biweeklybudget Documentation

Release 0.5.0

Jason Antman

Oct 28, 2017

Contents

Responsive Flask/SQLAlchemy personal finance app, specifically for biweekly budgeting.

For full documentation, see http://biweeklybudget.readthedocs.io/en/latest/

For screenshots, see http://biweeklybudget.readthedocs.io/en/latest/screenshots.html

For development activity, see https://waffle.io/jantman/biweeklybudget

CHAPTER 1

Overview

biweeklybudget is a responsive (mobile-friendly) Flask/SQLAlchemy personal finance application, specifically targeted at budgeting on a biweekly basis. This is a personal project of mine, and really only intended for my personal use. If you find it helpful, great! But this is provided as-is; I'll happily accept pull requests if they don't mess things up for me, but I don't intend on working any feature requests or bug reports at this time. Sorry.

The main motivation for writing this is that I get paid every other Friday, and have for almost all of my professional life. I also essentially live paycheck-to-paycheck; what savings I have is earmarked for specific purposes, so I budget in periods identical to my pay periods. No existing financial software that I know of handles this, and many of them have thousands of Google results of people asking for it; almost everything existing budgets on calendar months. I spent many years using Google Sheets and a handful of scripts to template out budgets and reconcile transactions, but I decided it's time to just bite the bullet and write something that isn't a pain.

Intended Audience: This is decidedly not an end-user application. You should be familiar with Python/Flask/MySQL. If you're going to use the automatic transaction download functionality, you should be familiar with Hashicorp Vault and how to run a reasonably secure installation of it. I personally don't recommend running this on anything other than your own computer that you physically control, given the sensitivity of the information. I also don't recommend making the application available to anything other than localhost, but if you do, you need to be aware of the security implications. This application is **not** designed to be accessible in any way to anyone other than authorized users (i.e. if you just serve it over the web, someone *will* get your account numbers, or worse).

Important Warning

This software should be considered *alpha* quality at best. At this point, I can't even say that I'm 100% confident it is mathematically correct, balances are right, all scheduled transactions will show up in the right places, etc. I'm going to be testing it for my own purposes, and comparing it against my manual calculations. Until further notice, if you decide to use this, please double-check *everything* produced by it before relying on its output.

Main Features

- Budgeting on a biweekly (fortnightly; every other week) basis, for those of us who are paid that way.

- Periodic (per-pay-period) or standing budgets.
- Optional automatic downloading of transactions/statements from your financial institutions and reconciling transactions (bank, credit, and investment accounts).
- Scheduled transactions - specific date or recurring (date-of-month, or number of times per pay period).
- Tracking of vehicle fuel fills (fuel log) and graphing of fuel economy.
- Cost tracking for multiple projects, including bills-of-materials for them. Optional synchronization from Amazon Wishlists to projects.
- Calculation of estimated credit card payoff amount and time, with configurable payment methods, payment increases on specific dates, and additional payments on specific dates.

CHAPTER 2

Requirements

Note: Alternatively, biweeklybudget is also distributed as a Docker container. Using the dockerized version will eliminate all of these dependencies aside from MySQL (which you can run in another container) and Vault (if you choose to take advantage of the OFX downloading), which you can also run in another container.

- Python 2.7 or 3.3+ (currently tested with 2.7, 3.3, 3.4, 3.5, 3.6 and developed with 3.6)
- Python VirtualEnv and `pip` (recommended installation method; your OS/distribution should have packages for these)
- MySQL, or a compatible database (e.g. MariaDB). biweeklybudget uses SQLAlchemy for database abstraction, but currently specifies some MySQL-specific options, and is only tested with MySQL.
- To use the automated OFX transaction downloading functionality:
  - A running, reachable instance of Hashicorp Vault with your financial institution web credentials stored in it.
  - PhantomJS for downloading transaction data from institutions that do not support OFX remote access ("Direct Connect").

CHAPTER 3

Installation

It's recommended that you install into a virtual environment (virtualenv / venv). See the virtualenv usage documentation for information on how to create a venv.

This app is developed against Python 3.6, but should work back to 2.7. It does not support Python3 < 3.3.

```
mkdir biweeklybudget
virtualenv --python=python3.6 .
source bin/activate
pip install biweeklybudget
```

CHAPTER 4

License

biweeklybudget itself is licensed under the GNU Affero General Public License, version 3. This is specifically intended to extend to anyone who uses the software remotely over a network, the same rights as those who download and install it locally. biweeklybudget makes use of various third party software, especially in the UI and frontend, that is distributed under other licenses. Please see `biweeklybudget/flaskapp/static` in the source tree for further information.

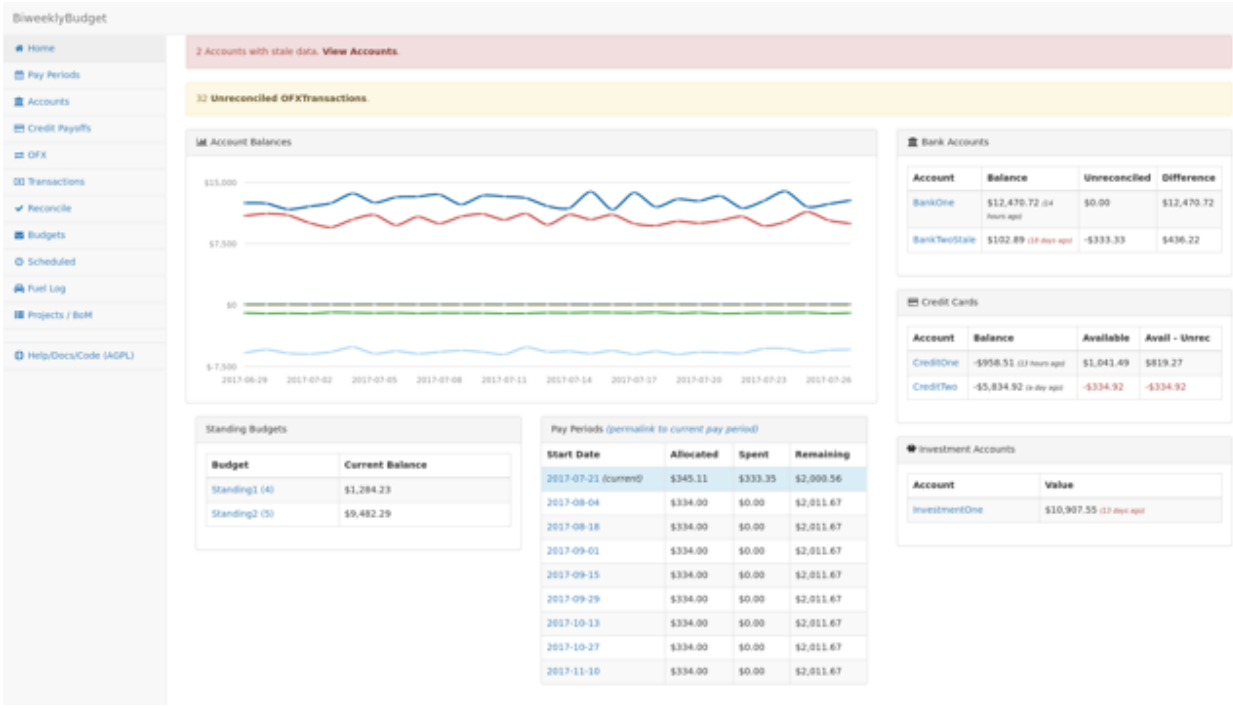
biweeklybudget includes a number of dependencies distributed alongside it, which are licensed and distributed under their respective licenses. See the `biweeklybudget/vendored` directory in the source distribution for further information.

