Django Hordak Documentation

Release 1.0

Adam Charnock

Jul 23, 2019

Contents:

Django Hordak is the core functionality of a double entry accounting system. It provides thoroughly tested core models with relational integrity constrains to ensure consistency.

Hordak also includes a basic accounting interface. This should allow you to get up-and-running quickly. However, the expectation is that you will either heavily build on this example or use one of the interfaces detailed below.

Interfaces which build on Hordak include:

- battlecat – General purpose accounting interface (work in progress)
- swiftwind – Accounting for communal households (work in progress)

CHAPTER 1

Requirements

Hordak is tested against:

- Django >= 1.10, <= 2.0
- Python >= 3.4
- Postgres >= 9.5

Postgres is required, MySQL is unsupported. This is due to the database constraints we apply to ensure data integrity. MySQL could be certainly supported in future, volunteers welcome.

1.1 Installation

Installation using pip:

```
pip install django-hordak
```

Add to installed apps:

```
INSTALLED_APPS = [
    ...
    'mptt',
    'hordak',
]
```

Note: Hordak uses django-mptt to provide the account tree structure. It must therefore be listed in `INSTALLED_APPS` as shown above.

Before continuing, ensure the `HORDAK_DECIMAL_PLACES` and `HORDAK_MAX_DIGITS` *settings* are set as desired. Changing these values in future will require you to create your own custom database migration in order to update your schema (perhaps by using Django's `MIGRATION_MODULES` setting). It is therefore best to be sure of these values now.

Once ready, run the migrations:

```
./manage.py migrate
```

1.1.1 Using the interface

Hordak comes with a basic interface. The intention is that you will either build on it, or use a *another interface*. To get started with the example interface you can add the following to your `urls.py`:

```
urlpatterns = [
    ...
    url(r'^', include('hordak.urls', namespace='hordak'))
]
```

You should then be able to create a user and start the development server (assuming you ran the migrations as detailed above):

```
# Create a user to login as
./manage.py createsuperuser
# Start the development server
./manage.py runserver
```

And now navigate to http://127.0.0.1:8000/.

1.1.2 Using the models

Hordak's primary purpose is to provide a set of robust models with which you can model the core of a double entry accounting system. Having completed the above setup you should be able to import these models and put them to use.

```
from hordak.models import Account, Transaction, ...
```

You can find further details in the *API documentation*. You may also find the *accounting for developers* section useful.

1.2 Settings

You can set the following your project's `settings.py` file:

1.2.1 DEFAULT_CURRENCY

Default: `"EUR"`

The default currency to use when creating new accounts

1.2.2 CURRENCIES

Default: `[]`

Any currencies (additional to `DEFAULT_CURRENCY`) for which you wish to create accounts. For example, you may have `"EUR"` for your `DEFAULT_CURRENCY`, and `["USD", "GBP"]` for your additional `CURRENCIES`.

1.2.3 HORDAK_DECIMAL_PLACES

Default: `2`

Number of decimal places available within monetary values.

1.2.4 HORDAK_MAX_DIGITS

Default: `13`

Maximum number of digits allowed in monetary values. Decimal places both right and left of decimal point are included in this count. Therefore a maximum value of 9,999,999.999 would require `HORDAK_MAX_DIGITS=10` and `HORDAK_DECIMAL_PLACES=3`.

1.3 Customising Templates

The easiest way to modify Hordak's default interface is to customise the default templates.

Note: This provides a basic level of customisation. For more control you will need to extend the *views*, or create entirely new views of your own which build on Hordak's *models*.

Hordak's templates can be found in hordak/templates/hordak. You can override these templates by creating similarly named files in your app's own `templates` directory.

For example, if you wish to override `hordak/account_list.html`, you should create the file `hordak/account_list.html` within your own app's template directory. Your template will then be used by Django rather than the original.

Important: By default Django searches for templates in each app's `templates` directory. It does this in the order listed in `INSTALLED_APPS`. Therefore, **your app must appear before 'hordak' in 'INSTALLED_APPS'**.

1.4 Double Entry Accounting for Developers

Hordak is inherently aimed at software developers as it provides core functionality only. Friendly interfaces can certainly be built on top of it, but if you are here there is a good change you are a developer.

If you are learning about accounting as developer you may feel – as I did – that most of the material available doesn't quite relate to the developer/STEM mindset. I therefore provide some resources here that may be of use.

