django-helcim Documentation

Joshua Robert Torrance

Sep 11, 2020

Contents

Django Helcim provides a Django-based integration with the Helcim Commerce API. You can view the source code on GitHub.

CHAPTER 1

Getting Started

Note: These instructions make a distinction between making direct calls to the Helcim Commerce API versus using Helcim.js. Within this documentation, "Helcim API" refers to direct calls, where as "Helcim.js" will refere to calls via Helcim.js.

1.1 Install django-helcim and its Dependencies

Install `django-helcim` from PyPI. It is strongly recommended you use a virtual environment for your projects. For example, you can do this easily with Pipenv:

```
$ pipenv install django-helcim
```

If you are integrating this package with Django Oscar, you will need to install this package as well:

```
$ pipenv install django-oscar
```

Attention: If using Django Oscar, it is assumed you will manage the proper configuration. If needed, you can follow the setup instructions for Django Oscar.

Make sure to add `django-helcim` to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'helcim',
    ...
]
```

1.2 Edit your Django Settings

Once `django-helcim` has been installed, you will need to update your Django settings. You will need to specify the required settings depending on the workflow you are using.

For the Helcim API, you will need to specify the following settings (at a minimum): `HELCIM_API_URL`, `HELCIM_ACCOUNT_ID`, `HELCIM_API_TOKEN`, and `HELCIM_TERMINAL_ID`.

For Helcim.js, you will need to specify the `HELCIM_JS_CONFIG` setting (at a minimum).

There are several other settings available to configure `django-helcim`. In most cases, defaults are applied automatically when a setting is not specified. Where possible, the package defaults to the most restrictive value possible (given the security requirements of financial data). A full list of settings you can configure is available on the *Settings page*.

1.3 Add URLs

There are a couple views you can use to review and manage various aspects of the `django-helcim` models. You can enable these by updating the *required settings* and adding the package URLs:

```
from django.urls import path

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

1.4 Final Details

Django Helcim also supplies some credit card logos to display with saved credit cards. To make use of these you will need to run `collectstatic`:

```
$ pipenv run python manage.py collectstatic
```

1.5 Next Steps

Once `django-helcim` is installed, you will need to integrate it into your payment workflow. This process will vary significantly between applications, but this application provides some standard objects and methods to streamline the process.

Working examples of integrations can be found in GitHub repo sandbox directory. You will find two working Django applications. The `oscar` sandbox is a working example of integration with Django Oscar. The `scratch` sandbox shows a minimal example of integration with a generic service requiring payment processing and demonstrates both the Helcim API and Helcim.js workflows.

See the *Sandbox page* for additional details on setting up and configuring the sandbox sites.

CHAPTER 2

Examples

The following are examples of how to integrate `django-helcim` into a service requiring payment processing. These are not intended to be comprehensive, but should highlight the major functionality.

More robust examples can be found within the sandbox sites. Details of these can be can be found on the *Sandbox page*.

You can also find explicit details of supported functionality and allowable options for all the `django-helcim` functions and classes on the *package documentation pages*.

Note: These instructions make a distinction between making direct calls to the Helcim Commerce API versus using Helcim.js. Within this documentation, "Helcim API" refers to direct calls, where as "Helcim.js" will refere to calls via Helcim.js.

2.1 Basic Helcim API Calls

You will most likely integrate `django-helcim` into your application during a POST request to submit payment information. There is typically a significant amount of handling to properly collect payment information and ensure a secure user experience; this example will focus exclusively on how you would integrate with `django-helcim` and not touch on the finer considerations of implementing a service that requires payment processing.

To streamline collection of payment information, you will probably utilize a Django form class.

```
# forms.py

from django import forms

class PaymentForm(forms.Form):
    cc_name = forms.CharField(max_length=128)
    cc_number = forms.CharField(max_length=16)
    cc_expiry = forms.CharField(max_length=4)
```

(continues on next page)

(continued from previous page)

```
    cc_cvv = forms.CharField(max_length=4)
    amount = forms.DecimalField(decimal_places=2)
```

Next you will need some view to receive the form POST request and integrate with `django-helcim`.

```
# views.py
form django.views.generic.edit import FormView

from helcim.exceptions import ProcessingError, PaymentError
from helcim.gateway import Purchase

from .forms import PaymentForm

class PaymentView(FormView):
    form_class = PaymentForm
    template_name = 'example_app/example_template.html'

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)

        if form.is_valid():
            # Prepare data for API submission with a django-helcim
            # gateway class
            purchase = Purchase(
                save_token=True,
                django_user=self.request.user,
                cc_name=form.cleaned_data['cc_name'],
                cc_number=form.cleaned_data['cc_number'],
                cc_expiry=form.cleaned_data['cc_expiry'],
                cc_cvv=form.cleaned_data['cc_cvv'],
                amount=form.cleaned_data['amount'],
            )

            # Attempt to make the API calls
            try:
                transaction, token = purchase.process()
            except ValueError as error:
                # Add handling for ValuError
                pass
            except ProcessingError as error:
                # Add handling for ProcessingError
                # These would usually be server-side errors and
                # not suitable for end user display
                pass
            except PaymentError as error
                # Add handling for PaymentError
                # These are some error that has caused payment to
                # fail (e.g. expired card, incorrect card number)
                pass

            # Add any final handling before redirecting user
            return self.form_valid()

        # Some error has occured during validation
        return self.form_invalid()
```

Note: This is a generic form and form view implementation; you will need to customize to your specific use case.

2.1.1 Gateways

`django-helcim` provides a variety of `gateway` classes to help streamline Helcim API calls. These classes represent the types of calls you can make to the API. Currently you can do the following:

- `Purchase()`: a purchase or sale API call
- `Preauthorize()`: a preauthorization API call
- `Capture()`: a capture API call
- `Refund()`: a refund API call
- `Verification()`: a verification or card tokenization API call

These gateways will allow you a consistent way to make API calls in a Python-based manner and not worry about data conversions and API authentication. Data will undergo some basic validation to ensure there are no type or format errors, but will not undergo any validation to ensure the call succeeds (e.g. you will not be notified that the credit card is expired or that you are missing details that the Helcim API requires).

2.2 Helcim.js Calls

Helcim.js provides a way to reduce your risk and security requirements when it comes to managing credit card data. In short, Helcim.js will intercept any payment calls to your server and instead direct them to the Helcim API server for processing directly. You will then receive an API response to your call.