CHAPTER 5

Contents

Screenshots

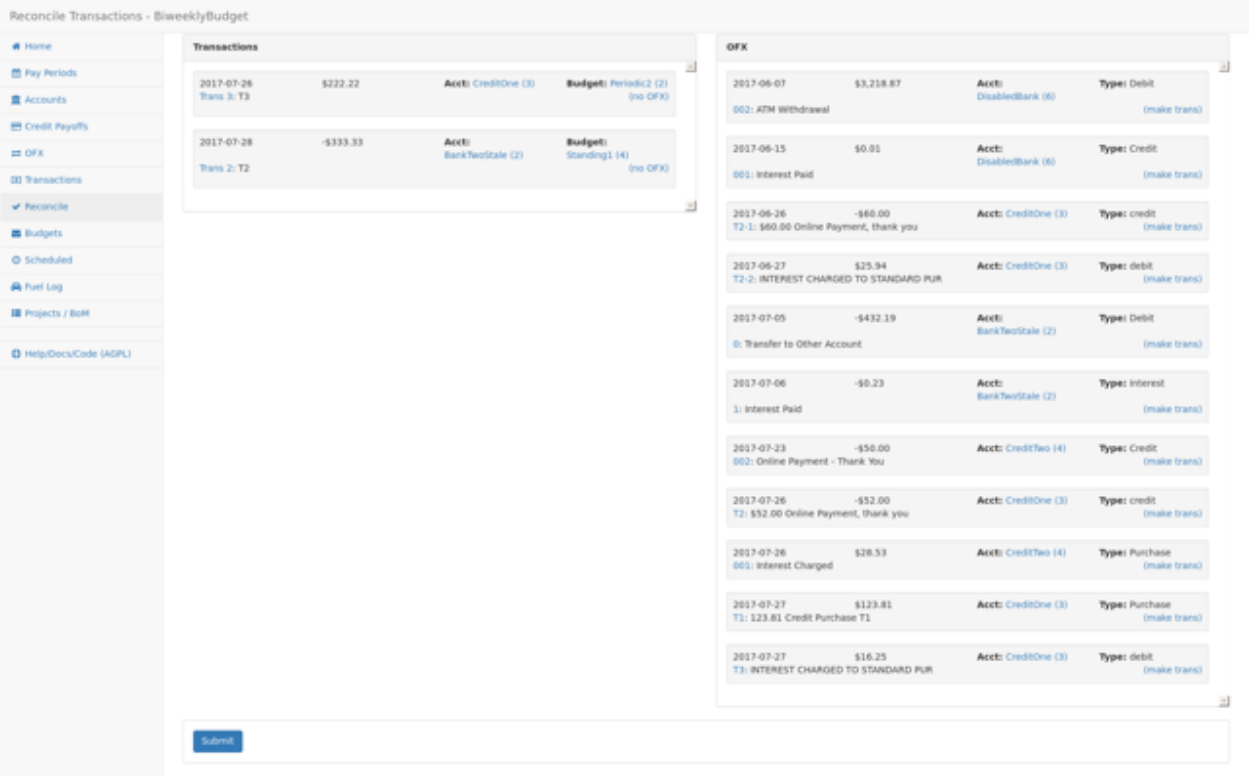
Index Page



Main landing page.

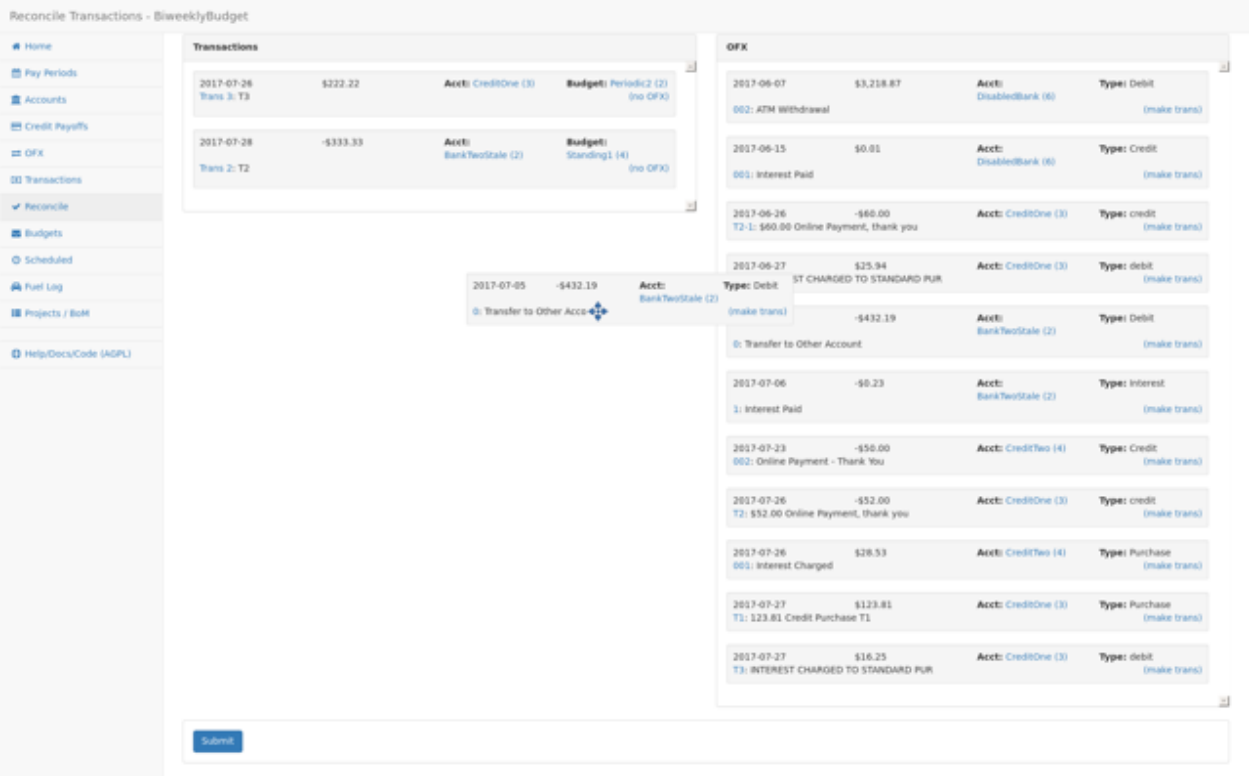
Reconcile Transactions with OFX

OFX Transactions reported by financial institutions can be marked as reconciled with a corresponding Transaction.



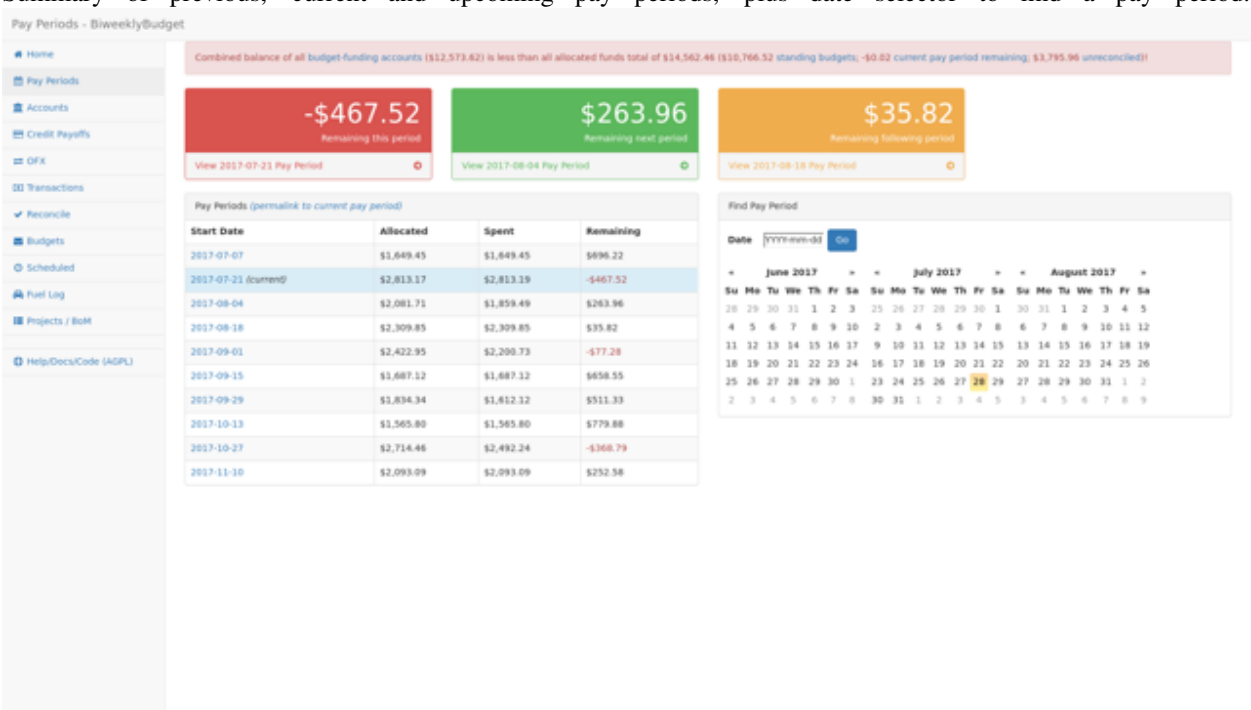
Drag-and-Drop Reconciling

To reconcile an OFX transaction with a Transaction, just drag and drop.



