django-user-payments Documentation

Release 0.3.3

Feinheit AG

Mar 31, 2022

Contents

Version 0.3.3

Create, track and settle payments by users.

django-user-payments consists of a few modules which help with managing payments and subscriptions by users on a Django-based site.

CHAPTER 1

Table of Contents

1.1 Prerequisites and installation

The prerequisites for django-user-payments are:

- Django 2.2 or better
- Python 3.5 or better
- django-mooch (installed as a dependency)

To install the package, start with installing the package using pip:

```
pip install django-user-payments
```

Add the apps to `INSTALLED_APPS` and override settings if you want to change the defaults:

```
INSTALLED_APPS = [
    ...

    # Required:
    "user_payments",

    # Optional, if you want those features:
    # "user_payments.user_subscriptions",
    # "user_payments.stripe_customers",

    ...
]

# Also optional, defaults:
from datetime import timedelta  # noqa

USER_PAYMENTS = {
    "currency": "CHF",
```

(continues on next page)

(continued from previous page)

```
    "grace_period": timedelta(days=7),
    "disable_autorenewal_after": timedelta(days=15),
}
```

1.2 Payments

django-user-payments allows quickly adding line items for a user and paying later for those.

For example, if some functionality is really expensive you might want to add a line item each time the user requests the functionality:

```
@login_required
def expensive_view(request):
    LineItem.objects.create(
        user=request.user,
        amount=Decimal("0.05"),
        title="expensive view at %s" % timezone.now(),
    )

    # .. further processing and response generation
```

At the time the user wants to pay the costs that have run up you create a pending payment and maybe process it using a moocher.

A quick introduction to moochers

Moochers (provided by django-mooch) take a request and a payment instance, show a form or a button, and handle interaction with and responses from payment service providers. They allow processing individual payments one at a time.

django-user-payments' `Payment` model extends the abstract `mooch.Payment` so that moochers may be readily used.

The first view below, `pay` creates the pending payment and redirects the user to the next step. `pay_payment` fetches the pending payment from the database and allows selecting a payment method. Further processing is the responsibility of the selected moocher.

```
@login_required
def pay(request):
    payment = Payment.objects.create_pending(user=request.user)
    if not payment:
        # No line items, redirect somewhere else!
        return ...

    # django-mooch's Payment uses UUID4 fields:
    return redirect('pay_payment', id=payment.id.hex)


@login_required
def pay_payment(request, id):
    payment = get_object_or_404(
        request.user.user_payments.pending(),
        id=id,
```

(continues on next page)

(continued from previous page)

```
    )
    return render(request, "pay_payment.html", {
        "payment": payment,
        "moochers": [
            moocher.payment_form(request, payment)
            for moocher in moochers.values()
        ],
    })
```

1.2.1 A payment life cycle

Payments will most often be created by calling `Payment.objects.create_pending(user=<user>)`. This creates an unpaid payment instance and binds all unbound line items to the payment instance by updating their `payment` foreign key field. The `amount` fields of all line items are summed up and assigned to the payments' `amount` field. If there were no unbound line items, no payment instance is created and the manager method returns `None`.

Next, the instance is hopefully processed by a moocher or django-user-payment's processing which will be discussed later. A paid-for payment has its nullable `charged_at` field (among some other fields) set to the date and time of payment.

If payment or processing failed for some reason, the payment instance is in most cases not very useful anymore. Deleting the instance directly fails because the line items' `payment` foreign key protects against cascading deletion. Instead, `payment.cancel_pending()` unbinds the line items from the payment and deletes the payment instance.

1.2.2 Undoing payments

In rare cases it may even be necessary to undo a payment which has already been marked as paid, respectively has its `charged_at` field set to a truthy value. In this case, the `payment.undo()` method sets `charged_at` back to `None` and unbinds all the payments' line items.

1.3 Subscriptions

Active subscriptions periodically create periods, which in turn create line items.

Creating a subscription for an user looks like this:

```
subscription = Subscription.objects.ensure(
    user=request.user,
    code="the-membership",
    periodicity="monthly",
    amount=Decimal("12"),

    # Optional:
    title="The Membership",
)
```

This example would also work with `Subscription.objects.create()`, but `Subscription.objects.ensure()` knows to update a users' subscription with the same `code` and also is smart when a subscription is updated – it does not only set fields, but also remove now invalid periods and delay the new start date if the subscription is still paid for.

django-user-payments' subscriptions have no concept of a plan or a product – this is purely your responsibility to add (if needed).

1.3.1 Periods and periodicity

Next, let's add some periods and create some line items for them:

```
for period in subscription.create_periods():
    period.create_line_item()
```

Subscriptions are anchored on the `starts_on` day. Available periodicities are:

- `yearly`
- `monthly`
- `weekly`
- `manually`

Simply incrementing the month and year will not always work in the case of `yearly` and `monthly` periodicity. If the naively calculated date does not exist, the algorithm returns later dates.

