django-oscar-wfrs Documentation

Release 0.18.0.post6

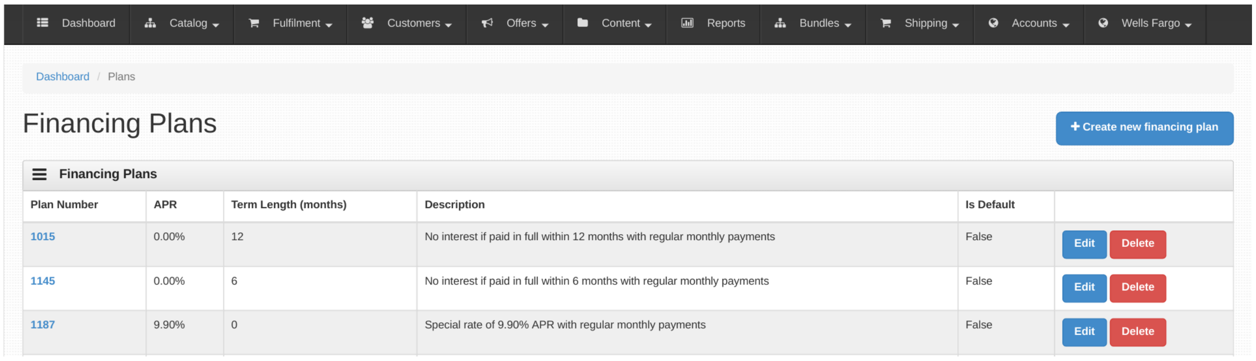
Craig Weber

Jul 19, 2019

CONTENTS

An extension on-top of django-oscar-api-checkout to allow interfacing with Wells Fargo Retail Services.

Full Documentation: https://django-oscar-wfrs.readthedocs.io



Plan Number	APR	Term Length (months)	Description	Is Default	
1015	0.00%	12	No interest if paid in full within 12 months with regular monthly payments	False	Edit Delete
1145	0.00%	6	No interest if paid in full within 6 months with regular monthly payments	False	Edit Delete
1187	9.90%	0	Special rate of 9.90% APR with regular monthly payments	False	Edit Delete

CHAPTER

ONE

CONTENTS

1.1 Concepts

Before installation, there are several concepts that are important to understand in regards to how Wells Fargo Retail Services works.

1.1.1 Credit Application

A credit application is a form filled out by a customer or for a customer when applying for a new account with Wells Fargo. The application data is submitted to Wells Fargo Retail Services and is either Approved or Denied. If approved, the response will include an Account Number.

1.1.2 Account Number

An account number is a 16 digit long number resembling a credit card number. It uniquely identifies a Wells Fargo account and can be used to authorize or charge payments to the account or to lookup information about the account, such as the credit limit, current balance, payment due, last payment, date, etc.

1.1.3 Transfer

Whenever a transaction is made on an account, this library records metadata about the transaction in the `TransferMetadata` model. This metadata includes the user who performed the transaction, the last 4 digits of the customer account number (in plain text), the full customer account number (in an encrypted blob), the plan number, amount, legal disclosure, etc.

1.1.4 Legal Disclosure

Whenever an authorization or charge is performed on a customs account, Wells Fargo returns a messages that must be displayed to the user who owns the account. The messages generally looks something like this.

> REGULAR TERMS WITH REGULAR PAYMENTS. THE REGULAR RATE IS 27.99%. THIS APR WILL VARY WITH THE MARKET BASED ON THE PRIME RATE.

This messages is stored in a text field on the `TransferMetadata` model and should be rendered into the order confirmation template after a user places an order using Wells Fargo Retail Services.

1.1.5 Financing Plan

A Financing Plan (or just Plan) defines the terms of the financing agreement (APR, term length, etc) that a customer will use to pay for an order. It is defined by a *Plan Number* which is a 4-digit numeric code between *1001* and *9999*. This number if sent to Wells Fargo when performing an authorization or charge. Financing Plan can be added, edited, and deleted in the Oscar dashboard.