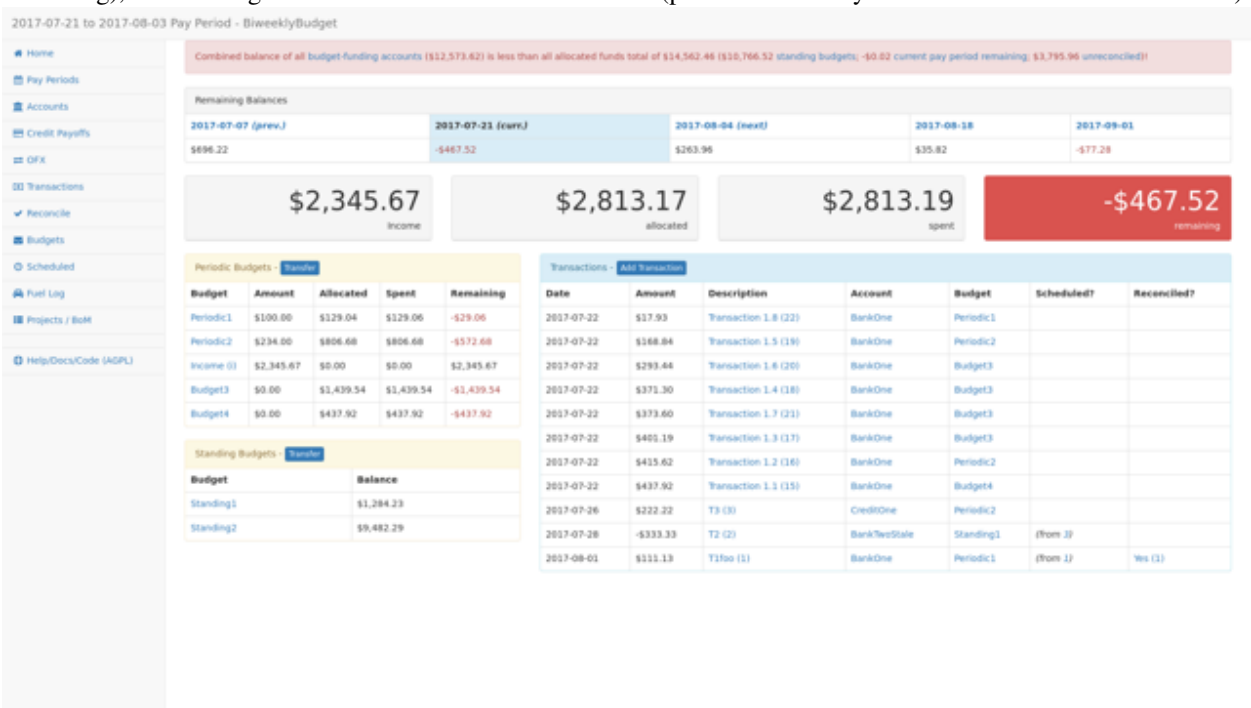
Pay Periods View

Summary of previous, current and upcoming pay periods, plus date selector to find a pay period.



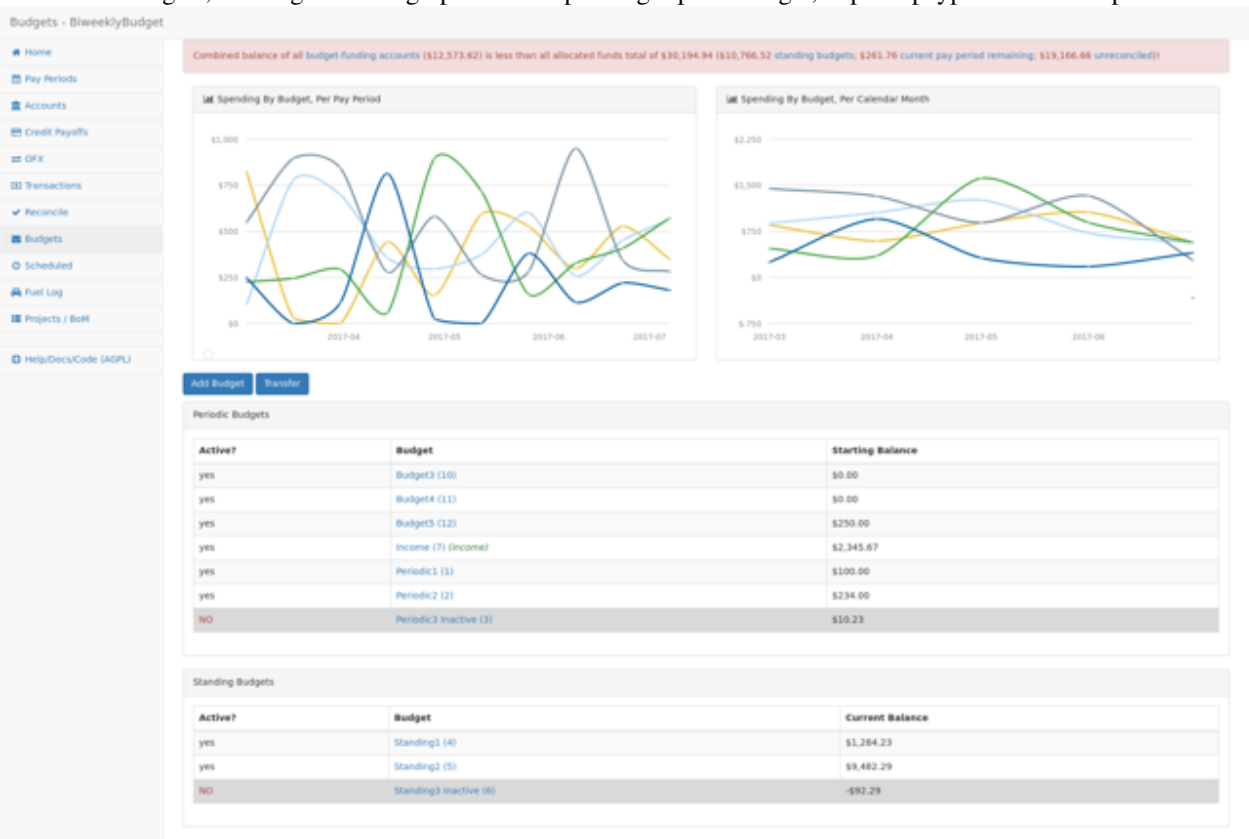
Single Pay Period View

Shows a pay period (current in this example) balances (income, allocated, spent, remaining), budgets and transactions (previous/manually-entered and scheduled).

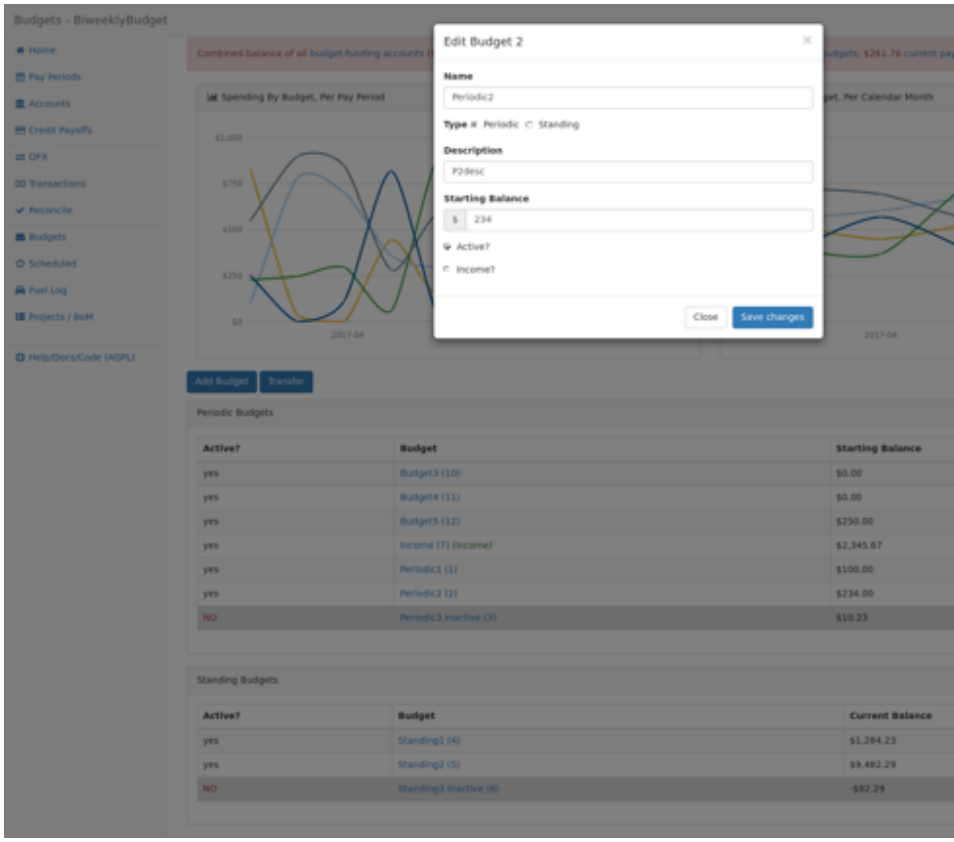


Budgets

List all budgets, along with graphs of spending per budget, per payperiod and per month.