This is a significantly different workflow than a normal Helcim API call, so `django-helcim` provides an additional `gateway` class to assist with these workflows.

You will likely still require some standard way to collect payment information (e.g. a Django form or coding the form manually in a Django template). This is not covered extensively in this example as this is more a requirement of Helcim.js than `django-helcim`. If you need assistance with this, it is recommended you view the Developer Documentation on the Helcim website.

To assist with Helcim.js implementation, `django-helcim` provides a mixin that lets you declare Helcim.js configuration details within your Django settings and then add them to a template via the context variable.

```
# settings.py

DJANGO_HELCIM_CONFIG = {
    'purchase': {
        'url': 'https://myhelcim.com/js/version2.js',
        'token': 'abdefg',
    },
    'preauthorization': {
        'url':  'https://myhelcim.com/js/version2.js',
        'token': 'hijklmnop',
    }
}
```

```
# views.py
form django.views.generic.edit import FormView

from helcim.mixins import HelcimJSMixin

from .forms import PaymentForm

class HelcimJSPaymentView(HelcimJSMixin, FormView):
    form_class = PaymentForm
    template_name = 'example_app/example_template.html'
```

```
<!-- example_template.html -->
<!-- SCRIPT -->
<script type="text/javascript" src="{{ helcim_js.purchase.url }}"></script>

<!-- FORM -->
<form name="helcimForm" id="helcimForm" action="" method="POST">
    <input type="hidden" id="token" value="{{ helcim_js.purchase.token }}">
</form>
```

Once you have made Helcim.js call, you will receive the response for handling. You will need to handle both error responses (e.g. expired credit card) and success calls. `django-helcim` provides a helper class to manage these responses and create the relevant `django-helcim` model instances.

```
# views.py
form django.views.generic.edit import FormView

from helcim.gateway import HelcimJSResponse

from .forms import PaymentForm

class PaymentView(FormView):
    # FormView used to allow a Django form to be used to help
    # with templating; it is not actually required for handling
    # the Helcim.js response
    form_class = PaymentForm
    template_name = 'example_app/example_template.html'

    def post(self, request, *args, **kwargs):
        response = HelcimJSResponse(
            response=request.POST,
            save_token=True,
            django_user=request.user,
        )

        # If response is valid, can save details and redirect
        if response.is_valid():
            transaction, token = response.record_purchase()

            # form_valid() used to trigger a success URL redirect
            return self.form_valid()

        # Invalid form submission/payment - render payment details
        # again. Could manage various error responses here for a
        # better user experience
        form = self.get_form()
```

(continues on next page)

(continued from previous page)

```
        return self.form_invalid(form)
```

The `HelcimJSResponse` class has three methods that can be used to provide the action supported by Helcim.js. Currently these are:

- `record_purchase()`: a purchase or sale call
- `record_preauthorization()`: a preauthorization API call
- `record_verification()`: a verification or card tokenization API call

CHAPTER 3

Sandbox Websites

The sandbox websites provide working examples of how to integrate `django-helcim` into a project. There are two sandboxes to demonstrate two potential use cases: integration with Django Oscar and integration into a generic service requiring payment processing.

Attention: These instructions assume you have already setup a pipenv virtual environment with `django-helcim` installed. See the *Getting started* page if you need additional instructions.

3.1 Scratch Sandbox

The Scratch Sandbox website is a generic example service that collects payment details for processing by Helcim. It includes two examples: the Helcim API workflow and the Helcim.js workflow.

3.1.1 Deploying the site

First You will need to create your own copy of the `config.env` file. This file contains some basic `django` and `django-helcim` settings to get the sandbox to work. A template config file can be found at `sandbox/.config.env`. Copy and rename this file to `config.env` and update the relevant settings as needed for your sandbox (e.g. your Helcim API and/or Helcim.js details).

Next, you will need to run the Django migrations:

```
$ pipenv run python sandbox/manage.py migrate
```

You can now start your site through the standard Django commands and access it at http://127.0.0.1:8000/:

```
$ pipenv run python sandbox/manage.py runserver
```

If needed, you can create a superuser account with the standard management command:

```
$ pipenv run python sandbox/manage.py createsuperuser
```

You can create regular user accounts by running the sandbox sever and creating it with the web form: http://127.0.0.1:8000/accounts/login/.

Tip: If you need to restart your site from scratch, delete the `db.sqlite3` file and complete the above steps again.

3.2 Oscar Sandbox

The Oscar Sandbox website is a barebones Django Oscar store that demonstrates how to use *django-helcim* with Django Oscar and handle payments via the Helcim API.

3.2.1 Deploying the site

You will need to create your own copy of the `config.env` file. This file contains some basic `django` and `django-helcim` settings to get the sandbox to work. A template config file can be found at `sandbox/.config.env`. Copy and rename this file to `config.env` and update the relevant settings as needed for your sandbox (e.g. your Helcim API details).

You should then be able to run the Django migrations:

```
$ pipenv run python sandbox/manage.py migrate
```

Next you will need to load country data (see the Django Oscar page for more details):

```
$ pipenv run python sandbox/manage.py oscar_populate_countries
```

Next, import a basic catalogue of store items to test with:

```
$ pipenv run python sandbox/manage.py oscar_import_catalogue sandbox/fixtures/
↪catalogue.csv
```

Finally, collect all the static files for the site:

```
$ pipenv run python sandbox/manage.py collectstatic
```

You can now start your site through the standard Django commands and access it at http://127.0.0.1:8000/:

```
$ pipenv run python sandbox/manage.py runserver
```

If needed, you can create a superuser account with the standard management command:

```
$ pipenv run python sandbox/manage.py createsuperuser
```

You can create regular user accounts by running the sandbox sever and creating it with the web form: http://127.0.0.1:8000/accounts/login/.

Tip: If you need to restart your site from scratch, delete the `db.sqlite3` file and complete the above steps again.

CHAPTER 4

Settings

Below is a comprehensive list of all the settings for Django Helcim.

4.1 API settings

These are settings that control interactions with the Helcim Commerce API.

4.1.1 HELCIM_API_URL

Required: `False`

Default (string): `https://secure.myhelcim.com/api/`

The URL to access the Helcim Commerce API. At this time there is only access to the API through the default URL, but this setting is available to handle any future situations where custom endpoints become available or to allow users to quickly update their application should the URL change before this package is updated.