1.1.6 Financing Plan Benefit

Since Financing Plans control what terms a customer gets with they financing, you may wish to have control over which customers have the ability to use which plans. For example, consider the following business rules.

1. Plan 1001 has a 27% APR and should be usable by everyone placing an order on the website.
2. Plan 1002 has a 0% APR, but should only be available for use when an order is over $500.00.

Financing Plan Benefits allow this to happen. A Financing Plan Benefit is a special type of offer / voucher benefit (just like a *$10 off* or *15% off* a normal benefit) whose sole job is to make Financing Plans available to customers. By default, a customer is not considered eligible for any Financing Plans. To model the business rules listed above, we'd do the following in the Oscar dashboard.

1. **Create both financing plans, using the *Financing Plans* view at *Dashboard > Wells Fargo > Financing Plans*.**
    1. Set the first plans plan number to 1001 and input the correct term length and APR.
    2. Set the second plans plan number to 1002 and input the correct term length and APR.
2. **Create two *Financing Plan Groups* using the view at *Dashboard > Wells Fargo > Financing Plan Groups*.**
    1. Give the first group a name like *Default Financing* and select plan 1001.
    2. Give the second group a name like *Special Rate Financing* and select plan 1002.
3. **Create two offer conditions (*Dashboard > Offers > Conditions*) to match the needed conditions.**
    1. The first should be a value condition requiring the basket contain more than $0.01.
    2. The second should be a value condition requiring the basket contain more than $500.00.
4. **Tie everything together by creating two offers (*Dashboard > Offers > Offers*).**
    1. The first offer should use the $0.01 condition and the *Default Financing* benefit.
    2. The first offer should use the $500.00 condition and the *Special Rate Financing* benefit.

Once this is down, Oscar will make plans available just like it applies other offers and benefits to baskets.

1.2 Installation

1.2.1 Caveats

django-oscar-wfrs is built on top of *django-oscar-api* and *django-oscar-api-checkout*. Out of the box, it will not work with the built-in django-oscar (non-ajax) checkout. You can extend

1.2.2 Before Installing

In your project, if you haven't already done so, follow the installation instructions for the following dependent libraries.

1. django-oscar
2. django-oscar-api
3. django-oscar-api-checkout
4. django-oscar-bluelight
5. django-haystack

1.2.3 Installing

Install the `django-oscar-wfrs` package.

```
$ pip install django-oscar-wfrs
```

Add `wellsfargo` to your `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.postgres',
    'wellsfargo',
]
```

Add the template directory to your template settings.

```
from oscar import OSCAR_MAIN_TEMPLATE_DIR
from oscarbluelight import BLUELIGHT_TEMPLATE_DIR
from wellsfargo import WFRS_TEMPLATE_DIR

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            WFRS_TEMPLATE_DIR,
            BLUELIGHT_TEMPLATE_DIR,
            OSCAR_MAIN_TEMPLATE_DIR,
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'oscar.apps.search.context_processors.search_form',
                'oscar.apps.promotions.context_processors.promotions',
                'oscar.apps.checkout.context_processors.checkout',
                'oscar.apps.customer.notifications.context_processors.notifications',
                'oscar.core.context_processors.metadata',
```

(continues on next page)

(continued from previous page)

```
            ],
        },
    },
]
```

Add the Wells Fargo views to the OSCAR_DASHBOARD_NAVIGATION setting in `settings.py`. This will add a new item to the navigation bar in the Oscar dashboard.