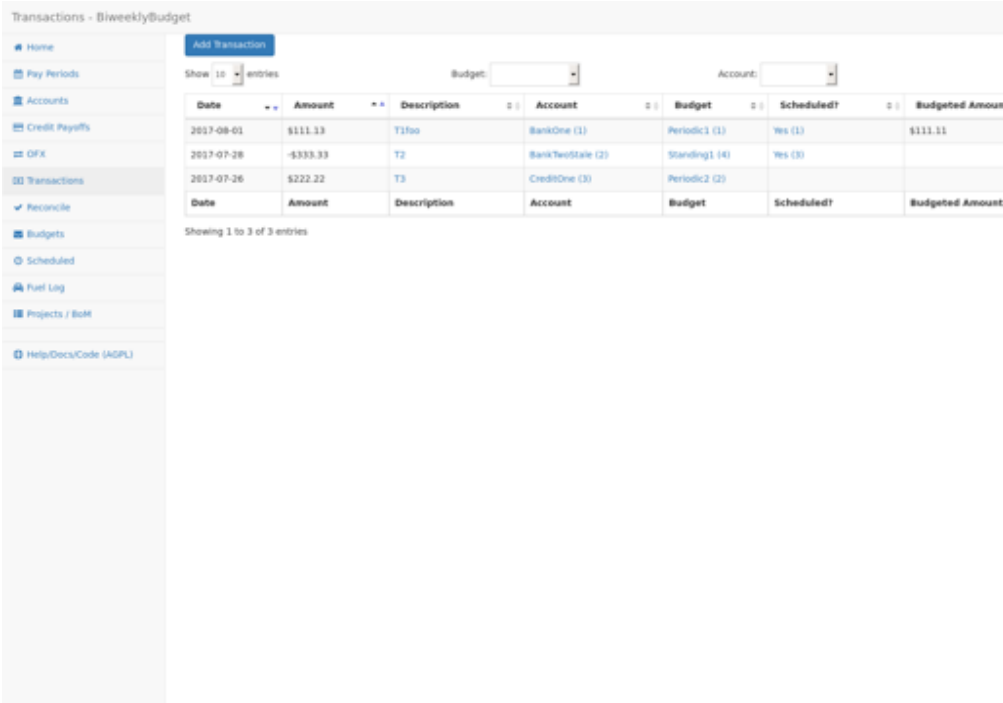


Single Budget View



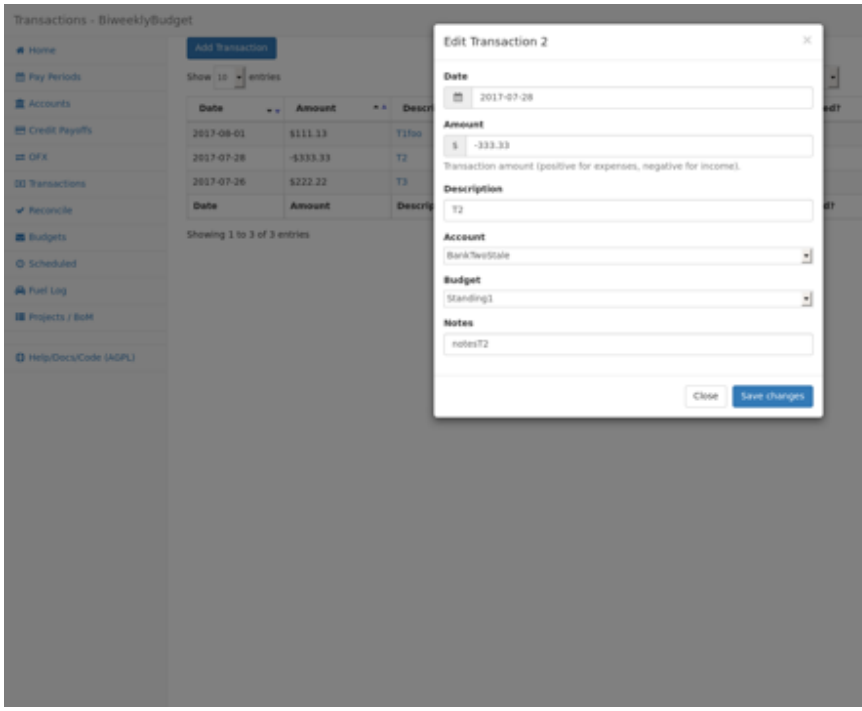
Budget detail modal to view and edit a budget.

Transactions View



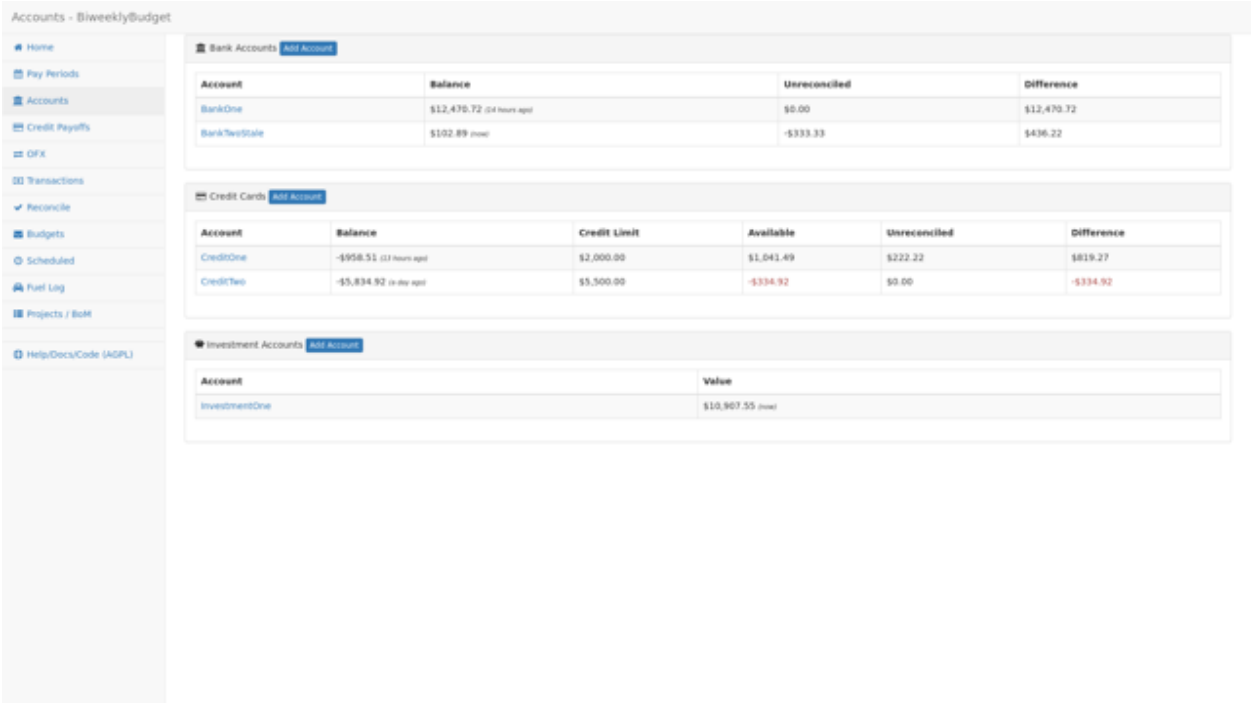
Shows all manually-entered transactions.

Transaction Detail



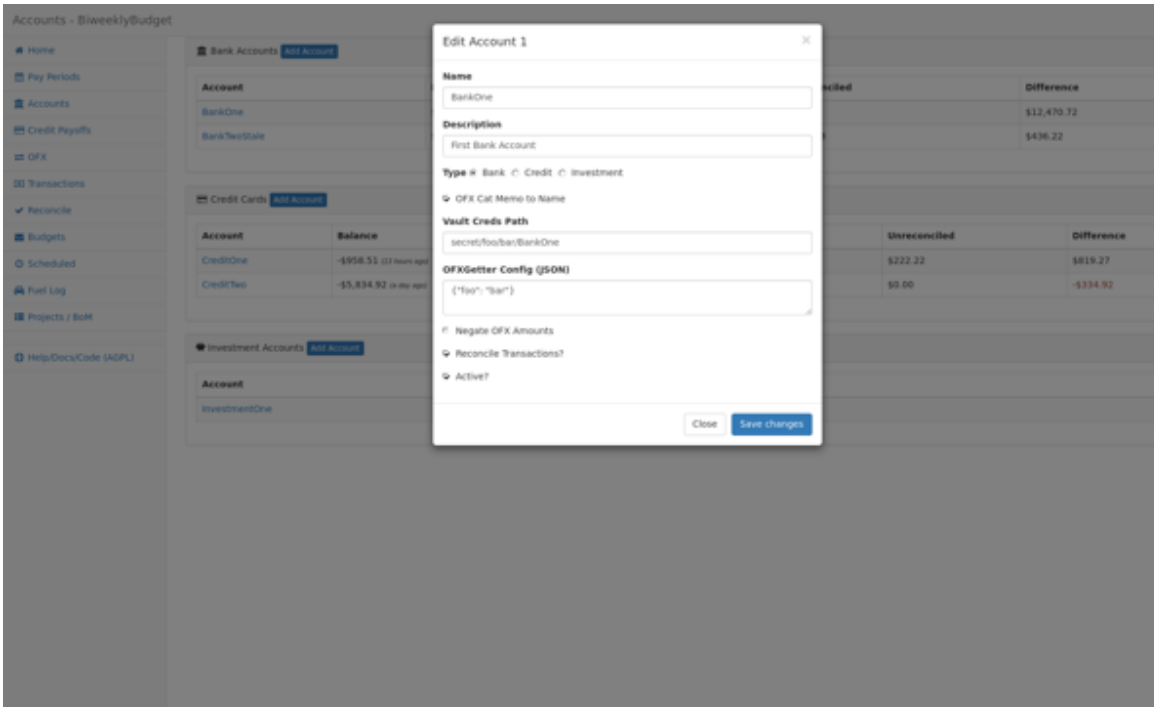
Transaction detail modal to view and edit a transaction.

