Events

Events

How do I create a new event listener?

To create a new listener class for one of our events, we can use Artisan to scaffold the listener from the project root directory:

```
php artisan make:listener UpdateDetails
```

After adding all the necessary functionality for our listener, we have to make Aero aware of it by adding it to a listen property, wrapped within an event like so:

```
protected $listen = [
    Registered::class => [
            UpdateDetails::class,
    ]
];
```

There is also an alternative, for events that should not be queued:

```
Event::listen(OrderPlaced::class, function ($event) {
    $order = $event->order;
    //...
});
```

Extending ManagedListener

In order to use a **ManagedListener**, we have to have an event extending **ManagedEvent**. This can be accomplished like so:

```
use Aero\Events\ManagedHandler;
```

```
protected $listen = [
    FormSubmitted::class => [
        ManagedHandler::class,
    ],
];
```

What are the events available?

Account

- AddressCreated - AddressDeleted - AddessUpdated - CustomerManualylCreated - CustomerRegistered - CustomerUpdated - CustomerChanged - PasswordResetRequest

Cart

- CartEmptied - CartItemAdded - CartItemRemoved - CartItemUpdated - OrderCanceled - OrderClosed - OrderComplete - OrderConfirmation - OrderDispatched - OrderItemBought - OrderOnHold - OrderPartiallyDispatched - OrderPartiallyReturned - OrderPlaced - OrderProcessing - OrderReturned - OrderStatusUpdated - OrderSuccessful - OrderUpdated

Catalog

- AttributesDeleted - AttributesUpdated - CategoryDeleted - CategoryUpdated - ListingCreated - ListingsDeleted - ListingsUpdated - ListingUpdated - ManufacturerUpdated - ProductCreated - ProductDeleted - ProductUpdated - TagsUpdated

Content

- BlockCreated - BlockDeleted - BlockUpdated

Fulfillment

- FulfillmentDispatched

Payment

- PaymentCanceled - PaymentCaptured - PaymentFailed - PaymentRefunded

Redirector

- RedirectHit - RedirectNotFound

Store

- FormSubmitted

Subscription

- SubscriptionCanceled - SubscriptionCardExpired - SubscriptionCreated - SubscriptionFailed - SubscriptionOrderFailed - SubscriptionPaused - SubscriptionPaymentFailed - SubscriptionPaymentSuccess - SubscriptionShippingFailed - SubscriptionSuccessful - SubscriptionSuccessful - SubscriptionUnpaused - SubscriptionUpcoming

How do I add a custom event to a Service Provider?

Laravel events provide an observer implementation, allowing you to listen and observe for specific tasks happening in your application. Adding a custom event to a **ServiceProvider**means that the application will listen to it as soon as it occurs. In terms of choosing the right **ServiceProvider**for the job, it breaks down into two options: a Module **ServiceProvider**, or any of the Providers listed under **app/Providers**of our shop's root directory.

Event directory:

```
app/Events
```

Listener directory:

```
app/Listeners
```

The **ServiceProvider**class has a **$listen**property, to which we simply add on the Event that we wish to add, alongside its listener.

What are event listeners?

Event listeners, as the name suggests, listen to events that have been assigned to them. This means that listeners have to first be manually mapped for which events they should listen to.

Mappings for **Event Listeners**are declared in the appropriate **Service Provider**, depending on what part of the platform it affects - for example, it could be a **Module Service Provider**or the **EventServiceProvider**in the app directory of Aero.

How do I add a listener to an event?

```
<?php

namespace App\Providers;

use Illuminate\Auth\Events\Registered;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Event;

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

    /**
```

```
     * Register any events for your application.
     *
     * @return void
     */
    public function boot()
    {
        parent::boot();

        //
    }
}
```

The listener is always nested within the event which it affects so that it can pass on the event to the listener’s handle method. Remember that the listener always goes within the array instantiating from the event class.

How do I create a new event?

To create a new event, we first have to scaffold the class using Artisan in the project root directory:

```
php artisan make:event Registered
```

The above command scaffolds a class into the **app/Events**and we can modify it accordingly to our needs.

After applying all the necessary functionality to the event, we have to also create a listener to listen to an event happening. In order for Aero to recognize the event, we have to specify it in a **Service Provider**of choice, for example, a module **Service Provider**:

```
protected $listen = [
        Registered::class => [
                // insert your listener here
        ]
];
```

Extending ManagedEvent

Aero's **ManagedEvent**is an **optional class**made specifically for Aero events and is used mainly for mailing events. These are a special type of event as all the events that extend **ManagedEvent** are actions.

It's important to note that the listener that listens to an event extending **ManagedEvent**, must also extend the **ManagedListener**class.

Variables

With **ManagedEvent**, we can create variables that can then be accessible in a mailing template. In essence, variables create helper text and allow the user to use data from the event itself. To define some variables, we have to create a property 'variables' in our event:

Adding variables

```
public static $variables = [
    'product.slug',
    'product.name',
    'product.model',
    'product.summary',
    'product.description',
    'product.thumbnail',
    'product.heading',
    'product.url',
    'product.categories.*.name',
    'product.categories.*.has_featured_image',
    'product.categories.*.featured_image_file',
    'product.manufacturer.name',
    'product.manufacturer.has_logo',
    'product.manufacturer.logo_file',
    'product.attributes.*.name',
    'product.attributes.*.image_file',
    'product.images.*.image_file',
    'product.all_images.*.image_file',
    'product.variants.*.sku',
    'product.variants.*.has_stock',
    'product.highest_price',
    'product.lowest_price',
    'product.has_reductions',
];
```

The above variables will then be accessible in our mail template.

An alternative method of adding variables:

```
EventClass::addVariable(‘product.lowest_price’)
```

Or multiple, as an array:

```
EventClass::addVariables([‘product.manufacturer.name’, ‘product.lowest_price’])
```

How do I make custom mail notification events?

To create an event that shows up in the Mail Notifications part of the admin your event needs to extend **Aero\Events\ManagedEvent**and have **Aero\Events\ManagedHandler**as a registered listener.

To create an event you need to create a class that extends **Aero\Events\ManagedEvent**.

```
<?php

namespace Acme\MyModule\Events;

use Aero\Events\ManagedEvent;

class MyEvent extends ManagedEvent
{
   //
}
```

To register **Aero\Events\ManagedHandler**as a listener of your event you need to add your event and the managed handler to a **$listen**array inside of a service provider.

```
<?php

namespace Acme\MyModule;

use Acme\MyModule\Events\MyEvent;
use Aero\Common\Providers\ModuleServiceProvider;
use Aero\Events\ManagedHandler;

class ServiceProvider extends ModuleServiceProvider
{
   protected $listen = [
       MyEvent::class => [
           ManagedHandler::class,
```

```
        ],
    ];

    public function setup()
    {
        //
    }
}
```