django-payments

Release 0.14.0.post50+gcbf872d

Sep 13, 2021

Contents

Contents:

CHAPTER 1

Installation

1. Install django-payments

```
$ pip install django-payments
```

Note that some providers have additional dependencies. For example, if using stripe, you should run:

```
$ pip install "django-payments[stripe]"
```

1. Add `payments` to your `INSTALLED_APPS`.
2. Add the callback processor to your URL router:

```
# urls.py
from django.conf.urls import include, path

urlpatterns = [
    path('payments/', include('payments.urls')),
]
```

3. Define a `Payment` model by subclassing `payments.models.BasePayment`:

```
# mypaymentapp/models.py
from decimal import Decimal

from payments import PurchasedItem
from payments.models import BasePayment

class Payment(BasePayment):

    def get_failure_url(self):
        return 'http://example.com/failure/'

    def get_success_url(self):
        return 'http://example.com/success/'
```

(continues on next page)

(continued from previous page)

```
    def get_purchased_items(self):
        # you'll probably want to retrieve these from an associated order
        yield PurchasedItem(name='The Hound of the Baskervilles', sku='BSKV',
                            quantity=9, price=Decimal(10), currency='USD')
```

The `get_purchased_items()` method should return an iterable yielding instances of `payments.PurchasedItem`.

4. Write a view that will handle the payment. You can obtain a form instance by passing POST data to `payment.get_form()`:

```
# mypaymentapp/views.py
from django.shortcuts import get_object_or_404, redirect
from django.template.response import TemplateResponse
from payments import get_payment_model, RedirectNeeded

def payment_details(request, payment_id):
    payment = get_object_or_404(get_payment_model(), id=payment_id)
    try:
        form = payment.get_form(data=request.POST or None)
    except RedirectNeeded as redirect_to:
        return redirect(str(redirect_to))
    return TemplateResponse(request, 'payment.html',
                            {'form': form, 'payment': payment})
```

Note: Please note that `Payment.get_form()` may raise a `RedirectNeeded` exception.

5. Prepare a template that displays the form using its *action* and *method*:

```
<!-- templates/payment.html -->
<form action="{{ form.action }}" method="{{ form.method }}">
    {% csrf_token %}
    {{ form.as_p }}
    <p><input type="submit" value="Proceed" /></p>
</form>
```

6. Configure your `settings.py`:

```
# settings.py
INSTALLED_APPS = [
    # ...
    'payments']

PAYMENT_HOST = 'localhost:8000'
PAYMENT_USES_SSL = False
PAYMENT_MODEL = 'mypaymentapp.Payment'
PAYMENT_VARIANTS = {
    'default': ('payments.dummy.DummyProvider', {})}
```

Variants are named pairs of payment providers and their configuration.

Note: Variant names are used in URLs so it's best to stick to ASCII.

Note: PAYMENT_HOST can also be a callable object.

CHAPTER 2

Making a payment

1. Create a `Payment` instance:

```
from decimal import Decimal

from payments import get_payment_model

Payment = get_payment_model()
payment = Payment.objects.create(
    variant='default',  # this is the variant from PAYMENT_VARIANTS
    description='Book purchase',
    total=Decimal(120),
    tax=Decimal(20),
    currency='USD',
    delivery=Decimal(10),
    billing_first_name='Sherlock',
    billing_last_name='Holmes',
    billing_address_1='221B Baker Street',
    billing_address_2='',
    billing_city='London',
    billing_postcode='NW1 6XE',
    billing_country_code='GB',
    billing_country_area='Greater London',
    customer_ip_address='127.0.0.1')
```

2. Redirect the user to your payment handling view.

2.1 Payment amounts

The `Payment` instance provides two fields that let you check the total charged amount and the amount actually captured:

```
>>> payment.total
Decimal('181.38')

>>> payment.captured_amount
Decimal('0')
```

2.2 Payment statuses

A payment may have one of several statuses, that indicates its current state. The status is stored in `status` field of your `Payment` instance. Possible statuses are:

`waiting` Payment is waiting for confirmation. This is the first status, which is assigned to the payment after creating it.

`input` Customer requested the payment form and is providing the payment data.

`preauth` Customer has authorized the payment and now it can be captured. Please remember, that this status is only possible when the `capture` flag is set to `False` (see *Authorization and capture* for details).

`confirmed` Payment has been finalized or the the funds were captured (when using `capture=False`).

`rejected` The payment was rejected by the payment gateway. Inspect the contents of the `payment.message` and `payment.extra_data` fields to see the gateway response.

`refunded` Payment has been successfully refunded to the customer (see *Refunding a payment* for details).

`error` An error occurred during the communication with the payment gateway. Inspect the contents of the `payment.message` and `payment.extra_data` fields to see the gateway response.