```
OSCAR_DASHBOARD_NAVIGATION.append({
    'label': 'Wells Fargo',
    'icon': 'icon-globe',
    'children': [
        # Wells Fargo Retail Services Views
        {
            'label': 'Apply for a Credit Line (Wells Fargo)',
            'url_name': 'wfrs-apply-step1',
        },
        {
            'label': 'Add existing Wells Fargo account',
            'url_name': 'wfrs-add-account',
        },
        {
            'label': 'Financing Plans',
            'url_name': 'wfrs-plan-list',
        },
        {
            'label': 'Financing Plan Groups',
            'url_name': 'wfrs-benefit-list',
        },
        {
            'label': 'Credit Applications',
            'url_name': 'wfrs-application-list',
        },
        {
            'label': 'Transfers',
            'url_name': 'wfrs-transfer-list',
        },
        {
            'label': 'Pre-Qualification Requests',
            'url_name': 'wfrs-prequal-list',
        },
    ]
})
```

Configure `django-oscar-api-checkout` to use `django-oscar-wfrs` as a possible payment option. The example below makes Wells Fargo payment available to everyone, but you may wish to set a different permission class and restrict it to staff users, members of a group, etc.

```
API_ENABLED_PAYMENT_METHODS = [
    {
        'method': 'wellsfargo.methods.WellsFargo',
        'permission': 'oscarapicheckout.permissions.Public',
    },
]
```

Add `wellsfargo.models.FinancingPlanBenefit` to `BLUELIGHT_BENEFIT_CLASSES` so that we can use the offers system to control financing plan availability. See *Concepts* for more information on why.

```
BLUELIGHT_BENEFIT_CLASSES += [
    ('wellsfargo.models.FinancingPlanBenefit', 'Activate Wells Fargo Plan Number Group
↪'),
]
```

Configure to connect to either the UAT or the Production Wells Fargo Retail Services SOAP API.

```
WFRS_TRANSACTION_WSDL = 'https://retailservices-uat.wellsfargo.com/services/
↪SubmitTransactionService?WSDL'
WFRS_INQUIRY_WSDL = 'https://retailservices-uat.wellsfargo.com/services/
↪SubmitInquiryService?WSDL'
WFRS_CREDIT_APP_WSDL = 'https://retailservices-uat.wellsfargo.com/services/
↪SubmitCreditAppService?WSDL'
```

Configure an encryption key to use when encrypting Wells Fargo Account Numbers. By default this uses symmetric encryption by means of Fernet. Alternatively, you may point to a different class implementing the same interface and do encryption by another means, like KMS (in which case you wouldn't need to specify a key argument). If you do use Fernet, keep in mind that…

1. …the key should be a a 32-byte sequence that's been base64 encoded.
2. …the key must be a byte sequence, not a string.
3. …the key should not be stored in source code or in the database. Please use an environment variable or a secret store like Hasicorp Vault.
4. …you must not lose the key. Losing the key will render any encrypted account number's you have saved unusable.

```
import os

# Key should be something like b'U3Nyi57e55H2weKVmEPzrGdv18b0bGt3e542rg1J1N8='
WFRS_SECURITY = {
    'encryptor': 'wellsfargo.security.fernet.FernetEncryption',
    'encryptor_kwargs': {
        'key': os.environ.get('WFRS_ENCRYPTION_KEY', '').encode(),
    },
}
```

Add the `django-oscar-wfrs` views to your projects url configuration.

```
from oscar.app import application as oscar_application
from oscarapi.app import application as oscar_api
from oscarapicheckout.app import application as oscar_api_checkout
from wellsfargo.api.app import application as wfrs_api
from wellsfargo.dashboard.app import application as wfrs_app

urlpatterns = [
    # Include plugins
    url(r'^dashboard/wfrs/', include(wfrs_app.urls)),
    url(r'^api/wfrs/', include(wfrs_api.urls)),
    url(r'^api/', include(oscar_api_checkout.urls)),
    url(r'^api/', include(oscar_api.urls)),

    # Include stock Oscar
    url(r'', include(oscar_application.urls)),
]
```

Add your Wells Fargo Retail Services SOAP API credentials to the database. You can do this directly, or via the Django Admin at `/admin/wellsfargo/apicredentials/`.

