Laravel Cashier

Introduction

Laravel Cashier provides an expressive, fluent interface to Stripe's subscription billing services. It handles almost all of the boilerplate subscription billing code you are dreading writing. In addition to basic subscription management, Cashier can handle coupons, swapping subscription, subscription "quantities", cancellation grace periods, and even generate invoice PDFs.

Upgrading Cashier

When upgrading to a new version of Cashier, it's important that you carefully review the upgrade guide.

> **!** To prevent breaking changes, Cashier uses a fixed Stripe API version. Cashier 10.1 utilizes Stripe API version `2019-08-14`. The Stripe API version will be updated on minor releases in order to make use of new Stripe features and improvements.

Installation

First, require the Cashier package for Stripe with Composer:

```
composer require laravel/cashier
```

To ensure Cashier properly handles all Stripe events, remember to set up Cashier's webhook handling.

Database Migrations

The Cashier service provider registers its own database migration directory, so remember to migrate your database after installing the package. The Cashier migrations will add several columns to your `users` table as well as create a new `subscriptions` table to hold all of your customer's subscriptions:

```
php artisan migrate
```

If you need to overwrite the migrations that ship with the Cashier package, you can publish them using the `vendor:publish` Artisan command:

```
php artisan vendor:publish --tag="cashier-migrations"
```

If you would like to prevent Cashier's migrations from running entirely, you may use the `ignoreMigrations` provided by Cashier. Typically, this method should be called in the `register` method of your `AppServiceProvider`:

```
use Laravel\Cashier\Cashier;

Cashier::ignoreMigrations();
```

Stripe recommends that any column used for storing Stripe identifiers should be case-sensitive. Therefore, you should ensure the column collation for the `stripe_id` column is set to, for example, `utf8_bin` in MySQL. More info can be found in the Stripe documentation.

Configuration

Billable Model

Before using Cashier, add the `Billable` trait to your model definition. This trait provides various methods to allow you to perform common billing tasks, such as creating subscriptions, applying coupons, and updating payment method information:

```
use Laravel\Cashier\Billable;

class User extends Authenticatable
{
    use Billable;
}
```

Cashier assumes your Billable model will be the `App\User` class that ships with Laravel. If you wish to change this you can specify a different model in your `.env` file:

```
CASHIER_MODEL=App\User
```

If you're using a model other than Laravel's supplied `App\User` model, you'll need to publish and alter the migrations provided to match your alternative model's table name.

API Keys

Next, you should configure your Stripe key in your `.env` file. You can retrieve your Stripe API keys from the Stripe control panel.

```
STRIPE_KEY=your-stripe-key
STRIPE_SECRET=your-stripe-secret
```

Currency Configuration

The default Cashier currency is United States Dollars (USD). You can change the default currency by setting the `CASHIER_CURRENCY` environment variable:

```
CASHIER_CURRENCY=eur
```

In addition to configuring Cashier's currency, you may also specify a locale to be used when formatting money values for display on invoices. Internally, Cashier utilizes PHP's `NumberFormatter` class to set the currency locale:

```
CASHIER_CURRENCY_LOCALE=nl_BE
```

In order to use locales other than `en`, ensure the `ext-intl` PHP extension is installed and configured on your server.

Logging

Cashier allows you to specify the log channel to be used when logging all Stripe related exceptions. You may specify the log channel using the `CASHIER_LOGGER` environment variable:

```
CASHIER_LOGGER=default
```

Customers

Creating Customers

Occasionally, you may wish to create a Stripe customer without beginning a subscription. You may accomplish this using the `createAsStripeCustomer` method:

```
$user->createAsStripeCustomer();
```

Once the customer has been created in Stripe, you may begin a subscription at a later date.

Payment Methods

Storing Payment Methods

In order to create subscriptions or perform "one off" charges with Stripe, you will need to store a payment method and retrieve its identifier from Stripe. The approach used to accomplish differs based on whether you plan to use the payment method for subscriptions or single charges, so we will examine both below.

