Job Events

Using the **before** and **after methods** on the Queue facade, you may specify callbacks to be executed **before** or **after** a queued job is processed. These callbacks are a great opportunity to perform additional logging or **increment statistics for a dashboard**. Typically, you should call these methods from the **boot method** of a service provider. For example, we may use the **AppServiceProvider** that is included with Laravel:

```
use Illuminate\Support\Facades\Queue;

use Illuminate\Support\ServiceProvider;

use Illuminate\Queue\Events\JobProcessed;

use Illuminate\Queue\Events\JobProcessing;


public function boot(): void

  {

     Queue::before(function (JobProcessing $event) {

        // $event->connectionName

        // $event->job

        // $event->job->payload()

     });


     Queue::after(function (JobProcessed $event) {

        // $event->connectionName

        // $event->job
```

```
            // $event->job->payload()
        });
    }
```
****

Using the looping method on the Queue facade, you may specify callbacks that execute before the worker attempts to fetch a job from a queue. For example, you might register a closure to rollback any transactions that were left open by a previously failed job:

```
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Queue;


Queue::looping(function () {
    while (DB::transactionLevel() > 0) {
        DB::rollBack();
    }
});
```
****

Laravel's events provide a simple observer pattern implementation, allowing you to subscribe and listen for various events that occur within your application. Event classes are typically stored in the **app/Events** directory, while their listeners are stored in **app/Listeners**.

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 **App\Events\OrderShipped** event which a listener can receive and use to dispatch a Slack notification.

****

Registering Events & Listeners

The **App\Providers\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.

```
use App\Events\OrderShipped;

use App\Listeners\SendShipmentNotification;


/**

 * The event listener mappings for the application.

 *

 * @var array<class-string, array<int, class-string>>

 */

protected $listen = [

    OrderShipped::class => [

        SendShipmentNotification::class,
```

],

];

****

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

****

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** Artisan command. This command will generate any events or listeners that are listed in your **EventServiceProvider** that do not already exist:

php artisan event:generate

****

Alternatively, you may use the **make:event** and **make:listener** Artisan commands to generate individual events and listeners:

php artisan make:event PodcastProcessed

php artisan make:listener SendPodcastNotification --event=PodcastProcessed

****

Manually Registering Events

Typically, events should be registered via the **EventServiceProvider $listen array**; however, you may also register class or closure based event listeners manually in the boot method of your **EventServiceProvider**:

```
use App\Events\PodcastProcessed;

use App\Listeners\SendPodcastNotification;

use Illuminate\Support\Facades\Event;


/**

 * Register any other events for your application.

 */

public function boot(): void

{

    Event::listen(

        PodcastProcessed::class,

        [SendPodcastNotification::class, 'handle']

    );


    Event::listen(function (PodcastProcessed $event) {

        // ...

    });

}
```

****

Queueable Anonymous Event Listeners

When registering closure based event listeners manually, you may wrap the listener closure within the **Illuminate\Events\queueable function** to instruct Laravel to execute the listener using the queue:

```
use App\Events\PodcastProcessed;

use function Illuminate\Events\queueable;

use Illuminate\Support\Facades\Event;


/**

 * Register any other events for your application.

 */

public function boot(): void

{

    Event::listen(queueable(function (PodcastProcessed
$event) {

        // ...

    }));

}
```

****

```
// Note: in this way we execute the event in response handler.
        Event::listen(
            ThirdEvent::class,
            [ThirdEventListener::class, 'handle'],
        );
```

****

Like queued jobs, you may use the **onConnection**, **onQueue**, and **delay methods** to customize the execution of the **queued listener**:

```
Event::listen(queueable(function (PodcastProcessed $event)
{
    // ...
})->onConnection('redis')->onQueue('podcasts')-
>delay(now()->addSeconds(10)));
```

****

If you would like to handle anonymous queued listener failures, you may provide a closure to the catch method while defining the queueable listener. This closure will receive the event instance and the Throwable instance that caused the listener's failure:

```
use App\Events\PodcastProcessed;

use function Illuminate\Events\queueable;

use Illuminate\Support\Facades\Event;

use Throwable;

Event::listen(queueable(function (PodcastProcessed $event)
{
    // ...
})->catch(function (PodcastProcessed $event, Throwable $e)
{
    // The queued listener failed...
}));
```

****

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 (string $eventName, array $data) {
    // ...
});
```

****

Event Discovery

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 PHP's reflection services. When Laravel finds any listener class method that begins with handle or **__invoke**, Laravel will register those methods as **event listeners** for the event that is type-hinted in the method's signature

****

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

```
protected function discoverEventsWithin(): array
{
    return [
        $this->app->path('Listeners'),
    ];
}
```

****

Event Discovery In Production

In production, it is not efficient for 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.

****

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 email or making an HTTP request. Before using queued listeners, make sure to configure your queue and start a queue worker 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** and **make:listener** Artisan commands 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
{
   // ...
}
```

****

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:***

```
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;
}
```

If you would like to define the listener's queue connection or queue name at runtime, you may
define **viaConnection** or **viaQueue methods** on the listener:

```
public function viaConnection(): string

{

   return 'sqs';

}


public function viaQueue(): string

{

   return 'listeners';

}
```

****