1.4.1 Accounting in six bullet points (& three footnotes)

I found the core explanation of double entry accounting to be confusing. After some time I distilled it down to the following:

1. Each account has a 'type' (asset, liability, income, expense, equity).

2. **Debits decrease** the value of an account. Always.[1]
3. **Credits increase** the value of an account. Always.[1]
4. The sign of any **asset** or **expense** account balance is **always flipped** upon display (i.e. multiply by -1)[2][3].
5. A transaction is comprised of 1 or more credits **and** 1 or more debits (i.e. money must come from somewhere and then go somewhere).
6. The value of a transaction's debits and credits must be equal (money into transaction = money out of transaction).

1.4.2 In a little more detail

I found Peter Selinger's tutorial to be very enlightening and is less terse than the functional description above. The first section is short and covers single entry accounting, and then shows how one can expand that to create double entry accounting. I found this background useful.

1.4.3 Examples

You live in a shared house. Everyone pays their share into a communal bank account every month.

Example 1: Saving money to pay a bill (no sign flipping)

You pay the electricity bill every three months. Therefore every month you take £100 from everyone's contributions and put it into Electricity Payable account (a liability account) in the knowledge that you will pay the bill from this account when it eventually arrives:

These accounts are income & liability accounts, so neither balance needs to be flipped (flipping only applies to asset & expense accounts). Therefore:

- Balances before:
  - *Housemate Contribution* (income): £500
  - *Electricity Payable* (liability): £0
- **Transaction**:
  - £100 from *Housemate Contribution* to *Electricity Payable*
- Balances after:
  - *Housemate Contribution* (income): £400
  - *Electricity Payable* (liability): £100

This should also make intuitive sense. Some of the housemate contributions will be used to pay the electricity bill, therefore the former decreases and the latter increases.

[1] This is absolutely not what accountancy teaches. You'll quickly see that there is a lot of wrangling over what account types get increased/decreased with a debit/credit. I've simplified this on the backend as I strongly feel this is a presentational issue, and not a business logic issue.

[2] Peter Selinger's tutorial will give an indication of why this is (hint: see the signs in the accounting equation). However, a simple explanation is, *'accountants don't like negative numbers.'* A more nuanced interpretation is that a positive number indicates not a positive amount of money, but a positive amount of whatever the account is. So an expense of $1,000 is a positive amount of expense, even though it probably means your $1,000 less well off.

[3] An upshot of this sign flipping in 4 is that points 2 & 3 appear not be be obeyed from an external perspective. If you debit (decrease) an account, then flip its sign, it will look like you have actually increased the account balance. This is because we are treating the sign of asset & expense accounts as a presentational issue, rather than something to be dealt with in the core business logic.

Example 2: Saving money to pay a bill (with sign flipping)

At the start of every month each housemate pays into the communal bank account. We should therefore represent this somehow in our double entry system (something we ignored in example 1).

We have an account called *Bank* which is an asset account (because this is money we actually have). We also have a *Housemate Contribution* account which is an income account.

Therefore, **to represent the fact that we have been paid money, we must create a transaction**. However, money cannot be injected from outside our double entry system, so how do we deal with this?

Let's show how we represent a single housemate's payment:

- Balances before:
  - *Bank* (asset): £0
  - *Housemate Contribution* (income): £0
- **Transaction:**
  - £500 from *Bank* to *Housemate Contribution*
- Balances after:
  - *Bank* (asset): -£500 * -1 = **£500**
  - *Housemate Contribution* (income): £500

Because the bank account is an asset account, we flip the sign of its balance. **The result is that both accounts increase in value.**

1.5 Hordak Database Triggers

Hordak uses triggers at the database level instead of Django signals. This ensures that if data does not pass through the Django ORM that integrity is still maintained via Hordak's accounting business rules.

Note: These triggers are automatically added to the database engine through custom Django migration files. When the migrate command is run these triggers will be created.

---

6 Triggers and constraints are added to interact with Hordak models:

- *check_leg*
- *zero_amount_check*
- *check_leg_and_account_currency_match*
- *bank_accounts_are_asset_accounts*
- *update_full_account_codes*
- *check_account_type*

1.5.1 The `check_leg` trigger

A trigger is added that executes a SQL procedure when each row in the `hordak.models.Leg` database table is **inserted**, **updated**, or **deleted**.

This constraint is set with execution timing of `DEFERRABLE INITIALLY DEFERRED`, which means it is executed when a transaction is finished.

Note: If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction.[1]

This trigger ensures that the total amount for the legs of a transaction is equal to 0. Or else it raises a database level exception.