Payment Methods For Subscriptions

When storing credit cards to a customer for future use, the Stripe Setup Intents API must be used to securely gather the customer's payment method details. A "Setup Intent" indicates to Stripe the intention to charge a customer's payment method. Cashier's `Billable` trait includes the `createSetupIntent` to easily create a new Setup Intent. You should call this method from the route or controller that will render the form which gathers your customer's payment method details:

```
return view('update-payment-method', [
    'intent' => $user->createSetupIntent()
]);
```

After you have created the Setup Intent and passed it to the view, you should attach its secret to the element that will gather the payment method. For example, consider this "update payment method" form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button" data-secret="{{ $intent->client_secret }}">
    Update Payment Method
</button>
```

Next, the Stripe.js library may be used to attach a Stripe Element to the form and securely gather the customer's payment details:

```
<script src="https://js.stripe.com/v3/"></script>

<script>
    const stripe = Stripe('stripe-public-key');

    const elements = stripe.elements();
    const cardElement = elements.create('card');

    cardElement.mount('#card-element');
</script>
```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using Stripe's `handleCardSetup` method:

```
const cardHolderName = document.getElementById('card-holder-name');
const cardButton = document.getElementById('card-button');
const clientSecret = cardButton.dataset.secret;

cardButton.addEventListener('click', async (e) => {
    const { setupIntent, error } = await stripe.handleCardSetup(
        clientSecret, cardElement, {
            payment_method_data: {
                billing_details: { name: cardHolderName.value }
            }
        }
    );

    if (error) {
        // Display "error.message" to the user...
    } else {
        // The card has been verified successfully...
    }
});
```

After the card has been verified by Stripe, you may pass the resulting `setupIntent.payment_method` identifier to your Laravel application, where it can be attached to the customer. The payment method can either be added as a new payment method or used to update the default payment method. You can also immediately use the payment method identifier to create a new subscription.

If you would like more information about Setup Intents and gathering customer payment details please review this overview provided by Stripe.

Payment Methods For Single Charges

Of course, when making a single charge against a customer's payment method we'll only need to use a payment method identifier a single time. Due to Stripe limitations, you may not use the stored default payment method of a customer for single charges. You must allow the customer to enter their payment method details using the Stripe.js library. For example, consider the following form:

```
<input id="card-holder-name" type="text">

<!-- Stripe Elements Placeholder -->
<div id="card-element"></div>

<button id="card-button">
    Process Payment
</button>
```

Next, the Stripe.js library may be used to attach a Stripe Element to the form and securely gather the customer's payment details:

```
<script src="https://js.stripe.com/v3/"></script>

<script>
    const stripe = Stripe('stripe-public-key');

    const elements = stripe.elements();
    const cardElement = elements.create('card');

    cardElement.mount('#card-element');
</script>
```

Next, the card can be verified and a secure "payment method identifier" can be retrieved from Stripe using Stripe's `createPaymentMethod` method:

```
const cardHolderName = document.getElementById('card-holder-name');
```

```
const cardButton = document.getElementById('card-button');

cardButton.addEventListener('click', async (e) => {
    const { paymentMethod, error } = await stripe.createPaymentMethod(
        'card', cardElement, {
            billing_details: { name: cardHolderName.value }
        }
    );

    if (error) {
        // Display "error.message" to the user...
    } else {
        // The card has been verified successfully...
    }
});
```

If the card is verified successfully, you may pass the `paymentMethod.id` to your Laravel application and process a single charge.

Retrieving Payment Methods

The `paymentMethods` method on the Billable model instance returns a collection of `Laravel\Cashier\PaymentMethod` instances:

```
$paymentMethods = $user->paymentMethods();
```

To retrieve the default payment method, the `defaultPaymentMethod` method may be used;

```
$paymentMethod = $user->defaultPaymentMethod();
```

Determining If A User Has A Payment Method

To determine if a Billable model has a payment method attached to their account, use the `hasPaymentMethod` method:

```
if ($user->hasPaymentMethod()) {
    //
}
```

Updating The Default Payment Method

The `updateDefaultPaymentMethod` method may be used to update a customer's default payment method information. This method accepts a Stripe payment method identifier and will assign the new payment method as the default billing payment method:

```
$user->updateDefaultPaymentMethod($paymentMethod);
```

To sync your default payment method information with the customer's default payment method information in Stripe, you may use the `updateDefaultPaymentMethodFromStripe` method:

```
$user->updateDefaultPaymentMethodFromStripe();
```

The default payment method on a customer can only be used for invoicing and creating new subscriptions. Due to limitations from Stripe, it may not be used for single charges.

Adding Payment Methods

To add a new payment method, you may call the `addPaymentMethod` method on the billable user, passing the payment method identifier:

```
$user->addPaymentMethod($paymentMethod);
```

To learn how to retrieve payment method identifiers please review the payment method storage documentation.

Deleting Payment Methods

To delete a payment method, you may call the `delete` method on the `Laravel\Cashier\PaymentMethod` instance you wish to delete:

```
$paymentMethod->delete();
```

The `deletePaymentMethods` method will delete all of the payment method information for the Billable model:

```
$user->deletePaymentMethods();
```

If a user has an active subscription, you should prevent them from deleting their default payment method.

Subscriptions

Creating Subscriptions

To create a subscription, first retrieve an instance of your billable model, which typically will be an instance of `App\User`. Once you have retrieved the model instance, you may use the `newSubscription` method to create the model's subscription:

```
$user = User::find(1);

$user->newSubscription('main', 'premium')->create($paymentMethod);
```

The first argument passed to the `newSubscription` method should be the name of the subscription. If your application only offers a single subscription, you might call this `main` or `primary`. The second argument is the specific plan the user is subscribing to. This value should correspond to the plan's identifier in Stripe.

The `create` method, which accepts a Stripe payment method identifier or Stripe `PaymentMethod` object, will begin the subscription as well as update your database with the customer ID and other relevant billing information.

Passing a payment method identifier directly to the `create()` subscription method will also automatically add it to the user's stored payment methods.

Additional User Details

If you would like to specify additional customer details, you may do so by passing them as the second argument to the `create` method:

```
$user->newSubscription('main', 'monthly')->create($paymentMethod, [
    'email' => $email,
]);
```

To learn more about the additional fields supported by Stripe, check out Stripe's documentation on customer creation.

Coupons

If you would like to apply a coupon when creating the subscription, you may use the `withCoupon` method:

```
$user->newSubscription('main', 'monthly')
     ->withCoupon('code')
     ->create($paymentMethod);
```

Checking Subscription Status

Once a user is subscribed to your application, you may easily check their subscription status using a variety of convenient methods. First, the `subscribed` method returns `true` if the user has an active subscription, even if the subscription is currently within its trial period:

```
if ($user->subscribed('main')) {
```

```
    //
}
```

The `subscribed` method also makes a great candidate for a route middleware, allowing you to filter access to routes and controllers based on the user's subscription status:

```
public function handle($request, Closure $next)
{
    if ($request->user() && ! $request->user()->subscribed('main')) {
        // This user is not a paying customer...
        return redirect('billing');
    }

    return $next($request);
}
```

If you would like to determine if a user is still within their trial period, you may use the `onTrial` method. This method can be useful for displaying a warning to the user that they are still on their trial period:

```
if ($user->subscription('main')->onTrial()) {
    //
}
```

The `subscribedToPlan` method may be used to determine if the user is subscribed to a given plan based on a given Stripe plan ID. In this example, we will determine if the user's `main` subscription is actively subscribed to the `monthly` plan:

```
if ($user->subscribedToPlan('monthly', 'main')) {
    //
}
```

By passing an array to the `subscribedToPlan` method, you may determine if the user's `main` subscription is actively subscribed to the `monthly` or the `yearly` plan:

```
if ($user->subscribedToPlan(['monthly', 'yearly'], 'main')) {
    //
}
```

The `recurring` method may be used to determine if the user is currently subscribed and is no longer within their trial period:

```
if ($user->subscription('main')->recurring()) {
    //
}
```

Cancelled Subscription Status

To determine if the user was once an active subscriber, but has cancelled their subscription, you may use the `cancelled` method:

```
if ($user->subscription('main')->cancelled()) {
    //
}
```

You may also determine if a user has cancelled their subscription, but are still on their "grace period" until the subscription fully expires. For example, if a user cancels a subscription on March 5th that was originally scheduled to expire on March 10th, the user is on their "grace period" until March 10th. Note that the `subscribed` method still returns `true` during this time:

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

To determine if the user has cancelled their subscription and is no longer within their "grace period", you may use the `ended` method:

```
if ($user->subscription('main')->ended()) {
    //
}
```

Incomplete and Past Due Status

If a subscription requires a secondary payment action after creation the subscription will be marked as `incomplete`. Subscription statuses are stored in the `stripe_status` column of Cashier's `subscriptions` database table.

Similarly, if a secondary payment action is required when swapping plans the subscription will be marked as `past_due`. When your subscription is in either of these states it will not be active until the customer has confirmed their payment. Checking if a subscription has an incomplete payment can be done using the `hasIncompletePayment` method on the Billable model or a subscription instance:

```
if ($user->hasIncompletePayment('main')) {
    //
}

if ($user->subscription('main')->hasIncompletePayment()) {
    //
}
```

When a subscription has an incomplete payment, you should direct the user to Cashier's payment confirmation page, passing the `latestPayment` identifier. You may use the `latestPayment` method available on subscription instance to retrieve this identifier:

```
<a href="{{ route('cashier.payment', $subscription->latestPayment()->id) }}">
    Please confirm your payment.
</a>
```

When a subscription is in an `incomplete` state it cannot be changed until the payment is confirmed. Therefore, the `swap` and `updateQuantity` methods will throw an exception when the subscription is in an `incomplete` state.

Changing Plans

After a user is subscribed to your application, they may occasionally want to change to a new subscription plan. To swap a user to a new subscription, pass the plan's identifier to the `swap` method:

```
$user = App\User::find(1);

$user->subscription('main')->swap('provider-plan-id');
```

If the user is on trial, the trial period will be maintained. Also, if a "quantity" exists for the subscription, that quantity will also be maintained.

If you would like to swap plans and cancel any trial period the user is currently on, you may use the `skipTrial` method:

```
$user->subscription('main')
        ->skipTrial()
        ->swap('provider-plan-id');
```

If you would like to swap plans and immediately invoice the user instead of waiting for their next billing cycle, you may use the `swapAndInvoice` method:

```
$user = App\User::find(1);

$user->subscription('main')->swapAndInvoice('provider-plan-id');
```

Subscription Quantity

Sometimes subscriptions are affected by "quantity". For example, your application might charge $10 per month **per user** on an account. To easily increment or decrement your subscription quantity, use the `incrementQuantity` and `decrementQuantity` methods:

```
$user = User::find(1);

$user->subscription('main')->incrementQuantity();

// Add five to the subscription's current quantity...
$user->subscription('main')->incrementQuantity(5);

$user->subscription('main')->decrementQuantity();
```

```
// Subtract five to the subscription's current quantity...
$user->subscription('main')->decrementQuantity(5);
```

Alternatively, you may set a specific quantity using the `updateQuantity` method:

```
$user->subscription('main')->updateQuantity(10);
```

The `noProrate` method may be used to update the subscription's quantity without pro-rating the charges:

```
$user->subscription('main')->noProrate()->updateQuantity(10);
```

For more information on subscription quantities, consult the Stripe documentation.

Subscription Taxes

To specify the tax percentage a user pays on a subscription, implement the `taxPercentage` method on your billable model, and return a numeric value between 0 and 100, with no more than 2 decimal places.

```
public function taxPercentage()
{
    return 20;
}
```

The `taxPercentage` method enables you to apply a tax rate on a model-by-model basis, which may be helpful for a user base that spans multiple countries and tax rates.

The `taxPercentage` method only applies to subscription charges. If you use Cashier to make "one off" charges, you will need to manually specify the tax rate at that time.

Syncing Tax Percentages

When changing the hard-coded value returned by the `taxPercentage` method, the tax settings on any existing subscriptions for the user will remain the same. If you wish to update the tax value for existing subscriptions with the returned `taxPercentage` value, you should call the `syncTaxPercentage` method on the user's subscription instance:

```
$user->subscription('main')->syncTaxPercentage();
```

Subscription Anchor Date

By default, the billing cycle anchor is the date the subscription was created, or if a trial period is used, the date that the trial ends. If you would like to modify the billing anchor date, you may use the `anchorBillingCycleOn` method:

```
use App\User;
use Carbon\Carbon;

$user = User::find(1);

$anchor = Carbon::parse('first day of next month');

$user->newSubscription('main', 'premium')
            ->anchorBillingCycleOn($anchor->startOfDay())
            ->create($paymentMethod);
```

For more information on managing subscription billing cycles, consult the Stripe billing cycle documentation

Cancelling Subscriptions

To cancel a subscription, call the `cancel` method on the user's subscription:

```
$user->subscription('main')->cancel();
```

When a subscription is cancelled, Cashier will automatically set the `ends_at` column in your database. This column is used to know when the `subscribed` method should begin returning `false`. For example, if a customer cancels a subscription on March

1st, but the subscription was not scheduled to end until March 5th, the `subscribed` method will continue to return `true` until March 5th.

You may determine if a user has cancelled their subscription but are still on their "grace period" using the `onGracePeriod` method:

```
if ($user->subscription('main')->onGracePeriod()) {
    //
}
```

If you wish to cancel a subscription immediately, call the `cancelNow` method on the user's subscription:

```
$user->subscription('main')->cancelNow();
```

Resuming Subscriptions

If a user has cancelled their subscription and you wish to resume it, use the `resume` method. The user **must** still be on their grace period in order to resume a subscription:

```
$user->subscription('main')->resume();
```

If the user cancels a subscription and then resumes that subscription before the subscription has fully expired, they will not be billed immediately. Instead, their subscription will be re-activated, and they will be billed on the original billing cycle.

Subscription Trials

With Payment Method Up Front

If you would like to offer trial periods to your customers while still collecting payment method information up front, you should use the `trialDays` method when creating your subscriptions:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')
            ->trialDays(10)
            ->create($paymentMethod);
```

This method will set the trial period ending date on the subscription record within the database, as well as instruct Stripe to not begin billing the customer until after this date. When using the `trialDays` method, Cashier will overwrite any default trial period configured for the plan in Stripe.

If the customer's subscription is not cancelled before the trial ending date they will be charged as soon as the trial expires, so you should be sure to notify your users of their trial ending date.

The `trialUntil` method allows you to provide a `DateTime` instance to specify when the trial period should end:

```
use Carbon\Carbon;

$user->newSubscription('main', 'monthly')
            ->trialUntil(Carbon::now()->addDays(10))
            ->create($paymentMethod);
```

You may determine if the user is within their trial period using either the `onTrial` method of the user instance, or the `onTrial` method of the subscription instance. The two examples below are identical:

```
if ($user->onTrial('main')) {
    //
}

if ($user->subscription('main')->onTrial()) {
    //
}
```

Without Payment Method Up Front

If you would like to offer trial periods without collecting the user's payment method information up front, you may set the `trial_ends_at` column on the user record to your desired trial ending date. This is typically done during user registration:

```
$user = User::create([
    // Populate other user properties...
    'trial_ends_at' => now()->addDays(10),
]);
```

Be sure to add a date mutator for `trial_ends_at` to your model definition.

Cashier refers to this type of trial as a "generic trial", since it is not attached to any existing subscription. The `onTrial` method on the `User` instance will return `true` if the current date is not past the value of `trial_ends_at`:

```
if ($user->onTrial()) {
    // User is within their trial period...
}
```

You may also use the `onGenericTrial` method if you wish to know specifically that the user is within their "generic" trial period and has not created an actual subscription yet:

```
if ($user->onGenericTrial()) {
    // User is within their "generic" trial period...
}
```

Once you are ready to create an actual subscription for the user, you may use the `newSubscription` method as usual:

```
$user = User::find(1);

$user->newSubscription('main', 'monthly')->create($paymentMethod);
```

Handling Stripe Webhooks

You may use Laravel Valet's `valet share` command to help test webhooks during local development.

Stripe can notify your application of a variety of events via webhooks. By default, a route that points to Cashier's webhook controller is configured through the Cashier service provider. This controller will handle all incoming webhook requests.

By default, this controller will automatically handle cancelling subscriptions that have too many failed charges (as defined by your Stripe settings), customer updates, customer deletions, subscription updates, and payment method changes; however, as we'll soon discover, you can extend this controller to handle any webhook event you like.

To ensure your application can handle Stripe webhooks, be sure to configure the webhook URL in the Stripe control panel. The full list of all webhooks you should configure in the Stripe control panel are:

- `customer.subscription.updated`
- `customer.subscription.deleted`
- `customer.updated`
- `customer.deleted`
- `invoice.payment_action_required`

Make sure you protect incoming requests with Cashier's included webhook signature verification middleware.