```
INSERT INTO wellsfargo_apicredentials
            (username,
             password,
             merchant_num,
             priority)
VALUES      ('WS000000000000000',
             'MY_WELLSFARGO_PASSWORD',
             '000000000000000');
```

1.3 Fraud Protection

To help prevent fraudulent transactions, `django-oscar-wfrs` supports pluggable fraud protection modules to screen transactions before they are sent to Wells Fargo. Currently, two modules are included:

Package Name	Description
wellsfargo.fraud.dummy.DummyFraudProtection	Default fraud protection class. Doesn't actually screen transactions—just approves everything.
wellsfargo.fraud.cybersource.DecisionManagerFraudProtection	Uses CyberSource's Decision Manager via a SOAP API to screen transactions. See Cybersource for more information.

1.3.1 Configuration

To configure fraud protection, use the `WFRS_FRAUD_PROTECTION` setting in Django settings. For example, to configure the Decision Manager module, add the following configuration to your project's settings file.

```
WFRS_FRAUD_PROTECTION = {
    'fraud_protection': 'wellsfargo.fraud.cybersource.DecisionManagerFraudProtection',
    'fraud_protection_kwargs': {
        'wsdl': 'https://ics2wstesta.ic3.com/commerce/1.x/transactionProcessor/
↪CyberSourceTransaction_1.141.wsdl',
        'merchant_id': 'my-merchant-id',
        'transaction_security_key': 'my-security-key',
    }
}
```

Follow Cybersource's documentation on how to obtain your merchant ID and transaction security key.

1.4 Integration

Since `django-oscar-api-checkout` and `django-oscar-wfrs` are designed around the idea of client-side AJAX centric checkouts, theres a bit of custom work to do to integrate `django-oscar-wfrs` into your client side application. Here is a basic overview of what the client-side application needs to do. The URLs below assume you followed the *installation instruction* and installed the WFRS API at `/api/wfrs/`.

1.4.1 Submitting a Credit Application

Make a `POST` request to `/api/wfrs/apply/` specifying the region and the application type. This will return a link to the Application endpoint to continue with.

Parameter	Values
region	`US` (United States) or `CA` (Canada)
app_type	`I` (Individual) or `J` (Joint)

```
POST /api/wfrs/apply/

region=US&app_type=I
```

The API will return the URL of the appropriate form.

```
{
    "url": https://mysite.com/api/wfrs/apply/us-individual/
}
```

Make an `OPTIONS` request to the returned URL to get information about which fields need to make up the application form.

```
OPTIONS /api/wfrs/apply/us-individual/
```

The API will return the fields and validation rules applicable to the selected form.

```
{
    "actions": {
        "POST": {
            "region": {
                "type": "choice",
                "required": false,
                "read_only": false,
                "label": "Region",
                "choices": [
                    {
                        "display_name": "United States",
                        "value": "US"
                    }
                ]
            },
            "app_type": {
                "type": "choice",
                "required": false,
                "read_only": false,
                "label": "App type",
                "choices": [
                    {
                        "display_name": "Individual",
                        "value": "I"
                    }
                ]
            },
            "language": {
                "type": "choice",
                "required": false,
```

(continues on next page)

(continued from previous page)

```
                "read_only": false,
                "label": "Language",
                "choices": [
                    {
                        "display_name": "English",
                        "value": "E"
                    }
                ]
            },
            "purchase_price": {
                "type": "integer",
                "required": false,
                "read_only": false,
                "label": "Requested Credit Amount",
                "min_value": 0,
                "max_value": 99999
            },
            "main_first_name": {
                "type": "string",
                "required": true,
                "read_only": false,
                "label": "First Name",
                "max_length": 15
            },
        }
    }
}
```

Using that response, build and display the application form to the customer to fill out. However this is down will vary greatly depending on the architecture of your client-side code.

After the user has completely filled out the form, `POST` the data back to the same URL as JSON.