Procedure Code

```
DECLARE
    tx_id INT;
    non_zero RECORD;
BEGIN
    IF (TG_OP = 'DELETE') THEN
        tx_id := OLD.transaction_id;
    ELSE
        tx_id := NEW.transaction_id;
    END IF;
    SELECT ABS(SUM(amount)) AS total, amount_currency AS currency
        INTO non_zero
        FROM hordak_leg
        WHERE transaction_id = tx_id
        GROUP BY amount_currency
        HAVING ABS(SUM(amount)) > 0
        LIMIT 1;
    IF FOUND THEN
        RAISE EXCEPTION 'Sum of transaction amounts in each currency must be 0.␣
↪Currency % has non-zero total %',
            non_zero.currency, non_zero.total;
    END IF;
    RETURN NEW;
END;
```

1.5.2 The `zero_amount_check` constraint

A constraint is added that checks the value of the `amount` field of *`hordak.models.Leg`*.

This constraint ensures that amount value for a single leg transaction does not equal 0. Or else it raises a database level exception.

Procedure Code

```
ALTER TABLE hordak_leg ADD CONSTRAINT zero_amount_check CHECK (amount != 0)
```

[1] Deferrable trigger parameters from CREATE TRIGGER.

1.5.3 The `check_leg_and_account_currency_match` constraint

A trigger is added that executes a SQL procedure when each row in the *hordak.models.Leg* database table is **inserted**, **updated**, or **deleted**. This constraint is set with execution timing of `DEFERRABLE INITIALLY DEFERRED`

This procedure ensures that destination account for a leg transaction has the same currency as the origin account.

Procedure Code

```
DECLARE
BEGIN
    IF (TG_OP = 'DELETE') THEN
        RETURN OLD;
    END IF;
    PERFORM * FROM hordak_account WHERE id = NEW.account_id AND NEW.amount_currency =␣
↪ANY(currencies);
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Destination account does not support currency %', NEW.amount_
↪currency;
    END IF;
    RETURN NEW;
END;
```

1.5.4 The `bank_accounts_are_asset_accounts` constraint

A constraint is added that interacts with the *hordak.models.Account* database table.

This constraint ensures that Account objects that have the is_bank_account flag set must be an asset account type.

Procedure Code

```
ADD CONSTRAINT bank_accounts_are_asset_accounts
CHECK (is_bank_account = FALSE OR _type = 'AS')
```

1.5.5 The `update_full_account_codes` trigger

A trigger is added that executes a SQL procedure when each row in the *hordak.models.Account* database table is **inserted**, **updated**, or **deleted** and where it is also a root Account. This trigger is set with default execution timing of `DEFERRABLE INITIALLY IMMEDIATE`

This procedure performs multiple activities:

- It sets any empty string *hordak.models.Account* `account.code` to `NULL` database value.
- It sets the `account.full_code` of children accounts to a combination of its parents `account.code`.
- If a parent `account.code` is `NULL` it sets the children's subsequent `account.full_code` to `NULL` also.

Procedure Code

```
BEGIN
    -- Set empty string codes to be NULL
    UPDATE hordak_account SET code = NULL where code = '';

    -- Set full code to the combination of the parent account's codes
    UPDATE
        hordak_account AS a
    SET
        full_code = (
            SELECT string_agg(code, '' order by lft)
            FROM hordak_account AS a2
            WHERE a2.lft <= a.lft AND a2.rght >= a.rght AND a.tree_id = a2.tree_id
        );

    -- Set full codes to NULL where a parent account includes a NULL code
    UPDATE
        hordak_account AS a
    SET
        full_code = NULL
    WHERE
        (
            SELECT COUNT(*)
            FROM hordak_account AS a2
            WHERE a2.lft <= a.lft AND a2.rght >= a.rght AND a.tree_id = a2.tree_id␣
↪AND a2.code IS NULL
        ) > 0;
    RETURN NULL;
END;
```

1.5.6 The `check_account_type` trigger

A trigger is added that executes a SQL procedure when each row in the *hordak.models.Account* database table is **inserted** or **updated** and where it is also a root Account. This trigger is set with default execution timing of `DEFERRABLE INITIALLY IMMEDIATE`

This procedure sets children accounts to the same type as the parent account.

Procedure Code

```
BEGIN
    IF NEW.parent_id::BOOL THEN
        NEW.type = (SELECT type FROM hordak_account WHERE id = NEW.parent_id);
    END IF;
    RETURN NEW;
END;
```

1.6 API Documentation

1.6.1 Models

Contents

Account

class hordak.models.**Account**(**args*, ***kwargs*)

Represents an account

An account may have a parent, and may have zero or more children. Only root accounts can have a type, all child accounts are assumed to have the same type as their parent.