4.1.2 HELCIM_ACCOUNT_ID

Required: `True`

Default (string): `''`

The account ID for your Helcim account.

4.1.3 HELCIM_API_TOKEN

Required: `True`

Default (string): `''`

The API token generated on your Helcim Commerce API dashboard to allow `django-helcim` to make transactions via the Helcim Commerce API.

4.1.4 HELCIM_TERMINAL_ID

Required: `False`

Default (string): `''`

The Helcim terminal ID you are using. If not provided the Helcim Commerce API will use the default terminal for the provided Account ID.

4.1.5 HELCIM_API_TEST

Required: `False`

Default (boolean): `False`

A flag declaring whether transactions should be submitted in test mode or not. When set to *True* all transactions will have `test=true` added to the POST data. This prevents the Helcim Commerce API from attempting to process the transaction.

4.1.6 HELCIM_JS_CONFIG

Required: `False`

Default (dictionary): `{}`

A dictionary that allows you to declare your Helcim.js configuration details within your Django settings. This dictionary is validated and made available in your view context via the `HelcimJSMixin` mixin. The dictionary requires the following format:

```
{
  'identifier': {
    'url': 'url-to-your-helcim-js-script',
    'token' 'your-helcim-js-token',
    'test': True,
  }
}
```

Once configured, you can add the `HelcimJSMixin` to the required views and access the details in your templates. Below is summary of the keys and mixin funtionality:

Settings Key	Description	Mixin Usage
`identifier`	An identifier used in templates to reference these configuration details.	`helcim_js.identifier`
`url`	URL to the Helcim.js file. Can be an empty string if you will serve the JS file yourself.	`helcim_js.identifier.url`
`token`	The Helcim.js token	`helcim_js.identifier.token`
`test`	Optional value; add key with a `truthy` value to enable. If enabled, will output the HTML input to trigger the Helcim.js test mode. Otherwise will be empty string.	‘‘helcim_js.identifier.test_input‘

4.2 Private data storage settings

These are settings that control what kind of private data is stored in your database. `django-helcim` does not record the Primary Account Number (PAN), but does give you the option to save select data that could be used to identify a specify customer and their account. **You should only store the minimum amount of data you need to reduce the risk and severity of a data breach.**

4.2.1 HELCIM_REDACT_ALL

Required: `False`

Default (boolean): `False`

If set to `True`, all references to sensitive cardholder information will be redacted. **This setting applies to and overrides any of the individual settings below.**

4.2.2 HELCIM_REDACT_CC_NAME

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card cardholder name.

4.2.3 HELCIM_REDACT_CC_NUMBER

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card number.

4.2.4 HELCIM_REDACT_CC_EXPIRY

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card expiry date.

4.2.5 HELCIM_REDACT_CC_CVV

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card CVV.

4.2.6 HELCIM_REDACT_CC_TYPE

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card type.

4.2.7 HELCIM_REDACT_CC_MAGNETIC

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card magnetic strip data.

4.2.8 HELCIM_REDACT_CC_MAGNETIC_ENCRYPTED

Required: `False`

Default (boolean): `True`

If set to `True`, redacts all reference to the credit card magnetic strip data and the terminal serial number.

4.2.9 HELCIM_REDACT_TOKEN

Required: `False`

Default (boolean): `False`

If set to `True`, redacts all reference to the Helcim Commerce credit card token and the 'first four last four' digits of the credit card number.

Note: This setting will not override the **Helcim Token Vault**. If you to turn off the vault, use the `HELCIM_ENABLE_TOKEN_VAULT` setting.

4.3 Helcim Transaction Functionality

These settings allow you to enable or disable additional functionality with the HelcimTransaction model.

4.3.1 HELCIM_ENABLE_TRANSACTION_CAPTURE

Required: `False`

Default (boolean): `False`

If set to `True`, will allow you to capture transactions from the `HelcimTransactionDetailView`.

4.3.2 HELCIM_ENABLE_TRANSACTION_REFUND

Required: `False`

Default (boolean): `False`

If set to `True`, will allow you to refund transactions from the `HelcimTransactionDetailView`.

4.4 Helcim Token Vault Functionality

4.4.1 HELCIM_ENABLE_TOKEN_VAULT

Required: `False`

Default (boolean): `False`

If set to `True`, enables the Helcim card token vault. This stores the card token returned from the Helcim Commerce API, along with the customer code. The token will also be associated to the logged in user.

4.5 Admin Functionality

A read-only admin view is available to assist with viewing data or debugging.

4.5.1 HELCIM_ENABLE_ADMIN

Required: `False`

Default (boolean): `False`

If set to `True`, will register the read-only admin views.

4.6 Other Settings

4.6.1 HELCIM_ASSOCIATE_USER

Required: `False`

Default (boolean): `True`

Specifies whether a django user model should be associated to `HelcimTransaction` and `HelcimToken` model instances. By default, the logged in user is added to all transactions and tokens. This can be turned off by setting this to `False`.

CHAPTER 5

Contributing

Contributions or forking of the project is always welcome. Below will provide a quick outline of how to get setup and things to be aware of when contributing.

5.1 Reporting issues

If you simply want to report an issue, you can use the GitHub Issue page.

5.2 Setting up the development environment

This package is built using Pipenv, which will take care of both your virtual environment and package management. If needed, you can install `pipenv` through `pip`:

```
$ pip install pipenv
```

To download the repository from GitHub via `git`:

```
$ git clone git://github.com/studybuffalo/django-helcim.git
```

You can then install all the required dependencies by changing to the package directory and installing from `Pipfile.lock`:

```
$ cd django-helcim
$ pipenv install --ignore-pipfile --dev
```

Finally, you will need to build the package:

```
$ pipenv run python setup.py develop
```

You should now have a working environment that you can use to run tests and setup the sandbox demo.

5.3 Testing

All pull requests must have unit tests built and must maintain or increase code coverage. The ultimate goal is to achieve a code coverage of 100%. While this may result in some superfluous tests, it sets a good minimum baseline for test construction.

5.3.1 Testing format

All tests are built with the pytest framework (and pytest-django for Django-specific components). There are no specific requirements on number or scope of tests, but at a bare minimum there should be tests to cover all common use cases. Wherever possible, try to test the smallest component possible.