Accounts View

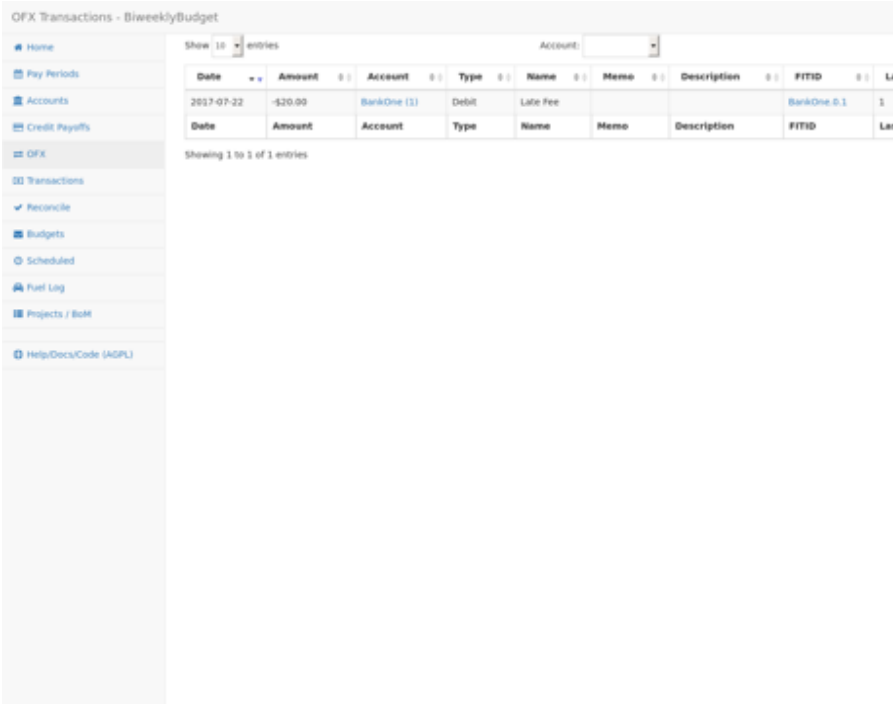


Account Details

Details of a single account.

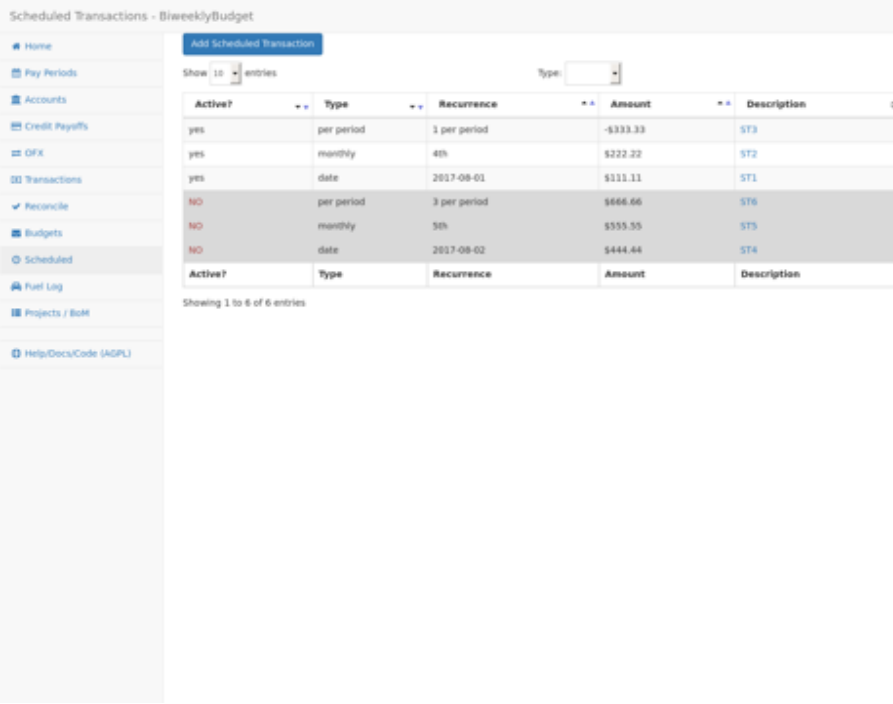


OFX Transactions



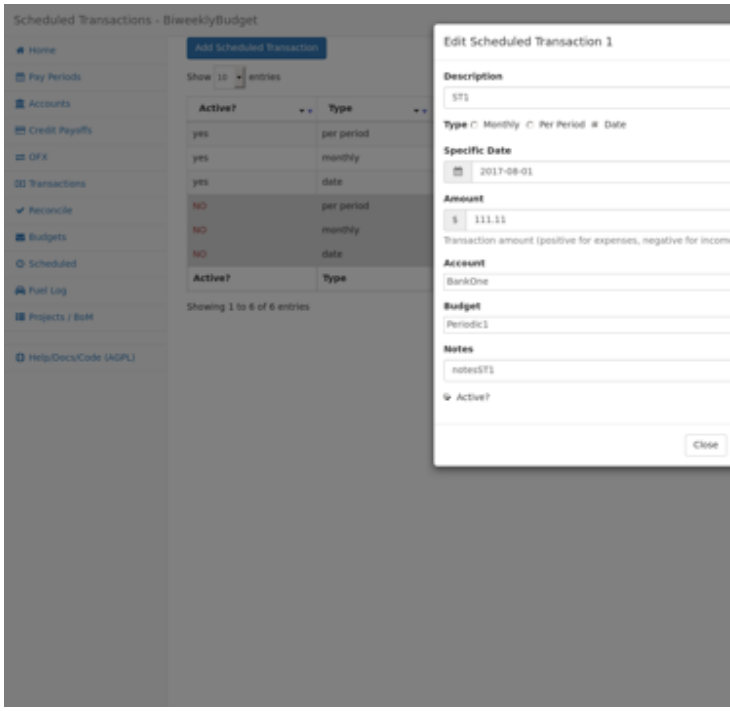
Shows transactions imported from OFX statements.

Scheduled Transactions



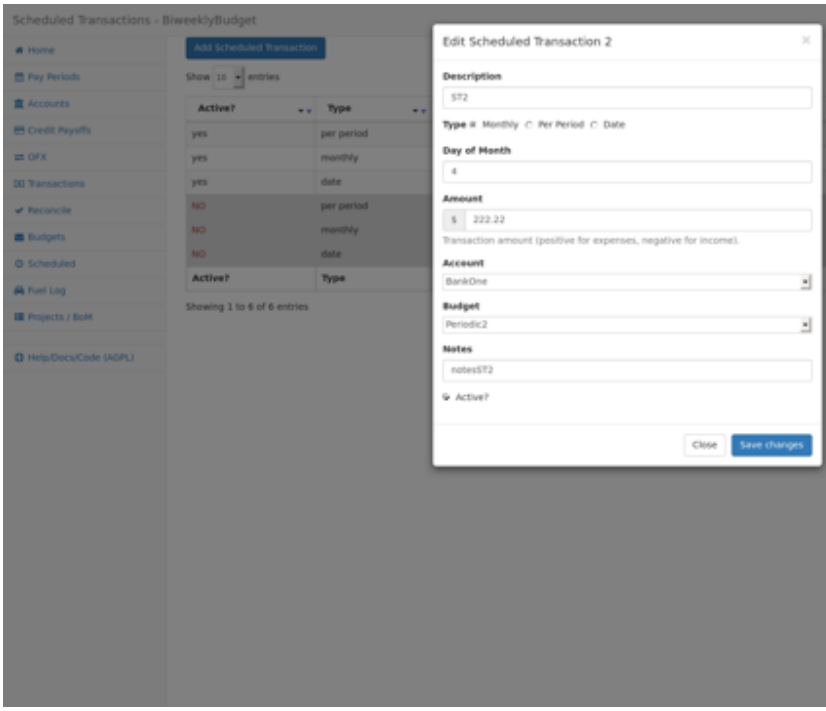
List all scheduled transactions (active and inactive).

Specific Date Scheduled Transaction



Scheduled transactions can occur one-time on a single specific date.