An account's balance is calculated as the sum of all of the transaction Leg's referencing the account.

uuid

UUID for account. Use to prevent leaking of IDs (if desired).

Type SmallUUID

name

Name of the account. Required.

Type str

parent

Parent account, nonen if root account

Type Account|None

code

Account code. Must combine with account codes of parent accounts to get fully qualified account code.

Type str

type

Type of account as defined by *Account.TYPES*. Can only be set on root accounts. Child accounts are assumed to have the same time as their parent.

Type str

TYPES

Available account types. Uses `Choices` from `django-model-utils`. Types can be accessed in the form `Account.TYPES.asset`, `Account.TYPES.expense`, etc.

Type Choices

is_bank_account
Is this a bank account. This implies we can import bank statements into it and that it only supports a single currency.

Type bool

save(**args*, ***kwargs*)
If this is a new node, sets tree fields up before it is inserted into the database, making room in the tree structure as neccessary, defaulting to making the new node the last child of its parent.

It the node's left and right edge indicators already been set, we take this as indication that the node has already been set up for insertion, so its tree fields are left untouched.

If this is an existing node and its parent has been changed, performs reparenting in the tree structure, defaulting to making the node the last child of its new parent.

In either case, if the node's class has its `order_insertion_by` tree option set, the node will be inserted or moved to the appropriate position to maintain ordering by the specified field.

classmethod validate_accounting_equation()
Check that all accounts sum to 0

sign
Returns 1 if a credit should increase the value of the account, or -1 if a credit should decrease the value of the account.

This is based on the account type as is standard accounting practice. The signs can be derrived from the following expanded form of the accounting equation:

Assets = Liabilities + Equity + (Income - Expenses)

Which can be rearranged as:

0 = Liabilities + Equity + Income - Expenses - Assets

Further details here: https://en.wikipedia.org/wiki/Debits_and_credits

balance(*as_of=None*, *raw=False*, *leg_query=None*, ***kwargs*)
Get the balance for this account, including child accounts

Parameters

- **as_of** (*Date*) – Only include transactions on or before this date
- **raw** (*bool*) – If true the returned balance should not have its sign adjusted for display purposes.
- **kwargs** (*dict*) – Will be used to filter the transaction legs

Returns Balance

See also:

simple_balance()

simple_balance(*as_of=None*, *raw=False*, *leg_query=None*, ***kwargs*)
Get the balance for this account, ignoring all child accounts

Parameters

- **as_of** (*Date*) – Only include transactions on or before this date
- **raw** (*bool*) – If true the returned balance should not have its sign adjusted for display purposes.

- **leg_query** (*models.Q*) – Django Q-expression, will be used to filter the transaction legs. allows for more complex filtering than that provided by **kwargs.
- **kwargs** (*dict*) – Will be used to filter the transaction legs

Returns Balance

transfer_to(*to_account*, *amount*, ***transaction_kwargs*)

Create a transaction which transfers amount to to_account

This is a shortcut utility method which simplifies the process of transferring between accounts.

This method attempts to perform the transaction in an intuitive manner. For example:

- Transferring income -> income will result in the former decreasing and the latter increasing
- Transferring asset (i.e. bank) -> income will result in the balance of both increasing
- Transferring asset -> asset will result in the former decreasing and the latter increasing

---

Note: Transfers in any direction between `{asset | expense} <-> {income | liability | equity}` will always result in both balances increasing. This may change in future if it is found to be unhelpful.

Transfers to trading accounts will always behave as normal.

Parameters

- **to_account** (Account) – The destination account.
- **amount** (*Money*) – The amount to be transferred.
- **transaction_kwargs** – Passed through to transaction creation. Useful for setting the transaction *description* field.

exception DoesNotExist

exception MultipleObjectsReturned

Transaction

class hordak.models.**Transaction**(**args*, ***kwargs*)

Represents a transaction

A transaction is a movement of funds between two accounts. Each transaction will have two or more legs, each leg specifies an account and an amount.

See also:

Account.transfer_to() is a useful shortcut to avoid having to create transactions manually.

Examples

You can manually create a transaction as follows:

```
from django.db import transaction as db_transaction
from hordak.models import Transaction, Leg

with db_transaction.atomic():
```

(continues on next page)

(continued from previous page)

```
    transaction = Transaction.objects.create()
    Leg.objects.create(transaction=transaction, account=my_account1,␣
↪amount=Money(100, 'EUR'))
    Leg.objects.create(transaction=transaction, account=my_account2,␣
↪amount=Money(-100, 'EUR'))
```

uuid
UUID for transaction. Use to prevent leaking of IDs (if desired).