5.3.2 Running Tests

You can run all tests with the standard `pytest` command:

```
$ pipenv run py.test
```

To check test coverage, you can use the following:

```
$ pipenv run py.test --cov=helcim --cov-report=html
```

You may specify the output of the coverage report by changing the `--cov-report` option to `html` or `xml`.

5.4 Sandbox

You may find it easier to do testing within the sandbox enviroments. See the *Sandbox page* for additional details on setting up and configuring the sandbox sites if you haven't already done so as part of the `django-helcim` installation.

5.5 Updating documentation

All documentation is hosted on Read the Docs and is built using Sphinx. All the module content is automatically built from the docstrings and the sphinx-apidoc tool and the sphinxcontrib-napoleon extension.

5.5.1 Docstring Format

The docstrings of this package follow the Google Python Style Guide wherever possible. This ensures proper formatting of the documentation generated automatically by Sphinx. Additional examples can be found on the Sphinx napoleon extension documentation.

5.5.2 Building package reference documentation

The content for the Package reference is built using the `sphinx-apidoc` tool. If any files are added or removed from the `helcim` module you will need to rebuild the `helcim.rst` file for the changes to populate on Read the Docs. You can do this with the following command:

```
$ pipenv run sphinx-apidoc -fTM -o docs helcim helcim/migrations helcim/urls.py␣
↪helcim/apps.py helcim/admin.py
```

5.5.3 Linting documentation

If you are having issues with the ReStructuredText (reST) formatting, you can use `rst-lint` to screen for syntax errors. You can run a check on a file with the following:

```
$ pipenv run rst-lint /path/to/file.rst
```

5.6 Distributing package

Django Helcim is designed to be distributed with PyPI. While most contributors will not need to worry about uploading to PyPI, the following instructions list the general process in case anyone wishes to fork the repository or test out the process.

Note: It is recommended you use TestPyPI to test uploading your distribution while you are learning and seeing how things work. The following examples below will use TestPyPI as the upload target.

To generate source archives and built distributions, you can use the following:

```
$ pipenv run python setup.py sdist bdist_wheel
```