Specifics of recurring date calculation

For example, if a subscription starts on 2016-02-29 (a leap year), the next three years' periods will start on March 1st. However, the period stays anchored at the start date, therefore in 2020 the period starts on February 29th again. Same with months: The next two period starts for a monthly subscription starting on 2018-03-31 will be 2018-05-01 and 2018-05-31. As you can see, since 2018-04-31 does not exist, no period starts in April, and two periods start in May.

Periods end one day before the next period starts. Respectively, subscriptions do not only offer date fields – all date fields have a corresponding property returning a date time in the default timezone. Periods always start at 00:00:00 and end at 23:59:59.999999.

`subscription.create_periods()` only creates periods that start no later than today. This can be overridden by passing a date using the `until` keyword argument.

1.3.2 Subscription status and grace periods

Once line items created by subscription periods are bound to a payment and the payment is paid for, the subscription automatically runs its `update_paid_until()` method. The method sets the subscriptions' `paid_until` date field to the date when the latest subscription period ends.

However, the subscription status is not only determined by `paid_until`. By default, subscriptions have a grace period of 7 days during which the subscription is still `subscription.is_active`, but also `subscription.in_grace_period`. The date time when the grace period ends is available as `subscription.grace_period_ends_at`.

Take note that the grace period also applies to subscriptions that have been newly created, that is, never been paid for.

Subscriptions should be canceled by calling `subscription.cancel()`. This method disabled automatic renewal and removes periods and their line items in case they haven't been paid for yet.

1.3.3 Periodical tasks and maintenance

The following commands would make sense to run periodically in a management command:

- `Subscription.objects.disable_autorenewal()`: Cancel subscriptions that are past due by `disable_autorenewal_after` days, by default 15 days.
- `Subscription.objects.create_periods()`: Run `subscription.create_periods()` on all subscriptions that should renew automatically.
- `SubscriptionPeriod.objects.create_line_items()`: Make periods create their line items in case they haven't done so already. By default only periods that start no later than today are considered. This can be changed by providing another date using the `until` keyword argument.

The processing documentation contains a management command where those functions are called in the recommended way and order.

1.3.4 Closing notes

As you can see subscriptions do not concern themselves with payment processing, only with creating line items. Subscriptions only use payments to automatically update their `paid_until` date field.

1.4 Stripe customers

The Stripe customers module offers a moocher which automatically creates a Stripe customer, a model which binds Stripe customers to user instances and a processor for payments.

Note: Stripe supports more than one source (that is, credit card) per customer, but our `user_payments.stripe_customers` module does not.

The Stripe customers app requires `STRIPE_PUBLISHABLE_KEY` and `STRIPE_SECRET_KEY` settings.

1.4.1 The moocher

The `user_payments.stripe_customers.moochers.StripeMoocher` is basically a drop-in replacement for django-mooch's `mooch.stripe.StripeMoocher`, except that:

- Instead of only charging the user once, our moocher creates a Stripe customer and binds it to a local Django user (in case the user is authenticated) to make future payments less cumbersome.
- If an authenticated user already has a Stripe customer, the moocher only shows basic credit card information (e.g. the brand and expiry date) and a "Pay" button instead of requiring entry of all numbers again.

1.5 Processing

django-user-payments comes with a framework for processing payments outside moochers.

The general structure of an individual processor is as follows:

```
from user_payments.processing import Result

def psp_process(payment):
    # Check prerequisites
    if not <prerequisites>:
        return Result.FAILURE

    # Try settling the payment
    if <success>:
        return Result.SUCCESS

    return Result.FAILURE
```

The processor **must** return a `Result` enum value. Individual processor results **must** not be evaluated in a boolean context.

The following `Result` values exist:

- `Result.SUCCESS`: Payment was successfully charged for.
- `Result.FAILURE`: This processor failed, try the next.
- `Result.TERMINATE`: Terminate processing for this payment, do not run any further processors.

When using `process_payment()` as you should (see below) and an individual processor raises exceptions the exception is logged, the payment is canceled if `cancel_on_failure` is `True` (the default) and the exception is reraised. In other words: Processors should **not** raise exceptions.

1.5.1 Writing your processors

django-user-payments does not bundle any processors, but makes it relatively straightforward to write your own.