Type SmallUUID

timestamp
The datetime when the object was created.

Type datetime

date
The date when the transaction actually occurred, as this may be different to *timestamp*.

Type date

description
Optional user-provided description

Type str

exception DoesNotExist

exception MultipleObjectsReturned

Leg

class hordak.models.**Leg**(**args*, ***kwargs*)
The leg of a transaction

Represents a single amount either into or out of a transaction. All legs for a transaction must sum to zero, all legs must be of the same currency.

uuid
UUID for transaction leg. Use to prevent leaking of IDs (if desired).

Type SmallUUID

transaction
Transaction to which the Leg belongs.

Type *Transaction*

account
Account the leg is transferring to/from.

Type *Account*

amount
The amount being transferred

Type Money

description
Optional user-provided description

Type str

type
hordak.models.DEBIT or hordak.models.CREDIT.

Type str

save(**args*, ***kwargs*)
Save the current instance. Override this in a subclass if you want to control the saving process.

The 'force_insert' and 'force_update' parameters can be used to insist that the "save" must be an SQL insert or update (or equivalent for non-SQL backends), respectively. Normally, they should not be set.

account_balance_after()
Get the balance of the account associated with this leg following the transaction

account_balance_before()
Get the balance of the account associated with this leg before the transaction

exception DoesNotExist

exception MultipleObjectsReturned

StatementImport

class hordak.models.**StatementImport**(**args*, ***kwargs*)
Records an import of a bank statement

uuid
UUID for statement import. Use to prevent leaking of IDs (if desired).

Type SmallUUID

timestamp
The datetime when the object was created.

Type datetime

bank_account
The account the import is for (should normally point to an asset account which represents your bank account)

Type *Account*

exception DoesNotExist

exception MultipleObjectsReturned

StatementLine

class hordak.models.**StatementLine**(**args*, ***kwargs*)
Records an single imported bank statement line

A StatementLine is purely a utility to aid in the creation of transactions (in the process known as reconciliation). StatementLines have no impact on account balances.

However, the *StatementLine.create_transaction()* method can be used to create a transaction based on the information in the StatementLine.

uuid
UUID for statement line. Use to prevent leaking of IDs (if desired).

Type SmallUUID

timestamp
The datetime when the object was created.

Type datetime

date
The date given by the statement line

Type date

statement_import
The import to which the line belongs

Type *StatementImport*

amount
The amount for the statement line, positive or nagative.

Type Decimal

description
Any description/memo information provided

Type str

transaction
Optionally, the transaction created for this statement line. This normally occurs during reconciliation. See also *`StatementLine.create_transaction()`*.

Type *Transaction*

is_reconciled
Has this statement line been reconciled?

Determined as `True` if *`transaction`* has been set.

Returns `True` if reconciled, `False` if not.

Return type bool

create_transaction(*to_account*)
Create a transaction for this statement amount and account, into to_account

This will also set this StatementLine's `transaction` attribute to the newly created transaction.

Parameters **to_account** (`Account`) – The account the transaction is into / out of.

Returns The newly created (and committed) transaction.

Return type *Transaction*

exception DoesNotExist

exception MultipleObjectsReturned

1.6.2 Views

Contents

Hordak provides a number of off-the-shelf views to aid in development. You may need to implement your own version of (or extend) these views in order to provide customised functionality.

Extending views

To extend a view you will need to ensure Django loads it by updating your `urls.py` file. To do this, alter you current urls.py:

```
# Replace this
urlpatterns = [
    ...
    url(r'^', include('hordak.urls', namespace='hordak'))
]
```

And changes it as follows, copying in the patterns from hordak's root `urls.py`:

```
# With this
from hordak import views as hordak_views

hordak_urls = [
    ... patterns from Hordak's root urls.py ...
]

urlpatterns = [
    url(r'^admin/', admin.site.urls),

    url(r'^', include(hordak_urls, namespace='hordak', app_name='hordak')),
    ...
]
```

Accounts

AccountListView

class hordak.views.**AccountListView**(***kwargs*)
View for listing accounts

Examples

```
urlpatterns = [
    ...
    url(r'^accounts/$', AccountListView.as_view(), name='accounts_list'),
]
```

model
alias of `hordak.models.core.Account`

template_name = 'hordak/accounts/account_list.html'

context_object_name = 'accounts'

AccountCreateView

class hordak.views.**AccountCreateView**(***kwargs*)
View for creating accounts

Examples

```
urlpatterns = [
    ...
    url(r'^accounts/create/$', AccountCreateView.as_view(success_url=reverse_lazy(
↪'accounts_list')), name='accounts_create'),
]
```

form_class
alias of `hordak.forms.accounts.AccountForm`

template_name = 'hordak/accounts/account_create.html'

success_url = '/'

AccountUpdateView

class hordak.views.**AccountUpdateView**(***kwargs*)
View for updating accounts

Note that `hordak.forms.AccountForm` prevents updating of the `currency` and `type` fields. Also note that this view expects to receive the Account's `uuid` field in the URL (see example below).