```
POST /api/wfrs/apply/us-individual/

{
    "region": "US",
    "app_type": "I",
    "language": "E",
    "purchase_price": "1000",
    "main_first_name": "Rusty"
}
```

If any of the data is invalid or if the credit application is denied by Wells Fargo, a response like this (with a description of the error) will be returned.

```
{
    "main_last_name": [
        "This field may not be blank."
    ]
}
```

If the application was successfully approved by Wells Fargo, a response like this will be returned.

```
{
    "account_number": "9999999999999999",
    "credit_limit": "7500.00",
```

(continues on next page)

(continued from previous page)

```
    "balance": "0.00",
    "open_to_buy": "7500.00",
}
```

You should then display the account number to the user and tell them to write it down, print it, etc. If lost, they will not be able to easily recover it, as the application has not recorded or saved it anywhere.

1.4.2 Placing an Order

Once a customer has indicated that they would like to pay using WFRS, make a `GET` request to `/api/wfrs/plans/` to list which plans the user is eligible to use. This is based on the offers data configured in the *Oscar dashboard*.

```
GET /api/wfrs/plans/
```

The API will return the plans available to the user.

```
[
    {
        "id": 1,
        "plan_number": 1001,
        "description": "Regular Terms Apply",
        "apr": "28.99",
        "term_months": 0,
        "allow_credit_application": false
    },
    {
        "id": 2,
        "plan_number": 1002,
        "description": "Special rate of 0% APR with for 48 months",
        "apr": "0.00",
        "term_months": 48,
        "allow_credit_application": true
    }
]
```

Use this data to construct a form to allow the customer to enter their account number and to pick which financing plan they would like to use.

After they've entered both their account number and picked their financing plan, they can place their order. To place the order, submit the checkout data to the `django-oscar-api-checkout` API endpoint with WFRS data included in the payment block.

Parameter	Values
account_number	The customer's full 16 digit account number devoid of spaces or other characters.
financing_plan	The ID (primary key) of the selected financing plan. *Note: not the plan number.*

```
POST /api/checkout/

{
    "payment": {
        "wells-fargo": {
            "enabled": true,
            "account_number": "9999999999999999",
            "financing_plan": 2
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "guest_email": "joe@example.com",
    "basket": "/api/baskets/1/",
    "shipping_address": {
        "first_name": "Joe",
        "last_name": "Schmoe",
        "line1": "234 5th Ave",
        "line4": "Manhattan",
        "postcode": "10001",
        "state": "NY",
        "country": "/api/countries/US/",
        "phone_number": "+1 (717) 467-1111"
    },
    "billing_address": {
        "first_name": "Joe",
        "last_name": "Schmoe",
        "line1": "234 5th Ave",
        "line4": "Manhattan",
        "postcode": "10001",
        "state": "NY",
        "country": "/api/countries/US/",
        "phone_number": "+1 (717) 467-1111"
    }
}
```

Upon submission, `django-oscar-wfrs` attempts to authorize payment on the given account. Regardless of whether or not payment is successful, the order is object is created and is returned in the response.

```
{
    "number": "1234",
    ...
}
```

You application must then check the status of the payment source to see if it was completed successfully.

```
GET /api/checkout/payment-states/
```

If the authorization was successful, the response will look like this.

```
{
    "order_status": "Authorized",
    "payment_method_states": {
        "wells-fargo": {
            "status": "Complete",
            "amount": "200.00",
            "required_action": null
        }
    }
}
```

If the authorization was unsuccessful, the response will look like this.