The Stripe customers processor

This processors' prerequisites are a Stripe customer instance. If the prerequisites are fulfilled, this processor tries charging the user, and if this fails, sends an error mail to the user and terminates further processing:

```
import json
import logging

from django.apps import apps
from django.core.mail import EmailMessage
from django.db.models import ObjectDoesNotExist
from django.utils import timezone

import stripe

from user_payments.processing import Result


logger = logging.getLogger(__name__)


def with_stripe_customer(payment):
    try:
        customer = payment.user.stripe_customer
```

(continues on next page)

(continued from previous page)

```
    except ObjectDoesNotExist:
        return Result.FAILURE

    s = apps.get_app_config("user_payments").settings
    try:
        charge = stripe.Charge.create(
            customer=customer.customer_id,
            amount=payment.amount_cents,
            currency=s.currency,
            description=payment.description,
            idempotency_key="charge-%s-%s" % (payment.id.hex, payment.amount_cents),
        )

    except stripe.error.CardError as exc:
        logger.exception("Failure charging the customers' card")
        EmailMessage(str(payment), str(exc), to=[payment.email]).send(
            fail_silently=True
        )
        return Result.TERMINATE

    else:
        payment.payment_service_provider = "stripe"
        payment.charged_at = timezone.now()
        payment.transaction = json.dumps(charge)
        payment.save()

        return Result.SUCCESS
```

A processor which sends a "Please pay" mail

This processor always fails, but sends a mail to the user first that they should please pay soon-ish:

```
from django.core.mail import EmailMessage

from user_payments.processing import Result


def please_pay_mail(payment):
    # Each time? Each time!
    EmailMessage(str(payment), "<No body>", to=[payment.email]).send(fail_
↪silently=True)
    return Result.FAILURE
```

Since this processor runs its action before returning a failure state, it only makes sense to run this one last.

1.5.2 Processing individual payments

The work horse of processing is the `user_payments.processing.process_payment` function. The function expects a payment instance and a list of processors and returns `True` if one of the individual processors returned a `Result.SUCCESS` state.

If all processors fail the payment is automatically canceled and the payments' line items returned to the pool of unbound line items. This can be changed by passing `cancel_on_failure=False` in case this behavior is undesirable.

1.5.3 Bulk processing

The `user_payments.processing` module offers the following functions to bulk process payments:

- `process_unbound_items(processors=[...])`: Creates pending payments for all users with unbound line items and calls `process_payment` on them. Cancels payments if no processor succeeds.
- `process_pending_payments(processors=[...])`: Runs all unpaid payments through `process_payment`, but does not cancel a payment upon failure. When you're only using processors and no moochers this function *should* have nothing to do since `process_unbound_items` always cleans up on failure. Still, it's better to be safe than sorry and run this function too.

1.5.4 Management command

My recommendation is to write a management command that is run daily and which processes unbound line items and unpaid payments. An example management command follows:

```
from django.core.management.base import BaseCommand

from user_payments.processing import process_unbound_items, process_pending_payments
# Remove this line if you're not using subscriptions:
from user_payments.user_subscriptions.models import Subscription, SubscriptionPeriod

# Import the processors defined above
from yourapp.processing import with_stripe_customer, please_pay_mail


processors = [with_stripe_customer, please_pay_mail]


class Command(BaseCommand):
    help = "Create pending payments from line items and try settling them"

    def handle(self, **options):
        # Remove those three lines if you're not using subscriptions:
        Subscription.objects.disable_autorenewal()
        Subscription.objects.create_periods()
        SubscriptionPeriod.objects.create_line_items()

        # Process payments
        process_unbound_items(processors=processors)
        process_pending_payments(processors=processors)
```

If you're using Sentry you probably want to wrap all commands in a `try..except` block:

```
...

from raven.contrib.django.raven_compat.models import client

class Command(BaseCommand):
    ...

    def handle(self, **options):
        try:
            ...
        except Exception:
```

(continues on next page)

(continued from previous page)

```
            client.captureException()
            raise  # Reraise, for good measure
```

1.6 Change log

1.6.1 Next version

- Ensured that the username is part of `search_fields` for all models registered with the admin interface.
- Added a new subscription periodicity, `quarterly`.
- Added a `SubscriptionPeriod.objects.zeroize_pending_periods()` helper for zeroizing past periods so that when users (finally) provide payment methods they do not have to pay past periods too (if you choose so).

1.6.2 0.3 (2018-09-21)

- Fixed the case where two consecutive `Subscription.objects.ensure()` calls would lead to the subscription being restarted and a second period being added right away. Also, fix a bunch of other edge cases in `ensure()` and add a few additional tests while at it.
- Made it impossible to inadvertently delete subscription periods by cascading deletions when removing line items.
- Changed the subscription admin to only show the period inline when updating a subscription.
- Added `Payment.undo()` to undo payments which have already been marked as paid.
- Fixed an edge case where setting `Subscription.paid_until` would produce incorrect results when no period was paid for yet.

1.6.3 0.2 (2018-08-05)

- Changed `SubscriptionPeriod.objects.create_line_items()` to only create line items for periods that start no later than today by default. A new `until` keyword argument allows overriding this.
- Fixed `MANIFEST.in` to include package data of `stripe_customers`.
- Changed the code for the updated Stripe Python library. Updated the requirement for `django-user-payments[stripe]` to `>=2`.
- Fixed a crash when creating a subscription with a periodicity of "manually" through the admin interface.

1.6.4 0.1 (2018-06-05)

- First release that should be fit for public consumption.