Examples

```
urlpatterns = [
    ...
    url(r'^accounts/update/(?P<uuid>.+)/$', AccountUpdateView.as_view(success_
↪url=reverse_lazy('accounts_list')), name='accounts_update'),
]
```

model
alias of `hordak.models.core.Account`

form_class
alias of `hordak.forms.accounts.AccountForm`

template_name = 'hordak/accounts/account_update.html'

slug_field = 'uuid'

slug_url_kwarg = 'uuid'

context_object_name = 'account'

success_url = '/'

AccountTransactionView

class hordak.views.**AccountTransactionsView**(***kwargs*)

template_name = 'hordak/accounts/account_transactions.html'

model
alias of `hordak.models.core.Leg`

slug_field = 'uuid'

slug_url_kwarg = 'uuid'

get(*request*, **args*, ***kwargs*)

get_object(*queryset=None*)
Return the object the view is displaying.

Require *self.queryset* and a *pk* or *slug* argument in the URLconf. Subclasses can override this to return any object.

get_context_object_name(*obj*)
Get the name to use for the object.

get_queryset()
Return the *QuerySet* that will be used to look up the object.

This method is called by the default implementation of get_object() and may not be called if get_object() is overridden.

Transactions

TransactionCreateView

class hordak.views.**TransactionCreateView**(***kwargs*)
View for creation of simple transactions.

This functionality is provided by *hordak.models.Account.transfer_to()*, see the method's documentation for additional details.

Examples

```
urlpatterns = [
    ...
    url(r'^transactions/create/$', TransactionCreateView.as_view(), name=
↪'transactions_create'),
]
```

form_class
alias of hordak.forms.transactions.SimpleTransactionForm

success_url = '/'

template_name = 'hordak/transactions/transaction_create.html'

TransactionsReconcileView

class hordak.views.**TransactionsReconcileView**(***kwargs*)
Handle rendering and processing in the reconciliation view

Note that this only extends ListView, and we implement the form processing functionality manually.

Examples

```
urlpatterns = [
    ...
    url(r'^transactions/reconcile/$', TransactionsReconcileView.as_view(), name=
↪'transactions_reconcile'),
]
```

template_name = 'hordak/transactions/reconcile.html'

model
alias of hordak.models.core.StatementLine

paginate_by = 50

context_object_name = 'statement_lines'

ordering = ['-date', '-pk']

success_url = '/'

1.6.3 Forms

Contents

As with views, Hordak provides a number of off-the-shelf forms. You may need to implement your own version of (or extend) these forms in order to provide customised functionality.

SimpleTransactionForm

class hordak.forms.**SimpleTransactionForm**(**args*, ***kwargs*)

A simplified form for transferring an an amount from one account to another

This only allows the creation of transactions with two legs. This also uses `Account.transfer_to()`.

See also:

- *hordak.models.Account.transfer_to()*.

TransactionForm

class hordak.forms.**TransactionForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*, *use_required_attribute=None*, *renderer=None*)

A form for managing transactions with an arbitrary number of legs.

You will almost certainly need to combine this with `LegFormSet` in order to create & edit transactions.

Note: For simple transactions (with a single credit and single debit) you a probably better of using the *SimpleTransactionForm*. This significantly simplifies both the interface and implementation.

---

description

Optional description/notes for this transaction

Type forms.CharField

See also:

This is a *ModelForm* for the *Transaction model*.

LegForm

class hordak.forms.**LegForm**(**args*, ***kwargs*)

A form for representing a single transaction leg

account

Choose an account the leg will interact with

Type TreeNodeChoiceField

description

Optional description/notes for this leg

Type forms.CharField

amount

The amount for this leg. Positive values indicate money coming into the transaction, negative values indicate money leaving the transaction.

Type MoneyField

See also:

This is a *ModelForm* for the *Leg model*.

LegFormSet

A formset which can be used to display multiple *Leg forms*. Useful when creating transactions.

