Django Hordak Documentation

Release 1.0

Adam Charnock

December 10, 2016

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.

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 >= 1.8, <= 1.10
- Python 2.7, 3.4, 3.5, nightly
- Postgres 9

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

Run the migrations:

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

You should now be able to import from Hordak:

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

1.2 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.

1.2.1 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 most 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).

1.2.2 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.

1.2.3 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

[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.

   - *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.

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.**

1.3 API Documentation

1.3.1 Models

Contents

Design

The core models consist of:

- `Account` - Such as 'Accounts Receivable', a bank account, etc. Accounts can be arranged as a tree structure, where the balance of the parent account is the summation of the balances of all its children.
- `Transaction` - Represents a movement between accounts. Each transaction must have two or more legs.
- `Leg` - Represents a flow of money into (debit) or out of (credit) a transaction. Debits are represented by negative amounts, and credits by positive amounts. The sum of all a transaction's legs must equal zero. This is enforced with a database constraint.

Additionally, there are models which related to the import of external bank statement data:

- `StatementImport` - Represents a simple import of zero or more statement lines relating to a specific `Account`.
- `StatementLine` - Represents a statement line. `StatementLine.create_transaction()` may be called to create a transaction for the statement line.

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`

SmallUUID

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

`name`

str

Name of the account. Required.

`parent`

Account|None

Parent account, nonen if root account

`code`

str

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

`type`

str

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.

`TYPES`

Choices

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

`is_bank_account`

bool

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

classmethod **`validate_accounting_equation`**`()`
Check that all accounts sum to 0

`full_code`
Get the full code for this account

Do this by concatenating this account's code with that of all the parent accounts.

`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*, ***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*, ***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.
- **`**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.

Parameters

- **`to_account`** (`Account`) – The destination account
- **`amount`** (*`Money`*) – The amount to be transferred

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():
    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`

SmallUUID

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

`timestamp`

datetime

The datetime when the object was created.

`date`

date

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

`description`

str

Optional user-provided description

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`

SmallUUID

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

`transaction`

Transaction

Transaction to which the Leg belongs.

account
Account

Account the leg is transferring to/from.

amount
Money

The amount being transferred

description
str

Optional user-provided description

type
str

`hordak.models.DEBIT` or `hordak.models.CREDIT`.

StatementImport

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

uuid
SmallUUID

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

timestamp
datetime

The datetime when the object was created.

bank_account
Account

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

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
SmallUUID

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

timestamp
datetime

The datetime when the object was created.

`date`
date

The date given by the statement line

`statement_import`
StatementImport

The import to which the line belongs

`amount`
Decimal

The amount for the statement line, positive or nagative.

`description`
str

Any description/memo information provided

`transaction`
Transaction

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

1.3.2 Currency

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])

`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

1.3.3 Forms

Contents

Base forms for Django Hordak

This section is a work in progress. Forms will be added as work progresses with the swiftwind and battlecat projects.

1.3.4 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

1.4 Notes

A collection of notes and points which may prove useful.

1.4.1 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
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index