```
{
    "order_status": "Payment Declined",
    "payment_method_states": {
```

(continues on next page)

(continued from previous page)

```
        "wells-fargo": {
            "status": "Declined",
            "amount": "200.00",
            "required_action": null
        }
    }
}
```

If you receive a successfully authorized response, you can now forward the customer to the order thank you page. Otherwise, you should inform them that the payment authorization was unsuccessful and that they should re-enter their account number and try again. Once they do this, you can retry the `POST` to `/api/checkout/`. If the authorization continues to fail, this most likely means that either they do not have a valid account number or that there isn't enough credit left on their account to cover the purchase.

1.5 Internals

1.5.1 Packages

`django-oscar-wfrs` is split into packages based on area of concerns.

Package Name	Description
wellsfargo	Top level package container. Contains files that django and haystack expect to be in specific spots (models.py, search_indexes.py, etc).
wellsfargo.api	Django Rest Framework based API exposing actions like credit applications and financing plan discover.
wellsfargo.connector	Wrapper for talking to the WFRS SOAP API.
wellsfargo.core	Core components like data structures and exceptions.
wellsfargo.dashboard	Oscar Dashboard application for managing financing plans, searching credit applications, etc.
wellsfargo.fraud	Pluggable transaction fraud protection connectors.
wellsfargo.security	Encryption utilities for protecting account numbers.
wellsfargo.templatetags	Django Template tags.
wellsfargo.tests	Test suite.

1.6 Changelog

1.6.1 0.19.0 *work in progress*

- Drop support for Oscar 1.5
- Use PostgreSQL full-text search (instead of Haystack) for Pre-Qualification data

1.6.2 0.18.0

- Internationalization
- Add PreQual request data into PreQualificationResponseSerializer serializer
- Add view to allow resuming a PreQual offer from a different session

1.6.3 0.17.0

- Make payment methods create separate `payment.Source` objects per Reference number (!24).
- Change behavior of denied and pended credit applications. Application records are now always saved to the database (!26).
- Made Fraud screen system fail-open (rather than closed, denying all orders) upon returning an error.

1.6.4 0.16.0

- **Improve Pre-Qualification Dashboard.**
  - Adds new columns
  - Improves search using Haystack
  - Adds export ability
- Add financing advertising thresholds to the API

1.6.5 0.15.1

- Fix widget rendering issue in Django 2.1

1.6.6 0.15.0

- Extend PreQual views to work with new Wells Fargo Pre-Approval SDK.
- Record transaction records for denied transaction attempts.
- Add support for Django 2.1
- Add support for Python 3.7

1.6.7 0.14.0

- Upgrade to django-oscar-bluelight 0.10.0.
- Make Wells Fargo offers use HiddenPostOrderAction results.

1.6.8 0.13.1

- Adds support for Django 2.0 and Oscar 1.6.

1.6.9 0.13.0

- Adds support for `django-oscar-api-checkout>=0.4.0`

1.6.10 0.12.1

- Update compatible django-oscar-api-checkout version

1.6.11 0.12.0

- Add new API endpoint for estimating loan payments based on advertised plan thresholds.

1.6.12 0.11.0

- Add support for Wells Fargo's Pre-Qualification (soft-credit check) API.

1.6.13 0.10.1

- Fix corrupted package build in version `0.10.0`.

1.6.14 0.10.0

- **Add support for django-localflavor 2.0 by switching to using django-phonenumber-field for phone number fields.**
    - This introduces a breaking change in the application APIs. Phone number fields were previously expected to be submitted in the format: `5555555555`. They must now be submitted in a format accepted by python-phonenumbers, such as `+1 (555) 555-5555` or `+1 555.555.5555`.
- Remove previously squashed migrations.
- Remove dependency on django-oscar-accounts and django-oscar-accounts2.
- Fix Django 2.0 deprecation warnings.

1.6.15 0.9.1

- Patch package requirements to require django-localflavor less than 2.0.

1.6.16 0.9.0

- Add automatic retries to transactions when they encounter a network issue.

1.6.17 0.8.0

- Add ability to gate transaction using pluggable fraud screen modules. By default fraud screening is disabled.

1.6.18 0.7.2

- Add support for Django 1.11 and Oscar 1.5
- **Add new field to the FinancingPlan model to contain a price threshold value.**