2.3 Fraud statuses

Some gateways provide services used for fraud detection. You can check the fraud status of your payment by accessing `payment.fraud_status` and `payment.fraud_message` fields. The possible fraud statuses are:

`unknown` The fraud status is unknown. This is the default status for gateways, that do not involve fraud detection.

`accept` Fraud was not detected.

`reject` Fraud service detected some problems with the payment. Inspect the details by accessing the `payment.fraud_message` field.

`review` The payment was marked for review.

CHAPTER 3

Refunding a payment

If you need to refund a payment, you can do this by calling the `refund()` method on your `Payment` instance:

```
>>> from payments import get_payment_model
>>> Payment = get_payment_model()
>>> payment = Payment.objects.get()
>>> payment.refund()
```

By default, the total amount would be refunded. You can perform a partial refund, by providing the `amount` parameter:

```
>>> from decimal import Decimal
>>> payment.refund(amount=Decimal(10.0))
```

Note: Only payments with the `confirmed` status can be refunded.

CHAPTER 4

Authorization and capture

Some gateways offer a two-step payment method known as Authorization & Capture, which allows you to collect the payment manually after the buyer has authorized it. To enable this payment type, you have to set the `capture` parameter to `False` in the configuration of payment backend:

```
# settings.py
PAYMENT_VARIANTS = {
    'default': ('payments.dummy.DummyProvider', {'capture': False})}
```

4.1 Capturing the payment

To capture the payment from the buyer, call the `capture()` method on the `Payment` instance:

```
>>> from payments import get_payment_model
>>> Payment = get_payment_model()
>>> payment = Payment.objects.get()
>>> payment.capture()
```

By default, the total amount will be captured. You can capture a lower amount, by providing the `amount` parameter:

```
>>> from decimal import Decimal
>>> payment.capture(amount=Decimal(10.0))
```

Note: Only payments with the `preauth` status can be captured.

4.2 Releasing the payment

To release the payment to the buyer, call the `release()` method on your `Payment` instance:

```
>>> from payments import get_payment_model
>>> Payment = get_payment_model()
>>> payment = Payment.objects.get()
>>> payment.release()
```

Note: Only payments with the `preauth` status can be released.

CHAPTER 5

Provided backends

These are the payment provider implementations included in this package. Note that you should not usually instantiate these yourself, but use `provider_factory()` instead.

5.1 Dummy

class payments.dummy.**DummyProvider**(*capture=True*)

Dummy payment provider.

This is a dummy backend suitable for testing your store without contacting any payment gateways. Instead of using an external service it will simply show you a form that allows you to confirm or reject the payment.

Example:

```
PAYMENT_VARIANTS = {
    'dummy': ('payments.dummy.DummyProvider', {})}
```

5.2 Authorize.Net

class payments.authorizenet.**AuthorizeNetProvider**(*login_id*, *transaction_key*, *endpoint='https://test.authorize.net/gateway/transact.dll'*, ***kwargs*)

Payment provider for Authorize.Net.

This backend implements payments using the Advanced Integration Method (AIM) from Authorize.Net.

This backend does not support fraud detection.

Parameters

- **login_id** – Your API Login ID assigned by Authorize.net
- **transaction_key** – Your unique Transaction Key assigned by Authorize.net

- **endpoint** – The API endpoint to use. For the production environment, use `'https://secure.authorize.net/gateway/transact.dll'` instead.

Example:

```
# use staging environment
PAYMENT_VARIANTS = {
    'authorizenet': ('payments.authorizenet.AuthorizeNetProvider', {
        'login_id': '1234login',
        'transaction_key': '1234567890abcdef',
        'endpoint': 'https://test.authorize.net/gateway/transact.dll'})}
```

5.3 Braintree

Example:

```
# use sandbox
PAYMENT_VARIANTS = {
    'braintree': ('payments.braintree.BraintreeProvider', {
        'merchant_id': '112233445566',
        'public_key': '1234567890abcdef',
        'private_key': 'abcdef123456',
        'sandbox': True})}
```

5.4 Coinbase

class payments.coinbase.**CoinbaseProvider**(*key*, *secret*, *endpoint='sandbox.coinbase.com'*, ***kwargs*)

Payment provider for coinbase.

This backend implements payments using Coinbase.

This backend does not support fraud detection.

Parameters

- **key** – Api key generated by Coinbase
- **secret** – Api secret generated by Coinbase
- **endpoint** – Coinbase endpoint domain to use. For the production environment, use `'coinbase.com'` instead

__init__(*key*, *secret*, *endpoint='sandbox.coinbase.com'*, ***kwargs*)

Create a new provider instance.

This method should not be called directly; use `provider_factory()` instead.