To upload the distributions, you can use the following `twine` commands:

```
$ pipenv run twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

You will need to provide a PyPI username and password before the upload will start.

CHAPTER 6

helcim package

6.1 Submodules

6.2 helcim.bridge_oscar module

Bridging module between Django Oscar and gateway module.

class helcim.bridge_oscar.**BaseCardTransactionBridge**(*amount*, *token_id=None*, *card=None*, *billing_address=None*, *save_token=False*, *django_user=None*, *customer_code=None*)

Bases: object

Base class to bridge Oscar and Helcim transactions.

Parameters

- **order_number** (*str*) – Order number for the transaction.
- **amount** (*dec*) – The transaction total.
- **card** (*obj*) – Instance of the Oscar bankcard class.
- **billing_address** (*dict*) – The billing address information.
- **save_token** (*bool*) – Whether the card token should be saved in the token vault or not.
- **django_user** (*obj*) – The user to associate with the saved card token.
- **customer_code** (*str*) – The Helcim customer code to associate with the saved card token.

class helcim.bridge_oscar.**CaptureBridge**(*transaction_id*)

Bases: object

Class to bridge Oscar and Helcim Capture transactions.

Parameters **transaction_id** (*int*) – the Helcim preauthorization transaction ID to capture.

process()
Attempts to process Capture with transaction details.

Returns the values of `gateway.Capture.process`.

Raises

- `GatewayError` – An Oscar error raised when there was an error with the payment API.
- `PaymentError` – An Oscar error raised when there was an error processing the payment.

class helcim.bridge_oscar.**PreauthorizeBridge**(*amount*, *token_id=None*, *card=None*, *billing_address=None*, *save_token=False*, *django_user=None*, *customer_code=None*)
Bases: *helcim.bridge_oscar.BaseCardTransactionBridge*

Class to bridge Oscar and Helcim preauthorization transactions.

process()
Attempts to process Preauthorize with transaction details.

Returns the values of `gateway.Preauthorize.process`.

Raises

- `GatewayError` – An Oscar error raised when there was an error with the payment API.
- `PaymentError` – An Oscar error raised when there was an error processing the payment.

class helcim.bridge_oscar.**PurchaseBridge**(*amount*, *token_id=None*, *card=None*, *billing_address=None*, *save_token=False*, *django_user=None*, *customer_code=None*)
Bases: *helcim.bridge_oscar.BaseCardTransactionBridge*

Class to bridge Oscar and Helcim purchase transactions.

process()
Attempts to process Purchase with transaction details.

Returns the values of `gateway.Purchase.process`.

Raises

- `GatewayError` – An Oscar error raised when there was an error with the payment API.
- `PaymentError` – An Oscar error raised when there was an error processing the payment.

class helcim.bridge_oscar.**RefundBridge**(*amount*, *token_id=None*, *card=None*, *billing_address=None*, *save_token=False*, *django_user=None*, *customer_code=None*)
Bases: *helcim.bridge_oscar.BaseCardTransactionBridge*

Class to bridge Oscar and Helcim refund transactions.

process()
Attempts to process Refund with transaction details.

Returns the values of `gateway.Refund.process`.

Raises

- `GatewayError` – An Oscar error raised when there was an error with the payment API.
- `PaymentError` – An Oscar error raised when there was an error processing the payment.

class helcim.bridge_oscar.**VerificationBridge**(*amount*, *token_id=None*, *card=None*, *billing_address=None*, *save_token=False*, *django_user=None*, *customer_code=None*)

Bases: *helcim.bridge_oscar.BaseCardTransactionBridge*

Class to bridge Oscar and Helcim Verification transactions.

process()

Attempts to process Verification with transaction details.

Returns the values of `gateway.Verification.process`.

Raises

- `GatewayError` – An Oscar error raised when there was an error with the payment API.
- `PaymentError` – An Oscar error raised when there was an error processing the payment.

helcim.bridge_oscar.**remap_oscar_billing_address**(*address*)

Remaps Oscar billing address dictionary to Helcim dictionary.

Parameters **address** (*dict*) – A dictionary of the billing address details provided by Django Oscar.

Returns

Billing address details formatted for django-helcim.

Return type dict

helcim.bridge_oscar.**remap_oscar_credit_card**(*card*)

Remaps Oscar credit card object as Helcim dictionary.

Parameters **card** (*obj*) – A credit card object provided by Django Oscar.

Returns Credit card details formatted for django-helcim.

Return type dict

helcim.bridge_oscar.**retrieve_saved_tokens**(*django_user=None*, *customer_code=None*)

Shortcut for retrieve_saved_tokens from the Gateway module.

Added as a convenience to allow access to core functions via the bridge module exclusively.

helcim.bridge_oscar.**retrieve_token_details**(*token_id*, *django_user=None*, *customer_code=None*)

Shortcut for retrieve_token_details from the Gateway module.

Added as a convenience to allow access to core functions via the bridge module exclusively.

6.3 helcim.conversions module

Process and validates data to and from the Helcim API.

class helcim.conversions.**Field**(*api_name*, *field_type='s'*, *min_length=None*, *max_length=None*)

Bases: `object`

A single API field.

Parameters

- **api_name** (*str*) – The Helcim API field name for the field.
- **field_type** (*str*) – The field type: `s` (string), `c` (decimal), `i` (integer), `b` (boolean), `d` (date), or `t` (time).
- **min** (*int, optional*) – The minimum field length.
- **max** (*int, optional*) – The maximum field length.

helcim.conversions.**convert_helcim_response_fields**(*fields*, *field_dictionary*)

Converts provided Helcim response to Python data types.

Handles both API responses and Helcim.js response.

Parameters

- **fields** (*dict*) – Helcim response fields.
- **field_dictionary** (*dict*) – dictionary of field references to use for data type conversion.

Returns converted fields as Python data types.

Return type dictionary

helcim.conversions.**create_f4l4**(*cc_number*)

Creates the token_f4l4 value if a CC number is provided.

Parameters cc_number (str): The Helcim response CC number.

Returns the token F4L4 value.

Return type string

helcim.conversions.**create_raw_request**(*data*)

Converts the raw request into a POST string.

Parameters **data** (*dict*) – The POST request data dictionary.

Returns The POST request data as a formatted string.

Return type str

helcim.conversions.**process_api_response**(*response*, *raw_request=None*, *raw_response=None*)

Updates API field names/types, and adds additional data.

Parameters

- **response** (*str*) – The API response (as an `OrderedDict`).
- **raw_request** (*str*) – Raw request (as string) submitted to API.
- **raw_response** (*str*) – Raw response (as string) returned by API.

Returns The validated and converted API response.

Return type dict

helcim.conversions.**process_helcim_js_response**(*response*)

Processes the Helcim.js response into a Python dictionary.

Parameters **response** (*dict*) – The Helicm.js POST response.

Returns The validated and converted Helcim.js response.

Return type dict

helcim.conversions.**process_request_fields**(*api*, *cleaned*, *additional=None*)
Converts all data to proper Helcim API request fields.

Parameters

- **api** (*dict*) – The API setting fields.
- **cleaned** (*dict*) – The cleaned POST data.
- **additional** – Any additional POST data to directly add (skips any conversion steps).

Returns The data ready for a POST request.

Return type dict

helcim.conversions.**validate_request_fields**(*details*)
Validates and coerces request field data prior to submission.

Uses the TO_API_FIELDS dictionary to determine type and required validation to apply.

Parameters **details** (*dict*) – The values to validated and coerce.

Returns The cleaned data (with the same keys as `details`).

Return type dict

Raises `ValueError` – Raised if data fails validation.

6.4 helcim.exceptions module

Custom exceptions for django-helcim.

exception helcim.exceptions.**DjangoError**
Bases: *helcim.exceptions.HelcimError*