1.6.4 Money Utilities

Ratio Split

hordak.utilities.money.**ratio_split**(*amount*, *ratios*)

Split in_value according to the ratios specified in *ratios*

This is special in that it ensures the returned values always sum to in_value (i.e. we avoid losses or gains due to rounding errors). As a result, this method returns a list of *Decimal* values with length equal to that of *ratios*.

Examples

```
>>> from hordak.utilities.money import ratio_split
>>> from decimal import Decimal
>>> ratio_split(Decimal('10'), [Decimal('1'), Decimal('2')])
[Decimal('3.33'), Decimal('6.67')]
```

Note the returned values sum to the original input of `10`. If we were to do this calculation in a naive fashion then the returned values would likely be `3.33` and `6.66`, which would sum to `9.99`, thereby loosing `0.01`.

Parameters

- **amount** (*Decimal*) – The amount to be split
- **ratios** (*list[Decimal]*) – The ratios that will determine the split

Returns: list(Decimal)

1.6.5 Currency Utilities

Contents

Overview

Hordak features multi currency support. Each account in Hordak can support one or more currencies. Hordak does provide currency conversion functionality, but should be as part of the display logic only. It is also a good idea to make it clear to users that you are showing converted values.

The preference for Hordak internals is to always store & process values in the intended currency. This is because currency conversion is an inherently lossy process. Exchange rates vary over time, and rounding errors mean that currency conversions are not reversible without data loss (e.g. ¥176.51 -> $1.54 -> ¥176.20).

Classes

`Money` **instances**:

> The `Money` class is provided by moneyd and combines both an amount and a currency into a single value. Hordak uses these these as the core unit of monetary value.

`Balance` **instances (see below for more details)**:

> An account can hold multiple currencies, and a *Balance* instance is how we represent this.
>
> A *Balance* may contain one or more `Money` objects. There will be precisely one `Money` object for each currency which the account holds.
>
> Balance objects may be added, subtracted etc. This will produce a new *Balance* object containing a union of all the currencies involved in the calculation, even where the result was zero.
>
> Accounts with `is_bank_account=True` may only support a single currency.

Caching

Currency conversion makes use of Django's cache. It is therefore recommended that you setup your Django cache to something other than the default in-memory store.

Currency Exchange

The `currency_exchange()` helper function is provided to assist in creating currency conversion Transactions.

`hordak.utilities.currency.`**`currency_exchange`**(*source*, *source_amount*, *destination*, *destination_amount*, *trading_account*, *fee_destination=None*, *fee_amount=None*, *date=None*, *description=None*)

> Exchange funds from one currency to another
>
> Use this method to represent a real world currency transfer. Note this process doesn't care about exchange rates, only about the value of currency going in and out of the transaction.
>
> You can also record any exchange fees by syphoning off funds to `fee_account` of amount `fee_amount`. Note that the free currency must be the same as the source currency.

Examples

> For example, imagine our Canadian bank has obligingly transferred 120 CAD into our US bank account. We sent CAD 120, and received USD 100. We were also changed 1.50 CAD in fees.
>
> We can represent this exchange in Hordak as follows:

```
from hordak.utilities.currency import currency_exchange

currency_exchange(
    # Source account and amount
    source=cad_cash,
    source_amount=Money(120, 'CAD'),
    # Destination account and amount
    destination=usd_cash,
    destination_amount=Money(100, 'USD'),
    # Trading account the exchange will be done through
    trading_account=trading,
    # We also incur some fees
    fee_destination=banking_fees,
    fee_amount=Money(1.50, 'CAD')
)
```

We should now find that:

1. `cad_cash.balance()` has decreased by `CAD 120`
2. `usd_cash.balance()` has increased by `USD 100`
3. `banking_fees.balance()` is `CAD 1.50`
4. `trading_account.balance()` is `USD 100, CAD -120`

You can perform `trading_account.normalise()` to discover your unrealised gains/losses on currency traded through that account.

Parameters

- **source** (`Account`) – The account the funds will be taken from
- **source_amount** (*Money*) – A `Money` instance containing the inbound amount and currency.
- **destination** (`Account`) – The account the funds will be placed into
- **destination_amount** (*Money*) – A `Money` instance containing the outbound amount and currency
- **trading_account** (`Account`) – The trading account to be used. The normalised balance of this account will indicate gains/losses you have made as part of your activity via this account. Note that the normalised balance fluctuates with the current exchange rate.
- **fee_destination** (`Account`) – Your exchange may incur fees. Specifying this will move incurred fees into this account (optional).
- **fee_amount** (*Money*) – The amount and currency of any incurred fees (optional).
- **description** (*str*) – Description for the transaction. Will default to describing funds in/out & fees (optional).
- **date** (*datetime.date*) – The date on which the transaction took place. Defaults to today (optional).

