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue
{
    /**
     * Reward a gift card to the customer.
     *
     * @param  \App\Events\OrderPlaced  $event
     * @return void
     */
    public function handle(OrderPlaced $event)
    {
        //
    }

    /**
     * Determine whether the listener should be queued.
     *
     * @param  \App\Events\OrderPlaced  $event
     * @return bool
     */
    public function shouldQueue(OrderPlaced $event)
    {
        return $event->order->subtotal >= 5000;
    }
}
```

## # Manually Accessing The Queue

If you need to manually access the listener's underlying queue job's `delete` and `release` methods, you may do so using the `Illuminate\Queue\InteractsWithQueue` trait. This trait is imported by default on generated listeners and provides access to these methods:

```php
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;

    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        if (true) {
            $this->release(30);
        }
    }
}
```

## # Handling Failed Jobs

Sometimes your queued event listeners may fail. If queued listener exceeds the maximum number of attempts as defined by your queue worker, the `failed` method will be called on your listener. The `failed` method receives the event instance and the exception that caused the failure:

```php
<?php

namespace App\Listeners;

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

class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;
    /**
     * Handle the event.
     *
     * @param  \App\Events\OrderShipped  $event
     * @return void
     */
    public function handle(OrderShipped $event)
    {
        //
    }

    /**
     * Handle a job failure.
     *
     * @param  \App\Events\OrderShipped  $event
     * @param  \Exception  $exception
     * @return void
     */
    public function failed(OrderShipped $event, $exception)
    {
        //
    }
}
```

## # Dispatching Events

To dispatch an event, you may pass an instance of the event to the `event` helper. The helper will dispatch the event to all of its registered listeners. Since the `event` helper is globally available, you may call it from anywhere in your application:

```
<?php

namespace App\Http\Controllers;

use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
use App\Order;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  int  $orderId
     * @return Response
     */
    public function ship($orderId)
    {
        $order = Order::findOrFail($orderId);

        // Order shipment logic...

        event(new OrderShipped($order));
    }
}
```

When testing, it can be helpful to assert that certain events were dispatched without actually triggering their listeners. Laravel's built-in testing helpers makes it a cinch.

## # Event Subscribers

### # Writing Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the class itself, allowing you to define several event handlers within a single class. Subscribers should define a `subscribe` method, which will be passed an event dispatcher instance. You may call the `listen` method on the given dispatcher to register event listeners:

```
<?php

namespace App\Listeners;

class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin($event) {}

    /**
     * Handle user logout events.
     */
    public function handleUserLogout($event) {}

    /**
     * Register the listeners for the subscriber.
     *
     * @param  \Illuminate\Events\Dispatcher  $events
     */
    public function subscribe($events)
    {
        $events->listen(
            'Illuminate\Auth\Events\Login',
            'App\Listeners\UserEventSubscriber@handleUserLogin'
        );

        $events->listen(
            'Illuminate\Auth\Events\Logout',
            'App\Listeners\UserEventSubscriber@handleUserLogout'
        );
    }
}
```

## # Registering Event Subscribers

After writing the subscriber, you are ready to register it with the event dispatcher. You may register subscribers using the `$subscribe` property on the `EventServiceProvider`. For example, let's add the `UserEventSubscriber` to the list:

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvide

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        //
    ];

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}
```

## Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

| Highlights | Resources | Partners | Ecosystem |
|---|---|---|---|
| Release Notes | Laracasts | Vehikl | Vapor |
| Getting Started | Laravel News | Tighten Co. | Forge |
| Routing | Laracon | Kirschbaum | Envoyer |
| Blade Templates | Laracon EU | Byte 5 | Horizon |
| Authentication | Laracon AU | 64Robots | Lumen |
| Authorization | Jobs | Cubet | Nova |
| Artisan Console | Certification | DevSquad | Echo |
| Database | Forums | Ideil | Valet |
| Eloquent ORM | | Cyber-Duck | Mix |
| Testing | | ABOUT YOU | Spark |
| | | Become A Partner | Cashier |
| | | | Homestead |
| | | | Dusk |
| | | | Passport |
| | | | Scout |
| | | | Socialite |

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.