Example:

```
# use sandbox
PAYMENT_VARIANTS = {
    'coinbase': ('payments.coinbase.CoinbaseProvider', {
        'key': '123abcd',
        'secret': 'abcd1234',
        'endpoint': 'sandbox.coinbase.com'})}
```

5.5 Cybersource

Example:

```
# use sandbox
PAYMENT_VARIANTS = {
    'cybersource': ('payments.cybersource.CyberSourceProvider', {
        'merchant_id': 'example',
        'password': '1234567890abcdef',
        'capture': False,
        'sandbox': True})}
```

5.5.1 Merchant-Defined Data

Cybersource allows you to pass Merchant-Defined Data, which is additional information about the payment or the order, such as an order number, additional customer information, or a special comment or request from the customer. This can be accomplished by passing your data to the `Payment` instance:

```
>>> payment.attrs.merchant_defined_data = {'01': 'foo', '02': 'bar'}
```

5.6 Dotpay

class payments.dotpay.**DotpayProvider**(*seller_id*, *pin*, *endpoint='https://ssl.dotpay.pl/test_payment/'*, *channel=0*, *channel_groups=None*, *ignore_last_payment_channel=False*, *lang='pl'*, *lock=False*, *type=2*, ***kwargs*)

Payment provider for dotpay.pl

This backend implements payments using a popular Polish gateway, Dotpay.pl.

Due to API limitations there is no support for transferring purchased items.

This backend does not support fraud detection.

Parameters

- **seller_id** – Seller ID assigned by Dotpay
- **pin** – PIN assigned by Dotpay
- **channel** – Default payment channel (consult reference guide). Ignored if channel_groups is set.
- **channel_groups** – Payment channels to choose from (consult reference guide). Overrides channel.
- **lang** – UI language
- **lock** – Whether to disable channels other than the default selected above
- **endpoint** – The API endpoint to use. For the production environment, use `'https://ssl.dotpay.pl/'` instead
- **ignore_last_payment_channel** – Display default channel or channel groups instead of last used channel.
- **type** – Determines what should be displayed after payment is completed (consult reference guide).

Example:

```
# use defaults for channel and lang but lock available channels
PAYMENT_VARIANTS = {
    'dotpay': ('payments.dotpay.DotpayProvider', {
        'seller_id': '123',
        'pin': '0000',
        'lock': True,
        'endpoint': 'https://ssl.dotpay.pl/test_payment/'})}
```

5.7 PayPal

class payments.paypal.**PaypalProvider**(*client_id*, *secret*, *endpoint='https://api.sandbox.paypal.com'*, *capture=True*)

Payment provider for Paypal, redirection-based.

This backend implements payments using PayPal.com.

Parameters

- **client_id** – Client ID assigned by PayPal or your email address
- **secret** – Secret assigned by PayPal
- **endpoint** – The API endpoint to use. For the production environment, use `'https://api.paypal.com'` instead
- **capture** – Whether to capture the payment automatically. See *Authorization and capture* for more details.

Example:

```
# use sandbox
PAYMENT_VARIANTS = {
    'paypal': ('payments.paypal.PaypalProvider', {
        'client_id': 'user@example.com',
        'secret': 'iseedeadpeople',
        'endpoint': 'https://api.sandbox.paypal.com',
        'capture': False})}
```

class payments.paypal.**PaypalCardProvider**(*client_id*, *secret*, *endpoint='https://api.sandbox.paypal.com'*, *capture=True*)

Payment provider for Paypal, form-based.

This backend implements payments using PayPal.com but the credit card data is collected by your site.

Parameters are the same as *PaypalProvider*.

This backend does not support fraud detection.

Example:

```
PAYMENT_VARIANTS = {
    'paypal': ('payments.paypal.PaypalCardProvider', {
        'client_id': 'user@example.com',
        'secret': 'iseedeadpeople'})}
```

5.8 Sage Pay

Example:

```
# use simulator
PAYMENT_VARIANTS = {
    'sage': ('payments.sagepay.SagepayProvider', {
        'vendor': 'example',
        'encryption_key': '1234567890abcdef',
        'endpoint': 'https://test.sagepay.com/Simulator/VSPFormGateway.asp'})}
```

5.9 Sofort / Klarna

Example:

```
PAYMENT_VARIANTS = {
    'sage': ('payments.sofort.SofortProvider', {
        'id': '123456',
        'key': '1234567890abcdef',
        'project_id': '654321',
        'endpoint': 'https://api.sofort.com/api/xml'})}
```

5.10 Stripe

Example:

```
# use sandbox
PAYMENT_VARIANTS = {
    'stripe': ('payments.stripe.StripeProvider', {
        'secret_key': 'sk_test_123456',
        'public_key': 'pk_test_123456'})}
```

5.11 MercadoPago

Example:

```
PAYMENT_VARIANTS: = {
    "mercadopago": (
        "payments.mercadopago.MercadoPagoProvider",
        {
            "access_token": "APP_USR-345345363464444-645434-
↪7f8da79f8da7f98ad7f98ad7f98df78e-454545466",
            "sandbox": DEBUG,
        },
    ),
}
```

Note that the API sandbox does not return Payment details, so all payments will seem unpaid.

Index

Symbols

A

C

D

P