Exception for issues relating to Django interface.

exception helcim.exceptions.**HelcimError**
Bases: `Exception`

Base exception for all package exceptions.

exception helcim.exceptions.**PaymentError**
Bases: *helcim.exceptions.HelcimError*

Exception to handle payments, pre-auths, and captures.

exception helcim.exceptions.**ProcessingError**
Bases: *helcim.exceptions.HelcimError*

Exception to handle system and API connection errors.

exception helcim.exceptions.**RefundError**
Bases: *helcim.exceptions.HelcimError*

Exception to handle refund errors.

exception helcim.exceptions.**VerificationError**
Bases: *helcim.exceptions.HelcimError*

Exception to handle errors during credit care verification.

6.5 helcim.gateway module

Interface functions with the Helcim Commerce API.

These functions provide an agonstic interface with the Helcim Commerce API and should work in any application.

class helcim.gateway.**BaseCardTransaction**(*save_token=False*, ***kwargs*)
Bases: *helcim.gateway.BaseRequest*

Base class for transactions involving credit card details.

determine_card_details()
Confirms valid payment details and updates self.cleaned.

Cycles through the provided details to determine the most appropriate payment method. If multiple methods provided, only the first match is returned (token > customer code > CC number > encrypted magnetic strip > magnetic strip).

Raises ValueError – No valid payment details provided.

class helcim.gateway.**BaseRequest**(*api_details=None*, *django_user=None*, ***kwargs*)
Bases: *helcim.mixins.ResponseMixin*

Base class to handle validation and submission to Helcim API.

Parameters

- **api_details** (*dict*) – Details to connect to Helcim API:
  - **url** (*str*): API URL.
  - **account_id** (*str*): Helcim account ID.
  - **token** (*str*): Helcim API token.
  - **terminal_id** (*str*): Helcim terminal ID.
- **django_user** (*obj*) – The Django model for the requesting user.
- ****kwargs** (*dict*) – Any additional transaction details.

Keyword Arguments

- **amount** (*dec, optional*) – The amount for the transaction.
- **currency** (*str, optional*) – The currency for the transaction.
- **cc_name** (*str, optional*) – Name of the credit cardholder.
- **cc_number** (*str, optional*) – 16 digit credit card number.
- **cc_expiry** (*str, optional*) – 4 digit (MMYY) credit card expiry.
- **cc_cvv** (*str, optional*) – 3 or 4 digit credit card CVV.
- **cc_address** (*str, optional*) – Address of the credit cardholder.
- **cc_postal_code** (*str, optional*) – Postal code/zip code of the credit cardholder.
- **customer_code** (*str, optional*) – Helcim customer code.
- **token** (*str, optional*) – 23 digit Helcim card token.
- **token_f4l4** (*str, optional*) – 8 digit "first four digits and last four digits" of the credit card number
- **token_f4l4_skip** (*bool, optional*) – Whether to skip the F4L4 verification.

- **mag** (*string, optional*) – Non-encrypted credit card magnetic strip data.
- **mag_enc** (*str, optional*) – Encrypted credit card magnestic strip data.
- **mag_enc_serial_number** (*string, optional*) – Terminal serial number.
- **order_number** (*str, optional*) – An assigned order number for the purchase.
- **ecommerce** (*bool, optional*) – Whether this is an e-commerce transaction or not.
- **comments** (*str, optional*) – Any additional comments with this transaction.
- **billing_contact_name** (*str, optional*) – Billing address contact name.
- **billing_business_name** (*str, optional*) – Billing address business name.
- **billing_street_1** (*str, optional*) – Billing street address (line 1).
- **billing_street_2** (*str, optional*) – Billing street address (line 2).
- **billing_city** (*str, optional*) – Billing city.
- **billing_province** (*str, optional*) – Billing province.
- **billing_country** (*str, optional*) – Billing country.
- **billing_postal_code** (*str, optional*) – Billing postal code.
- **billing_phone** (*str, optional*) – Billing phone number.
- **billing_fax** (*str, optional*) – Billing fax number
- **billing_email** (*str, optional*) – Billing email.
- **shipping_contact_name** (*str, optional*) – Shipping contact name.
- **shipping_business_name** (*str, optional*) – Shipping business name.
- **shipping_street_1** (*str, optional*) – Shipping street address (line 1).
- **shipping_street_2** (*str, optional*) – Shipping street address (line 2).
- **shipping_city** (*str, optional*) – Shipping city.
- **shipping_province** (*str, optional*) – Shipping province.
- **shipping_country** (*str, optional*) – Shipping country.
- **shipping_postal_code** (*str, optional*) – Shipping postal code.
- **shipping_phone** (*str, optional*) – Shipping phone number.
- **shipping_fax** (*str, optional*) – Shipping fax number.
- **shipping_email** (*str, optional*) – Shipping email address.
- **amount_shipping** (*dec, optional*) – Shipping cost.
- **amount_tax** (*dec, optional*) – Tax amount.
- **shipping_method** (*str, optional*) – Method of shipping.
- **tax_details** (*str, optional*) – Name for the tax (e.g. GST).
- **test** (*bool, optional*) – Whether this is a test transaction or not.

configure_test_transaction()

Adds test flag to post data if HELCIM_API_TEST is True.

Method applies to the cleaned data (not the raw POST data). If the test flag is declared in both the POST data and Django settings file, the POST data takes precedence.

post(*post_data=None*)
Makes POST to Helcim API and updates response attribute.

Parameters **post_data** (*dict*) – The parameters to pass with the POST request.

Raises ProcessingError – An error occurred connecting or communicating with Helcim API.

process_error_response(*response_message*)
Returns error response with proper exception type.

Parameters **response_message** (*str*) – The API error response message.

Raises

- PaymentError – Raised for any errors returned during a purchase or refund transaction.
- HelcimError – Raised for any other unhandled errors.

save_transaction(*transaction_type*)
Saves provided transaction data as Django model instance.

Parameters **transaction_type** (*str*) – The type of transaction (s for purchase (sale), p for pre-authorization, c for capture, and r for refund).

Returns A Django model instance of the saved data.

Return type obj

Raises DjangoError – Issue when attempting to save transaction to database.

set_api_details(*details*)
Sets the API details for this transaction.

Will either return a dictionary of the API details from the provided details argument, or will look to the Django settings file.

Parameters **details** (*dict*) – A dictionary of the API details.

Returns The proper API details from the provided data.

Return type dict

Raises ImproperlyConfigured – A required API setting is not found.

validate_fields()
Validates Helcim API request fields and coerces values.

class helcim.gateway.**Capture**(*api_details=None*, *django_user=None*, ***kwargs*)
Bases: *helcim.gateway.BaseRequest*

Makes a capture request (to complete a preauthorization).

process()
Completes a capture request.

validate_preauth_transaction()
Confirms that a preauth transaction ID was provided.

Raises PaymentError – Error when no transaction ID provided.

class helcim.gateway.**HelcimJSResponse**(*response*, *save_token=False*, *django_user=None*)
Bases: *helcim.mixins.ResponseMixin*

Class to handle Helcim.js Responses.

This is a helper class that takes a Helcim.js response, handles any errors, converts the response into Python types and applies any redactions.

Handling is more limited since all requests to the API are handled directly by Helcim.js. It is important to note that responses from Helcim.js are similar, but not identical to the basic Helcim API.

Parameters

- **response** (*obj*) – the Helcim.js response POST data.
- **save_token** (*bool*) – whether to save this to the `django-helcim` token vault.
- **django_user** (*obj*) – a django user instance to use with `django-helcim` model instances.

is_valid()
Validates format is correct and notifies of any errors.

Will also generate the redacted response at this point.

Returns True if valid without errors, otherwise false.

Return type boolean

record_preauthorization()