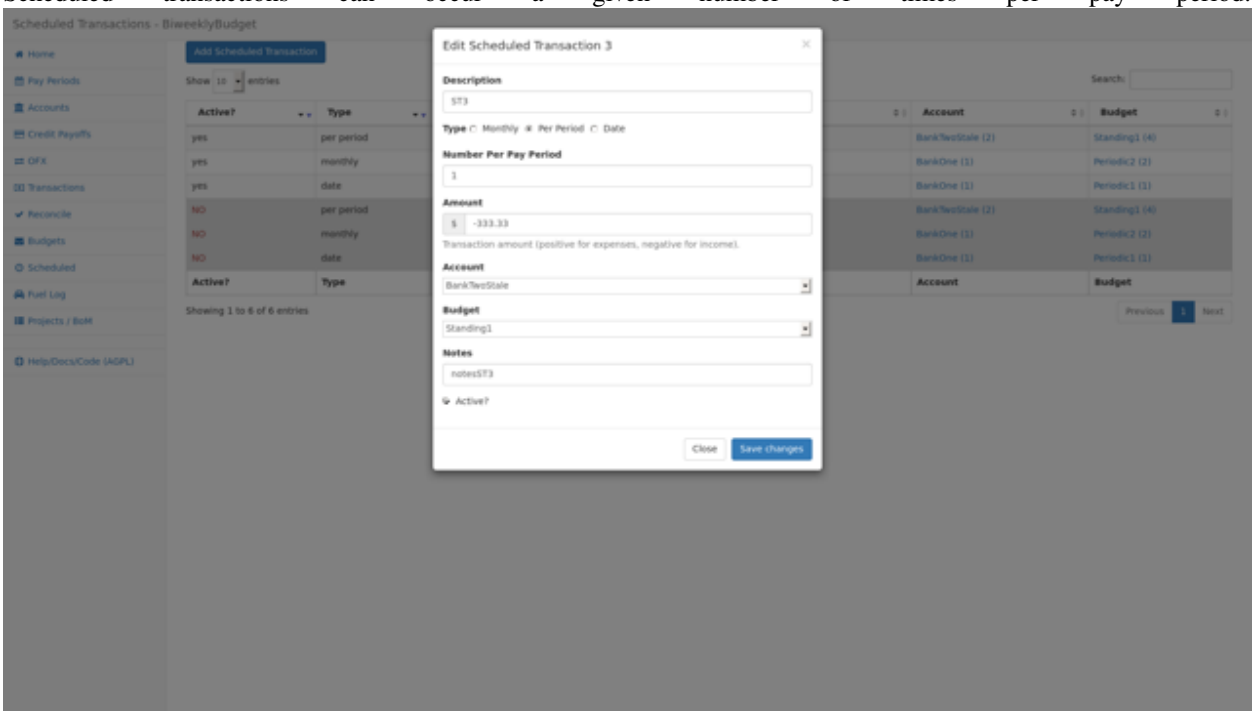
Monthly Scheduled Transaction



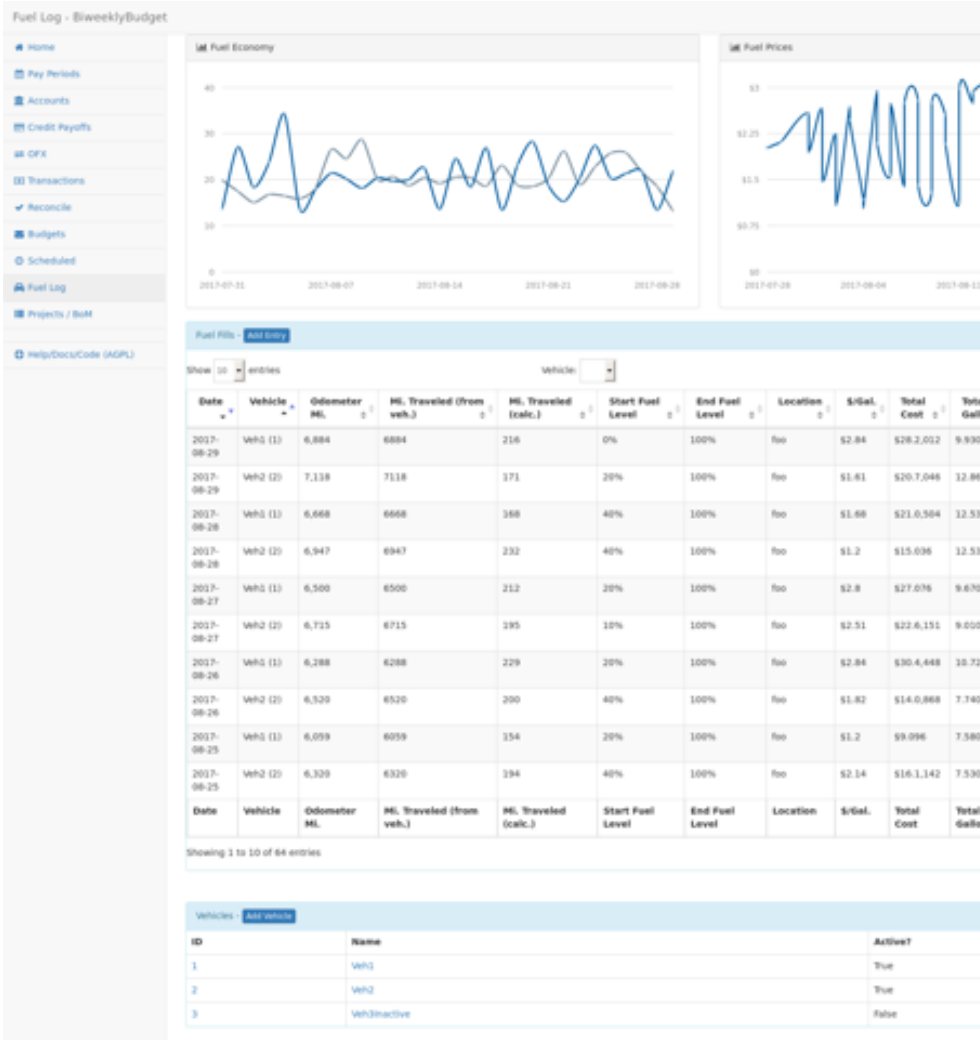
Scheduled transactions can occur monthly on a given date.

Number Per-Period Scheduled Transactions

Scheduled transactions can occur a given number of times per pay period.

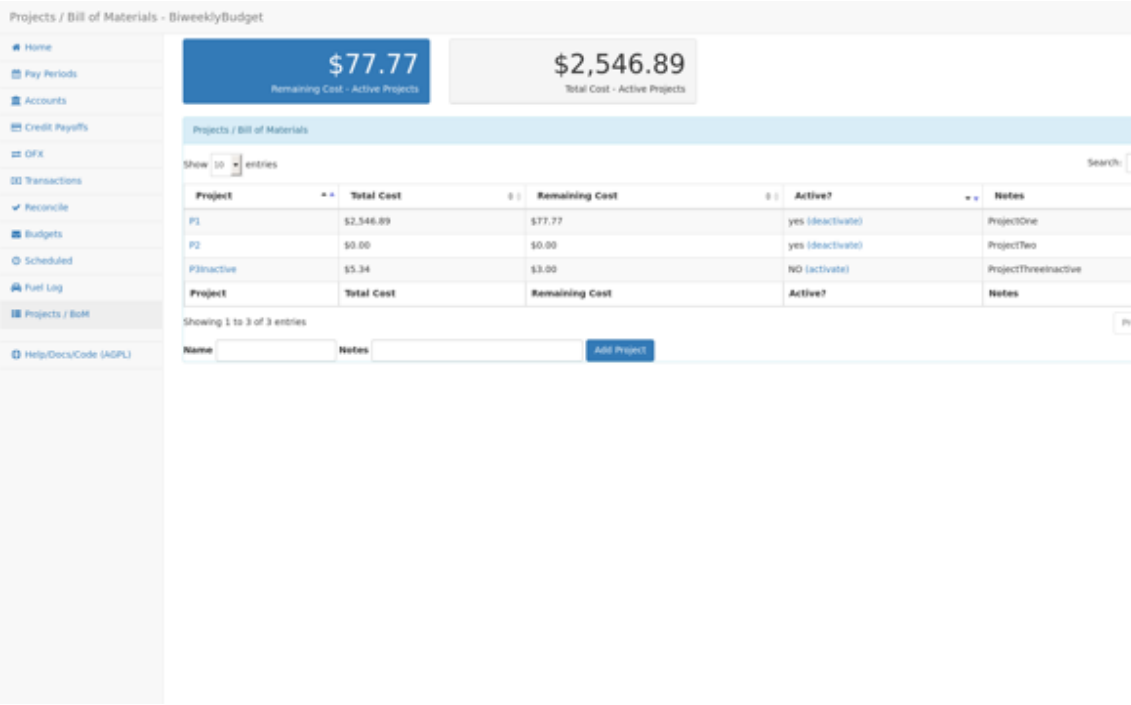


Fuel Log



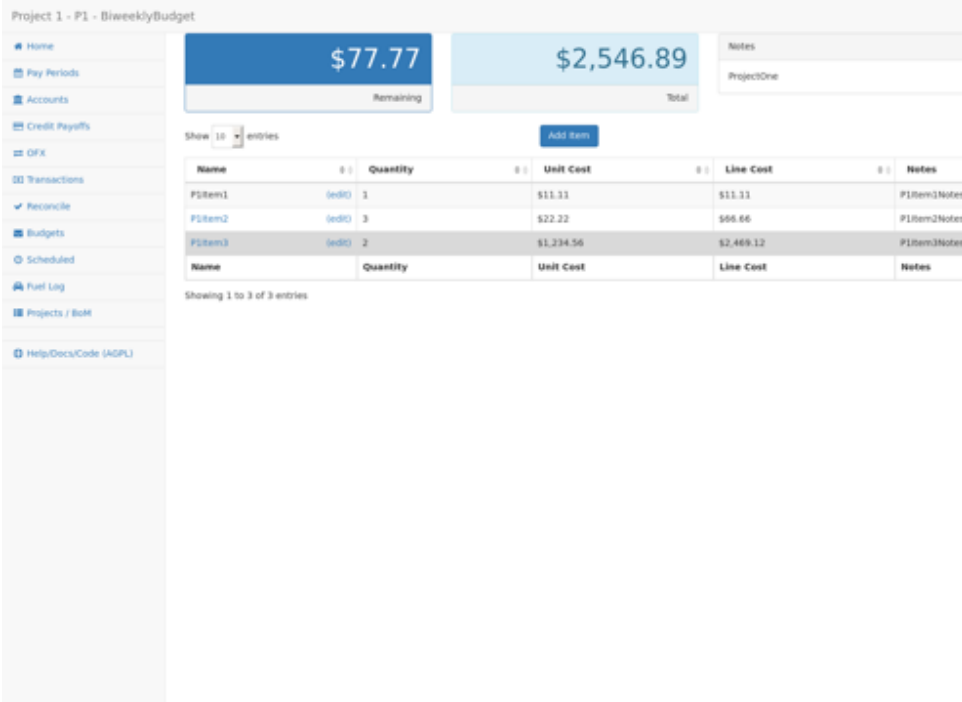
Vehicle fuel log and fuel economy tracking.