  - While the offers system is still used to determine what plans a basket is eligible for, sometimes plan data is needed before a product is in the basket. For example, you may wish to advertise a monthly payment price for a product outside of the basket context. Previously the `is_default_plan` flag was used for this purpose. Now, each plan can have a price threshold set in the `product_price_threshold`. Then, those threshold values can be used to determine which plan to display for each product. For example, if you configure plan 0001 with a threshold of $100.00 and plan 0002 with a threshold of $200.00, a product costing $150.00 would display a monthly price calculated based on plan 0001 while a product costing $500.00 would display a monthly price calculated based on plan 0002. The `is_default_plan` flag still exists and can be used as a fallback to products not meeting any of the configured thresholds.
  - Add template override in the sandbox store to demonstrate this behavior.

1.6.19 0.7.1

- Add new field to the FinancingPlan model to contain a superscript number, corresponding to fine print displayed elsewhere on the page.

1.6.20 0.7.0

- Fix 404ing JS in Oscar Dashboard
- **Add several new columns to the Credit Application dashboard:**
  - Merchant Name used for application
  - Application Source
  - Requested Credit Amount
  - Resulting Credit Limit
  - Order total of first related order
  - Merchant name used for order
- Fixes exception thrown when trying to decrypt invalid data using KMS backend
- Add button to export a CSV of credit applications from the dashboard
- **Make Wells Fargo Benefits use offer conditions to consume basket lines**
  - Use oscar-bluelight's offer groups feature to allow stacking other discounts with financing benefits. The recommended set-up is to place all Wells Fargo related offers into an offer group of their own, configured with a lower priority than any other group.

1.6.21 0.6.7

- Add new multi-encryptor class that combines multiple other encryptors together. This allows key rotation and graceful migration between different encryption methods.

1.6.22 0.6.6

- Handle pending application responses separately from denied responses. They now throw different API exceptions with different error messages and error codes.
- Add some basic dashboard view tests.

1.6.23 0.6.5

- Add foreign key from TransferMetadata to APICredentials used to make the transfer.

1.6.24 0.6.4

- Fix bug which prevented adding new plan groups via the dashboard.
- Adds unit tests for financing plan and financing plan group dashboard forms.

1.6.25 0.6.3

- Save last 4 digits of resulting account number to credit application models.
- Add `TransferMetadata.purge_encrypted_account_number` method.
- Handle ValidationError when submitting a transaction to prevent 500 errors in checkout.
- Fix 500 error in Credit App API when SOAP API returned a validation issue.
- Fix install documentation regarding API credentials.

1.6.26 0.6.2

- Fix bug when migrating account numbers to new encrypted fields.

1.6.27 0.6.1

- Moved Fernet encryption class from `wellsfargo.security.FernetEncryption` to `wellsfargo.security.fernet.FernetEncryption`.
- Added alternative AWS KMS encryption class as `wellsfargo.security.kms.KMSEncryption`.

1.6.28 0.6.0

- **Major Release. Breaking Changes.**
- Drop dependency on django-oscar-accounts.
- Stop tracking accounts in database.
- Account numbers are now encrypted at rest.

1.6.29 0.5.0

- Add support for Django 1.10, Python 3.6.
- Drop support for Django 1.8, Python 3.4.

1.6.30 0.4.3

- During reconciliation with WFRS, adjust credit limit before doing compensating transaction.

1.6.31 0.4.2

- Make application date times display in localized timezone in the dashboard search-results table.

1.6.32 0.4.1

- Upgrade dependencies.

1.6.33 0.4.0

- Add improved credit application search functionality to dashboard.
- Fix bug where AccountInquiryResult.reconcile() would sometimes attempt to make a debit with a negative amount.

1.6.34 0.3.1

- Add boolean for controlling whether or not to display a credit application form to the client.

1.6.35 0.3.0

- Move API credentials into database, optionally triggered by user group.

1.6.36 0.2.6

- Add a relation between wellsfargo.AccountMetadata and order.BillingAddress.

1.6.37 0.2.5

- Prevent creating invalid WFRS Plan Group Benefits in the standard bluelight benefit dashboard.

1.6.38 0.1.0

- Initial release.