django-stripe-payments Documentation

Release 2.0.0

James Tauber and Contributors

Nov 12, 2017

Contents

Important: django-stripe-payments has been renamed, refactored, and re-released as pinax-stripe

CHAPTER 1

Development

The source repository can be found at https://github.com/pinax/django-stripe-payments

1.1 Contents

1.1.1 ChangeLog

2.0

- refactored models to be in line with api changes
- added a new setting to enable setting the API version
- added new event types to webhook signals
- handle the transfer.update webhook
- added ability to sync customer data

1.3

- removed tag.js
- fixed bugs in template tags due to missing forms
- updated included templates to be ready for checkout.js integration

1.2.2

- handle deprected 'diputed' property on charges

1.2.1

- upgraded stripe library requirement to 1.7.9

1.2

- added better admin for Charge
- do not raise exception if customer is deleted first in Stripe
- updated install notes

1.1.1

- Update to stripe 1.7.7
- Handle case of some charge messages not getting attached to their invoice

1.1

1.0

- initial release

1.1.2 API

The views in django-stripe-payments are intentionally very thin. In fact, quite often you might want to implement your own views for some or all of this in your own site to fit your own integration needs.

The majority of the functionality exists in model methods and is designed to be consumed throughout your site, in whatever way integration makes sense for you.

Furthermore, all data stored in django-stripe-payments is a local cache of what exists in Stripe and is fetchable via the Stripe API. Even if you were to truncate all data in django-stripe-payments, there is no real data loss as you can pull it all again and load up your models. The important ones have sync methods to do this for you. Sync methods haven't been written for all models (yet).

Models

EventProcessingException

This is a simple model that is just a log of any exceptions encountered while processing events. It's really not designed to be consumed directly other than being able to view exceptions in the admin or querying the database.

The *Event* model is the only object that uses this model.

Event

An event is a webhook message sent from Stripe to the webhook url you set up.

The message is captured in a view, saved to the model. It is then validated and processed. Processing only occurs if the messages is indeed validated. It's only valid if the event content pulled by id from Stripe matches what was received

at the webhook endpoint. This is designed to prevent people from discovering your endpoint url and posting fake data to it.

Processing the event is where all the other models get populated in django-stripe-payments. Not all message types are processed, but all messages do send a signal so that you can handle specific messages in your site if you care about something that django-stripe-payments isn't capturing.

The *Event* model is the work horse of django-stripe-payments, however, it is not intended to be used directly either.

Transfer

The *Transfer* model stores records of Stripe transfers into your bank account.

It's a read only object but does have a special manager that provides some aggregates:

    Transfer.objects.during(year, month) Transfer.objects.paid_totals_for(year, month)

The first, *during*, will return a list of transfers for a given year and month, while the second, *paid_totals_for*, will return a list of aggregates:

- Total Gross
- Total Net
- Total Charge Fees
- Total Adjustment Fees
- Total Refunds
- Total Refund Fees
- Total Validation Fees
- Total Amount

TransferChargeFee

Stores details about each fee associated with a particular *Transfer*.

Customer

The *Customer* object maps to a customer record in Stripe and has a nullable foreign key to a *User* object. It's nullable because you maybe delete a user from your site but would likely want/need to keep financial record history.

There are lots of public API methods on *Customer* intances that are likely of some interest.

purge()

This method will set the user foreign key to *None* and set the card details to blank. The model's *delete()* method is overridden so that it calls *purge()* instead of destorying the instance in the database as the rest of the models in django-stripe-payments that have financial details are tied to the *Customer*.

Even if the customer object at this point is anonymous, it is important to not break the chain of financial details that might be important to you for various aggregates and other reporting needs.

CurrentSubscription

Invoice

InvoiceItem

Charge

1.1.3 Requirements

The models and management commands should work without any additional requirements, but if you want to use the templates, especially the templates involving forms that ship with this app, then you will need `django-forms-bootstrap` and `eldarion-ajax`.

Add `django-forms-bootstrap` to your project:

```
pip install django-forms-bootstrap
```

You will also want to add `django_forms_bootstrap` to your *INSTALLED_APPS*':

```
INSTALLED_APPS = [
    "django_forms_bootstrap",
]
```

Add `eldarion-ajax.js` to your base template. eldarion-ajax is a library used by the included templates to interact with the ajax views. You can certainly write your own javascript, but it is likely easier for you to just include the library.

Get it here: eldarion-ajax

1.1.4 Installation

- To install

```
pip install django-stripe-payments
```

- Add `'payments'` to your `INSTALLED_APPS` setting:

```
INSTALLED_APPS = [
    "payments",
]
```

- Add `'payments'` to your `urls.py`

```
urlpatterns = patterns("",
    ...
    url(r"^payments/", include("payments.urls")),
    ...
)
```

1.1.5 Configuration (Modifications to *settings.py*)

- Add entries to for your publishable and secret keys. The recommended method is to setup your production keys using environment variables. This helps to keep them more secure. Your test keys can be displayed in your code directly.