Saves validated preauthorization response to database.

Returns

tuple of the HelcimTransaction and HelcimToken instances. HelcimToken will be None if token is not saved.

record_purchase()
Saves validated purchase response to database.

Returns

tuple of HelcimTransaction and HelcimToken instances. HelcimToken will be None if token is not saved.

Return type tuple

record_verification()
Saves validated verification response to database.

Returns

tuple of the HelcimTransaction and HelcimToken instances. HelcimToken will be None if token is not saved.

class helcim.gateway.**Preauthorize**(*save_token=False*, ***kwargs*)
Bases: *helcim.gateway.BaseCardTransaction*

Makes a pre-authorization request to Helcim Commerce API.

process()
Makes a pre-authorization request.

class helcim.gateway.**Purchase**(*save_token=False*, ***kwargs*)
Bases: *helcim.gateway.BaseCardTransaction*

Makes a purchase request to Helcim Commerce API.

process()
Makes a purchase request.

Returns

The saved HelcimTransaction model instance and HelcimToken model.

Return type tuple

class helcim.gateway.**Refund**(*save_token=False*, ***kwargs*)
Bases: *helcim.gateway.BaseCardTransaction*

Makes a refund request.

process()
Makes a refund request to Helcim Commerce API.

class helcim.gateway.**Verification**(*save_token=False*, ***kwargs*)
Bases: *helcim.gateway.BaseCardTransaction*

Makes a verification request to Helcim Commerce API.

process()
Makes a verification request to Helcim Commerce API.

helcim.gateway.**retrieve_saved_tokens**(*django_user=None*, *customer_code=None*)
Returns list of tokens for specified customer.

Parameters

- **django_user** (*obj*) – A Django user model instance.
- **customer_code** (*str*) – A Helcim customer code.

Returns A queryset of the retrieved tokens

Return type obj

helcim.gateway.**retrieve_token_details**(*token_id*, *django_user=None*, *customer_code=None*)
Takes a HelcimToken ID and maps details to dictionary.

6.6 helcim.mixins module

Mixins to help support Helcim API interactions.

class helcim.mixins.**HelcimJSMixin**
Bases: object

Provides Helcim.js URL and token details in the view context.

This is a helper mixin that allows you to declare your Helcim.js configuration details within your Django settings. They are then injected into the view context to allow you to easily declare them within a template.

get_context_data(***kwargs*)
Overrides the view method to add Helcim.js details.

class helcim.mixins.**ResponseMixin**
Bases: object

Methods to support handling a Helcim repsonse.

Handles some data manipulations to prepare for saving to a model instance and applies relevant redactions to data.

create_model_arguments(*transaction_type*)
Creates dictionary for use as transaction model arguments.

Takes the redacted_response data and creates a dictionary that can be used as the keyword argumetns for the HelcimTransaction model.

Parameters **transaction_type** (*str*) – Transaction type for this transaction.

Returns dictionary of the HelcimTransaction arguments.

Return type dict

redact_data()
Removes sensitive and identifiable data.

By default will redact API fields and populate redacted_response attribute. Depending on Django settings, may also redact other fields in the formated and raw response.

save_token_to_vault()
Saves Helcim card token.

Returns

The HelcimToken model instance, or None (if model not created).

Return type obj

save_transaction(*transaction_type*)
Saves HelcimTransaction with redacted response details.

6.7 helcim.models module

Models for the django-helcim application.

class helcim.models.**HelcimToken**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

A Helcim card token.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

cc_expiry
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cc_name
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cc_type
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

customer_code
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_added
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

display_as_card_number
Displays token_f4l4 as a 16 character credit card number.

django_user
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

django_user_id

get_credit_card_png
Returns a path to a credit card .png for this token.

get_credit_card_svg
Returns a path to a credit card .svg for this token.

get_next_by_date_added(*, *field=<django.db.models.fields.DateTimeField: date_added>*, *is_next=True*, ***kwargs*)

get_previous_by_date_added(*, *field=<django.db.models.fields.DateTimeField: date_added>*, *is_next=False*, ***kwargs*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

token
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

token_f4l4
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

class helcim.models.**HelcimTransaction**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Details of a single Helcim transaction.

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

TRANSACTION_TYPES = (('s', 'purchase (sale)'), ('p', 'pre-authorization'), ('c', 'capt

amount
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

approval_code
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

avs_response
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

can_be_captured
Check if this transaction can be captured.

can_be_refunded
Check if this transaction can be refunded.

cc_expiry
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cc_name
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cc_number
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cc_type
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

currency
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

customer_code
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

cvv_response
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_created
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

date_response
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

django_user
Accessor to the related object on the forward side of a many-to-one or one-to-one (via ForwardOneToOneDescriptor subclass) relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`Child.parent` is a `ForwardManyToOneDescriptor` instance.

django_user_id

get_next_by_date_created(*, *field=<django.db.models.fields.DateTimeField: date_created>*, *is_next=True*, ***kwargs*)

get_next_by_date_response(*, *field=<django.db.models.fields.DateTimeField: date_response>*, *is_next=True*, ***kwargs*)

get_previous_by_date_created(*, *field=<django.db.models.fields.DateTimeField: date_created>*, *is_next=False*, ***kwargs*)

get_previous_by_date_response(*, *field=<django.db.models.fields.DateTimeField: date_response>*, *is_next=False*, ***kwargs*)

get_transaction_type_display(*, *field=<django.db.models.fields.CharField: transaction_type>*)

id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

notice
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

objects = <django.db.models.manager.Manager object>

order_number
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

raw_request
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

raw_response
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

response_message
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

token
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

token_f4l4
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

transaction_id
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

transaction_success
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

transaction_type
A wrapper for a deferred-loading field. When the value is read from this object the first time, the query is executed.

6.8 helcim.settings module

Determines relevant settings for django-helcim functioning.

helcim.settings.**determine_helcim_settings**()
Collects all possible django-helcim settings for easy use.

Performs basic validation of required settings and assigns defaults where applicable.

Returns Summary of all possible django-helcim settings.

Return type dict

6.9 helcim.views module

Views for Helcim Commerce API transactions.

class helcim.views.**TokenDeleteView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.views.generic.edit.DeleteView

