Note (Test this with Jobs).

Conditionally Queueing Listeners

Sometimes, you may need to determine whether a listener should be queued based on some data that are only available at runtime. To accomplish this, a **shouldQueue** method may be added to a listener to determine whether the listener should be queued. If the **shouldQueue method** returns false, the listener will not be executed:

```
<?php

namespace App\Listeners;

use App\Events\OrderCreated;

use Illuminate\Contracts\Queue\ShouldQueue;

class RewardGiftCard implements ShouldQueue

{

    /**

     * Reward a gift card to the customer.

     */

    public function handle(OrderCreated $event): void

    { }

    /**

     * Determine whether the listener should be queued.

     */

    public function shouldQueue(OrderCreated $event): bool

    {

        return $event->order->subtotal >= 5000;
```

```
    }
}
```

****

Manually Interacting With 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

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.

     */

    public function handle(OrderShipped $event): void

    {

        if (true) {
```

```
            $this->release(30);
        }
    }
}
```

****

Queued Event Listeners & Database Transactions

When queued listeners are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your listener depends on these models, unexpected errors can occur when the job that dispatches the queued listener is processed.

If your queue connection's **after_commit** configuration option is set to false, you may still indicate that a particular queued listener should be dispatched after all open database transactions have been committed by defining an **$afterCommit** property on the listener class:

```
<?php

namespace App\Listeners;

use Illuminate\Contracts\Queue\ShouldQueue;

use Illuminate\Queue\InteractsWithQueue;

class SendShipmentNotification implements ShouldQueue

{

    use InteractsWithQueue;
```

```
    public $afterCommit = true;
}
```

****

Handling Failed Jobs

Sometimes your queued event listeners may **fail**. If the 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 **Throwable** that caused the failure:

(This will be inside the listener class):

```
/**
* Handle a job failure.
*/
    public function failed(OrderShipped $event, Throwable $exception): void
    {
        // ...
    }
```

****

Specifying Queued Listener Maximum Attempts

If one of your queued listeners is encountering an error, you likely do not want it to keep retrying indefinitely. Therefore, Laravel provides various ways to specify how many times or for how long a listener may be attempted.

You may define a **$tries** property on your **listener class** to specify how many times the listener may be attempted before it is considered to have **failed**:

```
class SendShipmentNotification implements ShouldQueue
{
    use InteractsWithQueue;


    /**
     * The number of times the queued listener may be
       attempted.
     * @var int
     */
    public $tries = 5;
}
```

As an alternative to defining how many times a listener may be attempted before it fails, you may define a time at which the listener should no longer be attempted. This allows a listener to be attempted any number of times within a given time frame. To define the time at which a listener should no longer be attempted, add a **retryUntil method** to your listener class. This method should return a **DateTime instance**:

```
use DateTime;
/**
 * Determine the time at which the listener should timeout.
 */
public function retryUntil(): DateTime
{
    return now()->addMinutes(5);
}
```

****

Dispatching Events

To dispatch an event, you may call the static dispatch method on the event. This method is made available on the event by the **Illuminate\Foundation\Events\Dispatchable trait**. Any arguments passed to the dispatch method will be passed to the event's constructor:

```
class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);
```

```
        // Order shipment logic...

        OrderShipped::dispatch($order);

        return redirect('/orders');
    }
}
```

****

Note: Try This with Jobs.

If you would like to conditionally dispatch an event, you may use the **dispatchIf** and **dispatchUnless** methods:

OrderShipped::dispatchIf($condition, $order);

OrderShipped::dispatchUnless($condition, $order);

****

Event Subscribers

Writing Event Subscribers

Event subscribers are classes that may subscribe to multiple events from within the **subscriber 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
<?php
namespace App\Listeners;
use Illuminate\Auth\Events\Login;
use Illuminate\Auth\Events\Logout;
use Illuminate\Events\Dispatcher;
class UserEventSubscriber
{
    /**
     * Handle user login events.
     */
    public function handleUserLogin(string $event): void {}
    /**
     * Handle user logout events.
     */
    public function handleUserLogout(string $event): void {}
    /**
     * Register the listeners for the subscriber.
     */
    public function subscribe(Dispatcher $events): void
    {
        $events->listen(
            Login::class,
            [UserEventSubscriber::class, 'handleUserLogin']
```

```
        );

        $events->listen(
            Logout::class,
            [UserEventSubscriber::class, 'handleUserLogout']
        );
    }
}
```

****

If your event listener methods are defined within the subscriber itself, you may find it more convenient to return an array of events and method names from the **subscriber's subscribe method**. Laravel will automatically determine the subscriber's class name when registering the event listeners:

```
/**
* Register the listeners for the subscriber.
* @return array<string, string>
*/
    public function subscribe(Dispatcher $events): array
    {
        return [
            Login::class => 'handleUserLogin',
            Logout::class => '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:

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

    /**
     * The subscriber classes to register.
     *
     * @var array
     */
    protected $subscribe = [
```

```
        UserEventSubscriber::class,
    ];
}
**************************************************
```