Webhooks & CSRF Protection

Since Stripe webhooks need to bypass Laravel's CSRF protection, be sure to list the URI as an exception in your `VerifyCsrfToken` middleware or list the route outside of the `web` middleware group:

```
protected $except = [
    'stripe/*',
];
```

Defining Webhook Event Handlers

Cashier automatically handles subscription cancellation on failed charges, but if you have additional webhook events you would like to handle, extend the Webhook controller. Your method names should correspond to Cashier's expected convention, specifically, methods should be prefixed with `handle` and the "camel case" name of the webhook you wish to handle. For example, if you wish to handle the `invoice.payment_succeeded` webhook, you should add a `handleInvoicePaymentSucceeded` method to the controller:

```
<?php

namespace App\Http\Controllers;

use Laravel\Cashier\Http\Controllers\WebhookController as CashierController;

class WebhookController extends CashierController
{
    /**
     * Handle invoice payment succeeded.
     *
     * @param  array  $payload
     * @return \Symfony\Component\HttpFoundation\Response
     */
    public function handleInvoicePaymentSucceeded($payload)
    {
        // Handle The Event
    }
}
```

Next, define a route to your Cashier controller within your `routes/web.php` file. This will overwrite the default shipped route:

```
Route::post(
    'stripe/webhook',
    '\App\Http\Controllers\WebhookController@handleWebhook'
);
```

Failed Subscriptions

What if a customer's credit card expires? No worries - Cashier's Webhook controller will cancel the customer's subscription for you. Failed payments will automatically be captured and handled by the controller. The controller will cancel the customer's subscription when Stripe determines the subscription has failed (normally after three failed payment attempts).

Verifying Webhook Signatures

To secure your webhooks, you may use Stripe's webhook signatures. For convenience, Cashier automatically includes a middleware which validates that the incoming Stripe webhook request is valid.

To enable webhook verification, ensure that the `STRIPE_WEBHOOK_SECRET` environment variable is set in your `.env` file. The webhook `secret` may be retrieved from your Stripe account dashboard.

Single Charges

Simple Charge

The `charge` method accepts the amount you would like to charge in the **lowest denominator of the currency used by your application**.

If you would like to make a "one off" charge against a subscribed customer's payment method, you may use the `charge` method on a billable model instance. You'll need to provide a payment method identifier as the second argument:

```
// Stripe Accepts Charges In Cents...
$stripeCharge = $user->charge(100, $paymentMethod);
```

The `charge` method accepts an array as its third argument, allowing you to pass any options you wish to the underlying Stripe charge creation. Consult the Stripe documentation regarding the options available to you when creating charges:

```
$user->charge(100, $paymentMethod, [
    'custom_option' => $value,
]);
```

The `charge` method will throw an exception if the charge fails. If the charge is successful, an instance of `Laravel\Cashier\Payment` will be returned from the method:

```
try {
    $payment = $user->charge(100, $paymentMethod);
} catch (Exception $e) {
    //
}
```

Charge With Invoice

Sometimes you may need to make a one-time charge but also generate an invoice for the charge so that you may offer a PDF receipt to your customer. The `invoiceFor` method lets you do just that. For example, let's invoice the customer $5.00 for a "One Time Fee":

```
// Stripe Accepts Charges In Cents...
$user->invoiceFor('One Time Fee', 500);
```

The invoice will be charged immediately against the user's default payment method. The `invoiceFor` method also accepts an array as its third argument. This array contains the billing options for the invoice item. The fourth argument accepted by the method is also an array. This final argument accepts the billing options for the invoice itself:

```
$user->invoiceFor('Stickers', 500, [
    'quantity' => 50,
], [
    'tax_percent' => 21,
]);
```

The `invoiceFor` method will create a Stripe invoice which will retry failed billing attempts. If you do not want invoices to retry failed charges, you will need to close them using the Stripe API after the first failed charge.

Refunding Charges

If you need to refund a Stripe charge, you may use the `refund` method. This method accepts the Stripe Payment Intent ID as its first argument:

```
$payment = $user->charge(100, $paymentMethod);

$user->refund($payment->id);
```

Invoices

You may easily retrieve an array of a billable model's invoices using the `invoices` method:

```
$invoices = $user->invoices();

// Include pending invoices in the results...
$invoices = $user->invoicesIncludingPending();
```