Returns The transaction created

Return type (*Transaction*)

See also:

You can see the above example in practice in `CurrencyExchangeTestCase.test_fees` in test_currency.py.

Balance

class hordak.utilities.currency.**Balance**(*_money_obs=None*, **args*)
An account balance

Accounts may have multiple currencies. This class represents these multi-currency balances and provides math functionality. Balances can be added, subtracted, multiplied, divided, absolute'ed, and have their sign changed.

Examples

Example use:

```
Balance([Money(100, 'USD'), Money(200, 'EUR')])

# Or in short form
Balance(100, 'USD', 200, 'EUR')
```

Important: Balances can also be compared, but note that this requires a currency conversion step. Therefore it is possible that balances will compare differently as exchange rates change over time.

---

monies()
Get a list of the underlying `Money` instances

Returns A list of zero or money money instances. Currencies will be unique.

Return type ([Money])

currencies()
Get all currencies with non-zero values

normalise(*to_currency*)
Normalise this balance into a single currency

Parameters **to_currency** (*str*) – Destination currency

Returns A new balance object containing a single Money value in the specified currency

Return type (*Balance*)

Exchange Rate Backends

class hordak.utilities.currency.**BaseBackend**
Top-level exchange rate backend

This should be extended to hook into your preferred exchange rate service. The primary method which needs defining is *_get_rate()*.

cache_rate(*currency*, *date*, *rate*)
Cache a rate for future use

get_rate(*currency*, *date*)
Get the exchange rate for `currency` against `_INTERNAL_CURRENCY`

If implementing your own backend, you should probably override *_get_rate()* rather than this.

_get_rate(*currency*, *date*)
Get the exchange rate for `currency` against `INTERNAL_CURRENCY`

You should implement this in any custom backend. For each rate you should call *cache_rate()*.

Normally you will only need to call *cache_rate()* once. However, some services provide multiple exchange rates in a single response, in which it will likely be expedient to cache them all.

Important: Not calling *cache_rate()* will result in your backend service being called for every currency conversion. This could be very slow and may result in your software being rate limited (or, if you pay for your exchange rates, you may get a big bill).

---

class hordak.utilities.currency.**FixerBackend**
Use fixer.io for currency conversions

1.6.6 Exceptions

exception hordak.exceptions.**HordakError**
Abstract exception type for all Hordak errors

exception hordak.exceptions.**AccountingError**
Abstract exception type for errors specifically related to accounting

exception hordak.exceptions.**AccountTypeOnChildNode**
Raised when trying to set a type on a child account

The type of a child account is always inferred from its root account

exception hordak.exceptions.**ZeroAmountError**
Raised when a zero amount is found on a transaction leg

Transaction leg amounts must be none zero.

exception hordak.exceptions.**AccountingEquationViolationError**
Raised if - upon checking - the accounting equation is found to be violated.

The accounting equation is:

0 = Liabilities + Equity + Income - Expenses - Assets

exception hordak.exceptions.**LossyCalculationError**
Raised to prevent a lossy or imprecise calculation from occurring.

Typically this may happen when trying to multiply/divide a monetary value by a float.

exception hordak.exceptions.**BalanceComparisonError**
Raised when comparing a balance to an invalid value

A balance must be compared against another balance or a Money instance

exception hordak.exceptions.**TradingAccountRequiredError**
Raised when trying to perform a currency exchange via an account other than a 'trading' account

exception hordak.exceptions.**InvalidFeeCurrency**
Raised when fee currency does not match source currency

exception hordak.exceptions.**CannotSimplifyError**
Used internally by Currency class

1.7 Notes

A collection of notes and points which may prove useful.

1.7.1 Fixtures

The following should work well for creating fixtures for your Hordak data:

```
./manage.py dumpdata hordak --indent=2 --natural-primary --natural-foreign > fixtures/
↪my-fixture.json
```

1.8 Hordak Changelog

1.8.1 1.1.0

- Multi-currency support

CHAPTER 2

Current limitations

Django Hordak currently does not guarantee sequential primary keys of database entities. IDs are created using regular Postgres sequences, and as a result IDs may skip numbers in certain circumstances. This may conflict with regulatory and audit requirements for some projects. This is an area for future work (1, 2, 3, 4).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

Symbols

A

B

C

D

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Z