Project Tracking



Track projects and their cost.

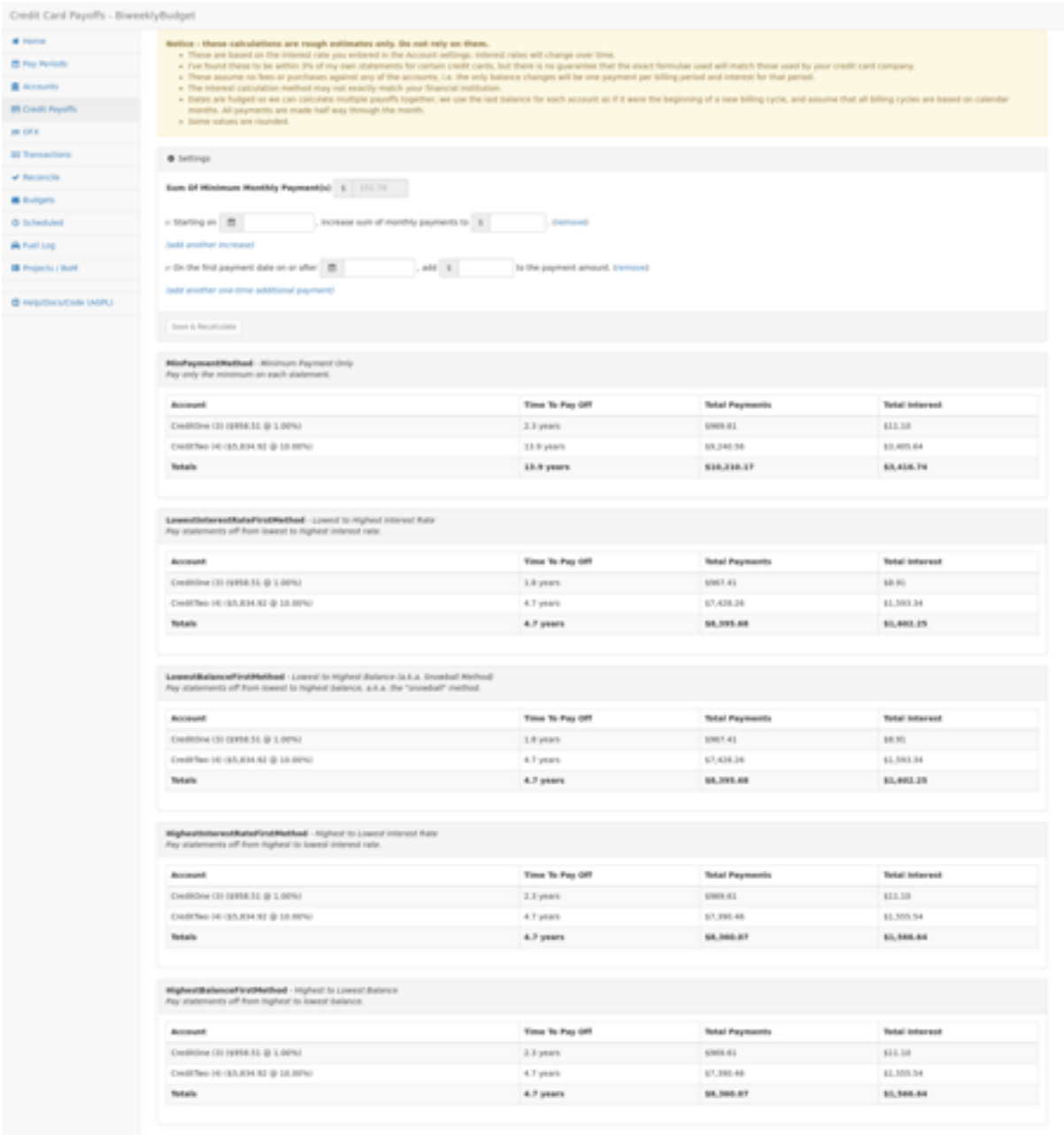
Projects - Bill of Materials



Track individual items/materials for projects.

Credit Card Payoff Calculations

Credit card payoff calculations based on a variety of payment methods, with configurable payment increases over time or one-time additional payment amounts.



Getting Started

Requirements

Note: Alternatively, biweeklybudget is also distributed as a *Docker container*. Using the dockerized version will eliminate all of these dependencies aside from MySQL and Vault (the latter only if you choose to take advantage of the OFX downloading), both of which you can also run in containers.

- Python 2.7 or 3.3+ (currently tested with 2.7, 3.3, 3.4, 3.5, 3.6 and developed with 3.6)
- Python VirtualEnv and `pip` (recommended installation method; your OS/distribution should have packages for these)
- MySQL, or a compatible database (e.g. MariaDB). biweeklybudget uses SQLAlchemy for database abstraction, but currently specifies some MySQL-specific options, and is only tested with MySQL.
- To use the automated OFX transaction downloading functionality:
  - A running, reachable instance of Hashicorp Vault with your financial institution web credentials stored in it.
  - PhantomJS for downloading transaction data from institutions that do not support OFX remote access ("Direct Connect").

Installation

It's recommended that you install into a virtual environment (virtualenv / venv). See the virtualenv usage documentation for information on how to create a venv.

This app is developed against Python 3.6, but should work back to 2.7. It does not support Python3 < 3.3.

```
mkdir biweeklybudget
virtualenv --python=python3.6 .
source bin/activate
pip install biweeklybudget
```

Important Note: Anyone who's using this project for actual data should install from the package on PyPI. While the `master` branch of the git repository is always in a runnable state, there is no guarantee that data will be not be harmed by upgrading directly to master. Specifically, database migrations are only compatible between released versions; `master` is considered a pre-release/development version, and can have migrations removed or altered in breaking ways between official releases.

Upgrading

Documentation for upgrades depends on how you've installed and run biweeklybudget:

- For non-docker installations, see *Flask Application - Database Migrations*
- For Docker installations, no special action is needed.
- For development installations, see *Development - Alembic DB Migrations*

In all cases, you should always perform a full backup of your database before an upgrade.

Configuration

biweeklybudget can take its configuration settings via either constants defined in a Python module or environment variables. Configuration in environment variables always overrides configuration from the settings module.

Settings Module

`biweeklybudget.settings` imports all globals/constants from a module defined in the `SETTINGS_MODULE` environment variable. The recommended way to configure this is to create your own separate Python package for customization (either in a private git repository, or just in a directory on your computer) and install this package into

the same virtualenv as biweeklybudget. You then set the `SETTINGS_MODULE` environment variable to the Python module/import path of this module (i.e. the dotted path, like `packagename.modulename`).

Once you've created the customization package, you can install it in the virtualenv with `pip install -e <git URL>` (if it is kept in a git repository) or `pip install -e <local path>`.

This customization package can also be used for *Loading Data* during development, or implementing *Custom OFX Downloading via Selenium*. It is the recommended configuration method if you need to include more logic than simply defining static configuration settings.

Environment Variables

Every configuration setting can also be specified by setting an environment variable with the same name; these will override any settings defined in a `SETTINGS_MODULE`, if specified. Note that some environment variables require specific formatting of their values; see the *`settings module documentation`* for a list of these variables and the required formats.

Running Locally

Setup

```
source bin/activate
export SETTINGS_MODULE=<settings module>
```

It's recommended that you create an alias to do this for you. Alternatively, instead of setting `SETTINGS_MODULE`, you can export the required environment variables (see above).

Flask

For information on the Flask application and on running the Flask development server, see *Flask App*.

Running In Docker

Biweeklybudget is also distributed as a docker image, to make it easier to run without installing as many *Requirements*.

You can pull the latest version of the image with `docker pull jantman/biweeklybudget:latest`, or a specific release version `X.Y.Z` with `docker pull jantman/biweeklybudget:X.Y.Z`. It is recommended that you run a specific version number, and that you make sure to perform a database backup before upgrading.

The only dependencies for a Docker installation are:

- MySQL, which can be run via Docker (MariaDB official image recommended) or local on the host
- Vault, if you wish to use the OFX downloading feature, which can also be run via Docker

Important Note: If you run MySQL and/or Vault in containers, please make sure that their data is backed up and will not be removed.

The image runs with the tini init wrapper and uses gunicorn under Python 3.6 to serve the web UI, exposed on port 80. Note that, while it runs with 4 worker threads, there is no HTTP proxy in front of Gunicorn and this image is intended for local network use by a single user/client. The image also automatically runs database migrations in a safe manner at start, before starting the Flask application.

For ease of running, the image defaults the `SETTINGS_MODULE` environment variable to `biweeklybudget.settings_example`. This allows leveraging the environment variable *configuration* overrides so that you need only specify configuration options that you want to override from settings_example.py.

For ease of running, it's highly recommended that you put your configuration in a Docker-readable environment variables file.