When listing the invoices for the customer, you may use the invoice's helper methods to display the relevant invoice information. For example, you may wish to list every invoice in a table, allowing the user to easily download any of them:

```
<table>
    @foreach ($invoices as $invoice)
        <tr>
            <td>{{ $invoice->date()->toFormattedDateString() }}</td>
            <td>{{ $invoice->total() }}</td>
```

```
            <td><a href="/user/invoice/{{ $invoice->id }}">Download</a></td>
        </tr>
    @endforeach
</table>
```

Generating Invoice PDFs

From within a route or controller, use the `downloadInvoice` method to generate a PDF download of the invoice. This method will automatically generate the proper HTTP response to send the download to the browser:

```
use Illuminate\Http\Request;

Route::get('user/invoice/{invoice}', function (Request $request, $invoiceId) {
    return $request->user()->downloadInvoice($invoiceId, [
        'vendor' => 'Your Company',
        'product' => 'Your Product',
    ]);
});
```

Strong Customer Authentication

If your business is based in Europe you will need to abide by the Strong Customer Authentication (SCA) regulations. These regulations were imposed in September 2019 by the European Union to prevent payment fraud. Luckily, Stripe and Cashier are prepared for building SCA compliant applications.

Before getting started, review Stripe's guide on PSD2 and SCA as well as their documentation on the new SCA API's.

Payments Requiring Additional Confirmation

SCA regulations often require extra verification in order to confirm and process a payment. When this happens, Cashier will throw an `IncompletePayment` exception that informs you that this extra verification is needed. After catching this exception, you have two options on how to proceed.

First, you could redirect your customer to the dedicated payment confirmation page which is included with Cashier. This page already has an associated route that is registered via Cashier's service provider. So, you may catch the `IncompletePayment` exception and redirect to the payment confirmation page:

```
use Laravel\Cashier\Exceptions\IncompletePayment;

try {
    $subscription = $user->newSubscription('default', $planId)
                         ->create($paymentMethod);
} catch (IncompletePayment $exception) {
    return redirect()->route(
        'cashier.payment',
        [$exception->payment->id, 'redirect' => route('home')]
    );
}
```

On the payment confirmation page, the customer will be prompted to enter their credit card info again and perform any additional actions required by Stripe, such as "3D Secure" confirmation. After confirming their payment, the user will be redirected to the URL provided by the `redirect` parameter specified above.

Alternatively, you could allow Stripe to handle the payment confirmation for you. In this case, instead of redirecting to the payment confirmation page, you may setup Stripe's automatic billing emails in your Stripe dashboard. However, if an `IncompletePayment` exception is caught, you should still inform the user they will receive an email with further payment confirmation instructions.

Incomplete payment exceptions may be thrown for the following methods: `charge`, `invoiceFor`, and `invoice` on the `Billable` user. When handling subscriptions, the `create` method on the `SubscriptionBuilder`, and the `incrementAndInvoice` and `swapAndInvoice` methods on the `Subscription` model may throw exceptions.

Incomplete and Past Due State

When a payment needs additional confirmation, the subscription will remain in an `incomplete` or `past_due` state as indicated by its `stripe_status` database column. Cashier will make automatically activate the customer's subscription via a webhook as soon as payment confirmation is complete.

For more information on `incomplete` and `past_due` states, please refer to our additional documentation.

Off-Session Payment Notifications

Since SCA regulations require customers to occasionally verify their payment details even while their subscription is active, Cashier can send a payment notification to the customer when off-session payment confirmation is required. For example, this may occur when a subscription is renewing. Cashier's payment notification can be enabled by setting the `CASHIER_PAYMENT_NOTIFICATION` environment variable to a notification class. By default, this notification is disabled. Of course, Cashier includes a notification class you may use for this purpose, but you are free to provide your own notification class if desired:

```
CASHIER_PAYMENT_NOTIFICATION=Laravel\Cashier\Notifications\ConfirmPayment
```

To ensure that off-session payment confirmation notifications are delivered, verify that Stripe webhooks are configured for your application and the `invoice.payment_action_required` webhook is enabled in your Stripe dashboard. In addition, your `Billable` model should also use Laravel's `Illuminate\Notifications\Notifiable` trait.

Notifications will be sent even when customers are manually making a payment that requires additional confirmation. Unfortunately, there is no way for Stripe to know that the payment was done manually or "off-session". But, a customer will simply see a "Payment Successful" message if they visit the payment page after already confirming their payment. The customer will not be allowed to accidentally confirm the same payment twice and incur an accidental second charge.

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

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- 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.