The following entries look for your STRIPE_PUBLIC_KEY and STRIPE_SECRET_KEY in your environment and, if it can't find them, uses your test keys values instead:

```
STRIPE_PUBLIC_KEY = os.environ.get("STRIPE_PUBLIC_KEY", "<your publishable test␣
↪key>")
STRIPE_SECRET_KEY = os.environ.get("STRIPE_SECRET_KEY", "<your secret test key>")
```

- Setup your payment plans by defining the setting `PAYMENTS_PLANS`:

```
PAYMENTS_PLANS = {
    "monthly": {
        "stripe_plan_id": "pro-monthly",
        "name": "Web App Pro ($25/month)",
        "description": "The monthly subscription plan to WebApp",
        "price": 25,
        "currency": "usd",
        "interval": "month"
    },
    "yearly": {
        "stripe_plan_id": "pro-yearly",
        "name": "Web App Pro ($199/year)",
        "description": "The annual subscription plan to WebApp",
        "price": 199,
        "currency": "usd",
        "interval": "year"
    },
    "monthly-trial": {
        "stripe_plan_id": "pro-monthly-trial",
        "name": "Web App Pro ($25/month with 30 days free)",
        "description": "The monthly subscription plan to WebApp",
        "price": 25,
        "currency": "usd",
        "interval": "month",
        "trial_period_days": 30
    },
}
```

- If you're using Stripe to send email receipts for you, set `SEND_EMAIL_RECEIPTS = False` (and configure your emails from your Stripe Account Settings). Alternatively `payments` can send them for you - you'll want to set `PAYMENTS_INVOICE_FROM_EMAIL` to the email address that your receipts will be sent from.

1.1.6 Static Media

The included templates have been tested to work with Checkout.

An example of integrating Checkout is to put this in your base template:

```
<script src="//checkout.stripe.com/v2/checkout.js"></script>
<script>
    $(function() {
        $('body').on("click", '.change-card, .subscribe-form button[type=submit]',␣
↪function(e) {
          e.preventDefault();
          var $form = $(this).closest("form"),
              token = function(res) {
                $form.find("input[name=stripe_token]").val(res.id);
                $form.trigger("submit");
```

```
            };

          StripeCheckout.open({
            key:         $form.data("stripe-key"),
            name:        'Payment Method',
            panelLabel:  'Add Payment Method',
            token:       token
          });

          return false;
        });
    });
</script>
```

1.1.7 Signals

There are a handful of application level signals as well as a 1:1 signal for webhook that Stripe sends to allow you to respond to whatever activity Stripe sends as a webhook.

cancelled

providing_args *stripe_response*

card_changed

providing_args *stripe_response*

subscription_made

providing_args *plan*, *stripe_response*

webhook_processing_error

providing_args *data*, *exception*

WEBHOOK_SIGNALS

This is a dictionary, indexed by the names for each webhook event. The signal object found as the value for each item in the dictionary provides an *event* argument the the instance of *payments.models.Event* that has all the data related to the event.

1.1.8 Template Tags

There are two inclusion tags to make it easy to put various forms within your templates.

change_plan_form

template payments/_change_plan_form.html

context *form*, which is an instance of the *payments.forms.ChangePlanForm*

subscribe_form

template payments/_subscribe_form.html

context *form*, which is an instance of the *payments.forms.SubscribeForm*; *plans*, which is the *settings.PAYMENTS_PLANS* dictionary

1.1.9 Usage

What you do with payments and subscriptions is a highly custom thing so it is pretty hard to write a generic integration guide, but one typical thing you might want to do is disable access to most of the site if the subscription fails being active. You can accomplish this by adding the *payments.middleware.ActiveSubscriptionMiddleware* to your *settings.py*:

```
MIDDLEWARE_CLASSES = [
    ...
    "payments.middleware.ActiveSubscriptionMiddleware",
    ...
]
```

There are two settings you'll need to define for this middleware to work. The first, `SUBSCRIPTION_REQUIRED_EXCEPTION_URLS` is a list of url names that the user can access no matter what, and the second one, `SUBSCRIPTION_REQUIRED_REDIRECT` is the url to redirect them to if they hit a pay-only page.

Of course, your site might function more on levels and limits rather than lockout. It's up to you to write the necessary code to interpret how your site should behave, however, you can rely on `request.user.customer` giving you an object with relevant information to make that decision such as `customer.plan` and `customer.has_active_subscription`.