Allows deletion of a Helcim API token.

context_object_name = 'token'

delete(*request*, **args*, ***kwargs*)
Override delete to allow success message to be added.

model
alias of *helcim.models.HelcimToken*

permission_required = 'helcim.helcim_tokens'

pk_url_kwarg = 'token_id'

raise_exception = True

success_message = 'Token successfully deleted.'

success_url

template_name = 'helcim/token_delete.html'

class helcim.views.**TokenListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.views.generic.list.ListView

List of all transactions submitted made by django-helcim.

context_object_name = 'tokens'

model
alias of *helcim.models.HelcimToken*

permission_required = 'helcim.helcim_tokens'

raise_exception = True

template_name = 'helcim/token_list.html'

class helcim.views.**TransactionDetailView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.views.generic.detail.DetailView

Details of a specific transaction made by django-helcim.

capture()
Processes a capture transaction.

context_object_name = 'transaction'

get_context_data(***kwargs*)
Insert the single object into the context dict.

model
alias of *helcim.models.HelcimTransaction*

permission_required = 'helcim.helcim_transactions'

pk_url_kwarg = 'transaction_id'

post(*request*, **args*, ***kwargs*)
Handles any capture and refund requests.

raise_exception = True

refund()
Processes a refund transaction.

template_name = 'helcim/transaction_detail.html'

class helcim.views.**TransactionListView**(***kwargs*)
Bases: django.contrib.auth.mixins.PermissionRequiredMixin, django.views.generic.list.ListView

List of all transactions submitted made by django-helcim.

context_object_name = 'transactions'

model
alias of *helcim.models.HelcimTransaction*

permission_required = 'helcim.helcim_transactions'

raise_exception = True

template_name = 'helcim/transaction_list.html'

CHAPTER 7

Changelog

7.1 Version 0 (Beta)

7.1.1 0.9.1 (2020-Apr-25)

Bug Fixes

- Marking the `helcim_js.identifier.test_input` as a Django `SafeString`, to allow usage without manual escpaing in templates.

7.1.2 0.9.0 (2020-Apr-23)

Feature Updates

- Adding new settings key under the `HELCIM_JS_CONFIG` setting. Can now include `test: True` for each Helcim.js configuration. When enabled, you can use `helcim_js.identifier.test_input` to enable or disable test mode within the Django template.

7.1.3 0.8.1 (2020-Apr-23)

Bug Fixes

- Fixing bug with the Helcim token "First 4 Last 4" characters saving; Helcim.js response may include whitespace in credit card number, so this needs to be stripped out.

7.1.4 0.8.0 (2020-Apr-21)

Feature Updates

- Switching the `HELCIM_ASSOCIATE_USER` setting to `HELCIM_ALLOW_ANONYMOUS`. This will allow clearer support for anonymous transactions.
- Adding support for Helcim.js.
  - Provides new `gateway` class which can be used to manage Helcim.js responses, perform data conversion into Python data types, complete validations, complete redactions, and save details to the `HelcimTransaction` and `HelcimToken` models.
  - Updates allowable settings to allow declaration of a `HELCIM_JS_CONFIG` setting. This setting will allow declaration of Helcim.js configuration details within the Django settings.
  - Provides new mixin for Django Views. Updates view context to include the `HELCIM_JS_CONFIG` details, allowing streamlined declaration of configuration details in Django templates.
- `django-helcim` will no longer require specifying Helcim API details in the settings, as either the API or Helcim.js workflow can be used with the package.
- Adding a new sandbox site that shows how Django Helcim can be integrated into a project from scratch. This site demonstrates both the Helcim API workflow and the Helcim.js workflow.
- Docs updated with new details on integrating `django-helcim` with a Django project.

Bug Fixes

- Add verify/verification (`v`) as an allowable transaction type for the `HelcimTransaction` model. Other calls in package were expecting this type.

7.1.5 0.7.1 (2020-Apr-05)

Bug Fixes

- Fixed issue with `__str__` of transaction model in Django admin when timezones were used for `datetimes`.

7.1.6 0.7.0 (2019-Dec-15)

Feature Updates

- Removing official support for Django 2.1 (has reached end of life).
- Removing Tox from testing. Too many conflicting issues and CI system can handle this better now.

7.1.7 0.6.0 (2019-Nov-17)

Feature Updates

- Adding a Django user reference to the `HelcimTransaction` model.
- Reworking the settings around associating a user model to the django-helcim models. Now all settings are controlled by the `HELCIM_ASSOCIATE_USER` setting.

7.1.8 0.5.0 (2019-Nov-16)

Feature Updates

- Adding cardholder name and CC expiry to token model and saving details with token.
- Updating related_name of token model to reduce naming conflicts.

7.1.9 0.4.0 (2019-Sep-07)

Feature Updates

- Updating the Oscar components for Oscar 2.0 compatibility.
- Sandbox updated to accomodate Oscar changes.

7.1.10 0.3.1 (2018-Nov-25)

Bug Fixes

- Fixed bug where Helcim Commerce API may return a blank response for a string field, resulting in coercion of `None` to `'None'`.

7.1.11 0.3.0 (2018-Nov-24)

Feature Updates

- Added new functions to `gateway` module to:
  - retrieve a user's saved Helcim tokens;
  - retrieve details of a single Helcim token; and
  - manage settings (this will verify that any required settings declared and will provide defaults for all other settings as appropriate).
- Extended the `bridge_oscar` module to:
  - streamline validating a Helcim token for payment processing by Django Oscar;
  - handle whether a token should be associated with a `django_user` instance or a Helcim `customer_code`; and
  - provide convenience shortcut functions/dictionaries to the `bridge_oscar` module to access functionality in the `gateway` module.
- Updated the `TO_API` dictionary to remove `order_number` (not recognized by the Helcim Commerce API).
- Updated sandbox site to demonstrate a workflow that makes use of the Token Vault.
- Updated HelcimToken model to:
  - record credit card type;
  - display the "first 4 last 4" digits of the credit card number as a 16 character string; and
  - retrieve and display an image for the corresponding credit card type.

Bug Fixes

- Fixed the `refund` and `capture` views to make use of the proper settings (were still using the outdated `HELCIM_TRANSACTIONS_READ_ONLY` setting).

7.1.12 0.2.2 (2018-Nov-17)

Bug Fixes

- Removing unnecessary `app_name` from urls which may cause namespace issues.

7.1.13 0.2.1 (2018-Nov-17)

Bug Fixes

- HTML template files are now included in package when installed via PyPI.

7.1.14 0.2.0 (2018-Nov-14)

Feature Updates

- Added the HelcimToken model (AKA the "Token Vault") - allows saving of Helcim Commerce card tokens to allow re-use in later transactions.
- Made functions for HelcimTransaction views more configurable - can now choose exactly which features to enable and disable.
- Reworked settings to default to more conservative values. Thinks will generally be disabled or redacted by default, but can be enabled as needed.
- Added redaction settings for credit card CVV, credit card magnetic strip data, and encrypted credit card magnetic strip data.
- Improved sandbox functioning to make setting up a new demo site easier.

Bug Fixes

- Fixed issues where the `HELCIM_REDACT_ALL` flag was not overriding properly.
- Updated redaction functions to accommodate all known fields containing cardholder information.
- Fixed issue where POST requests were returning byte-data rather than string data.

7.1.15 0.1.2 (2018-Nov-08)

Feature Updates

- Adding DepreciationWarning for Django 1.11 in anticipation of eventual end of support in 2020.
- Adding Tox test environments for all combinations of supported Python and Django versions.

7.1.16 0.1.1 (2018-Nov-07)

Bug Fixes

- Properly specifying dependencies for PyPI installation and updating Pipfile.

7.1.17 0.1.0 (2018-Nov-03)

Feature Updates

- Initial package release
- Supports basic API functions: purchase (sale), pre-authorization, capture, refund
- Basic `django-oscar` support with the bridge module.

Python Module Index

h

Index

F

G

H

I

M

N

O

P

R

S

T

V