Environment Variable File

In the following examples, we reference the following environment variable file. It will override settings from settings_example.py as needed; specifically, we need to override the database connection string, pay period start date and reconcile begin date. In the examples below, we would save this as `biweeklybudget.env`:

```
DB_CONNSTRING=mysql+pymysql://USERNAME:PASSWORD@HOST:PORT/DBNAME?charset=utf8mb4
PAY_PERIOD_START_DATE=2017-03-28
RECONCILE_BEGIN_DATE=2017-02-15
```

Containerized MySQL Example

This assumes that you already have a MySQL database container running with the container name "mysql" and exposing port 3306, and that we want the biweeklybudget web UI served on host port 8080:

In our `biweeklybudget.env`, we would specify the database connection string for the "mysql" container:

```
DB_CONNSTRING=mysql+pymysql://USERNAME:PASSWORD@mysql:3306/DBNAME?charset=utf8mb4
```

And then run biweeklybudget:

```
docker run --name biweeklybudget --env-file biweeklybudget.env \
-p 8080:80 --link mysql jantman/biweeklybudget:latest
```

Host-Local MySQL Example

It is also possible to use a MySQL server on the physical (Docker) host system. To do so, you'll need to know the host system's IP address. On Linux when using the default "bridge" Docker networking mode, this will coorespond to a `docker0` interface on the host system. The Docker documentation on adding entries to the Container's hosts file provides a helpful snippet for this (on my systems, this results in `172.17.0.1`):

```
ip -4 addr show scope global dev docker0 | grep inet | awk '{print $2}' | cut -d / -f␣
↪1
```

In our `biweeklybudget.env`, we would specify the database connection string that uses the "dockerhost" hosts file entry, created by the `--add-host` option:

```
# "dockerhost" is added to /etc/hosts via the `--add-host` docker run option
DB_CONNSTRING=mysql+pymysql://USERNAME:PASSWORD@dockerhost:3306/DBNAME?charset=utf8mb4
```

So using that, we could run biweeklybudget listening on port 8080 and using our host's MySQL server (on port 3306):

```
docker run --name biweeklybudget --env-file biweeklybudget.env \
--add-host="dockerhost:$(ip -4 addr show scope global dev docker0 | grep inet | awk '
↪{print $2}' | cut -d / -f 1)" \
-p 8080:80 jantman/biweeklybudget:latest
```

You may need to adjust those commands depending on your operating system, Docker networking mode, and MySQL server.

Settings Module Example

If you need to provide biweeklybudget with more complicated configuration, this is still possible via a Python settings module. The easiest way to inject one into the Docker image is to mount a python module directly into the biweeklybudget package directory. Assuming you have a custom settings module on your local machine at `/opt/biweeklybudget-settings.py`, you would run the container as shown below to mount the custom settings module into the container and use it. Note that this example assumes using MySQL in another container; adjust as necessary if you are using MySQL running on the Docker host:

```
docker run --name biweeklybudget -e SETTINGS_MODULE=biweeklybudget.mysettings \
-v /opt/biweeklybudget-settings.py:/app/lib/python3.6/site-packages/biweeklybudget/
↪mysettings.py \
-p 8080:80 --link mysql jantman/biweeklybudget:latest
```

Note on Locales

biweeklybudget uses Python's locale module to format currency. This requires an appropriate locale installed on the system. The docker image distributed for this package only includes the `en_US.UTF-8` locale. If you need a different one, please cut a pull request against `docker_build.py`.

Running ofxgetter in Docker

If you wish to use the *ofxgetter* script inside the Docker container, some special settings are needed:

1. You must mount the statement save path (*`STATEMENTS_SAVE_PATH`*) into the container.
2. You must mount the Vault token file path (*`TOKEN_PATH`*) into the container.
3. You must set either the `VAULT_ADDR` environment variable, or the *`VAULT_ADDR`* setting.

As an example, for using ofxgetter with *`STATEMENTS_SAVE_PATH`* in your settings file set to `/statements` and *`TOKEN_PATH`* set to `/.token` (root paths used here for simplicity in the example), you would add to your `docker run` command:

```
-v /statements:/statements \
-v /.token:/.token
```

Assuming your container was running with `--name biweeklybudget`, you could run ofxgetter (e.g. via cron) as:

We run explicitly in the statements directory so that if `ofxgetter` encounters an error when using a *`ScreenScraper`* class, the screenshots and HTML output will be saved to the host filesystem.

Command Line Entrypoints and Scripts

biweeklybudget provides the following setuptools entrypoints (command-line script wrappers in `bin/`). First setup your environment according to the instructions above.

- `bin/db_tester.py` - Skeleton of a script that connects to and inits the DB. Edit this to use for one-off DB work. To get an interactive session, use `python -i bin/db_tester.py`.
- `loaddata` - Entrypoint for dropping **all** existing data and loading test fixture data, or your base data. This is an awful, manual hack right now.

- `ofxbackfiller` - Entrypoint to backfill OFX Statements to DB from disk.
- `ofxgetter` - Entrypoint to download OFX Statements for one or all accounts, save to disk, and load to DB. See *OFX*.
- `wishlist2project` - For any projects with "Notes" fields matching an Amazon wishlist URL of a public wishlist (`^https://www.amazon.com/gp/registry/wishlist/`), synchronize the wishlist items to the project. Requires `wishlist==0.1.2`.

Application Usage

This documentation is a work in progress. I suppose if anyone other than me ever tries to use this, I'll document it a bit more.

Pay Periods

Flask Application

Running Flask Development Server

Flask comes bundled with a builtin development server for fast local development and testing. This is an easy way to take biweeklybudget for a spin, but carries some important and critical warnings if you use it with real data. For upstream documentation, see the Flask Development Server docs. Please note that the development server is a single process and defaults to single-threaded, and is only realistically usable by one user.

1. First, setup your environment per *Getting Started - Setup*.
2. `export FLASK_APP="biweeklybudget.flaskapp.app"`
3. If you're running against an existing database, see important information in the "Database Migrations" section, below.
4. `flask --help` for information on usage:

- Run App: `flask run`
- Run with debug/reload: `flask rundev`

To run the app against the acceptance test database, use: `DB_CONNSTRING='mysql+pymysql://budgetTester@127.0.0.1:3306/budgettest?charset=utf8mb4' flask run`

By default, Flask will only bind to localhost. If you want to bind to all interfaces, you can add `--host=0.0.0.0` to the `flask run` commands. Please be aware of the implications of this (see "Security", below).

If you wish to run the flask app in a multi-process/thread/worker WSGI container, be sure that you run the `initdb` entrypoint before starting the workers. Otherwise, it's likely that all workers will attempt to create the database tables or run migrations at the same time, and fail.

Database Migrations

If you run the Flask application (whether in the flask development server or a separate WSGI container) against an existing database and there are unapplied Alembic database migrations, it's very likely that multiple threads or processes will attempt to perform the same migrations at the same time, and leave the database in an inconsistent and unusable state. As such, there are two important warnings:

1. Always be sure that you have a recent database backup before upgrading.
2. You must manually trigger database migrations before starting Flask. This can be done by running the `initdb` console script provided by the biweeklybudget package (`bin/initdb` in your virtualenv).

Security

This code hasn't been audited. It might have SQL injection vulnerabilities in it. It might dump your bank account details in HTML comments. Anything is possible!

To put it succinctly, this was written to be used by me, and me only. It was written with the assumption that anyone who can possibly access any of the application at all, whether in a browser or locally, is authorized to view and/or edit anything and everything related to the application (configuration, everything in the database, everything in Vault if it's being used). If you even think about making this accessible to anything other than localhost on a computer you physically own, it's entirely up to you how you secure it, but make sure you do it really well.

OFX Transaction Downloading

biweeklybudget has the ability to download OFX transaction data from your financial institutions, either manually or automatically (via an external command scheduler such as `cron`).

There are two overall methods of downloading transaction data; for banks that support the OFX protocol, statement data can be downloaded using HTTP only, via the ofxclient project (note we vendor-in a fork with some bug fixes). For banks that do not support the OFX protocol and require you to use their website to download OFX format statements, biweeklybudget provides a base *`ScreenScraper`* class that can be used to develop a selenium-based tool to automate logging in to your bank's site and downloading the OFX file.

In order to use either of these methods, you must have an instance of Hashicorp Vault running and have your login credentials stored in it.

Important Note on Transaction Downloading

biweeklybudget includes support for automatically downloading transaction data from your bank. Credentials are stored in an instance of Hashicorp Vault, as that is a project the author has familiarity with, and was chosen as the most secure way of storing and retrieving secrets non-interactively. Please keep in mind that it is your decision and your decision alone how secure your banking credentials are kept. What is considered acceptable to the author of this program may not be acceptably secure for others; it is your sole responsibility to understand the security and privacy implications of this program as well as Vault, and to understand the risks of storing your banking credentials in this way.

Also note that biweeklybudget includes a base class (*`ScreenScraper`*) intended to simplify developing selenium-based browser automation to log in to financial institution websites and download your transactions. Many banks and other financial institutions have terms of service that *explicitly forbid automated or programmatic use of their websites*. As such, it is up to you as the user of this software to determine your bank's policy and abide by it. I provide a base class to help in writing automated download tooling if your institution allows it, but I cannot and will not distribute institution-specific download tooling.

ofxgetter entrypoint

This package provides an `ofxgetter` command line entrypoint that can be used to download OFX statements for one or all Accounts that are appropriately configured. The script used for this provides exit codes and logging suitable

for use via `cron` (it exits non-zero if any accounts failed, and unless options are provided to increase verbosity, only outputs the number of accounts successfully downloaded as well as any errors).

Vault Setup

Configuring and running Vault is outside the scope of this document. Once you have a Vault installation running and appropriately secured (you shouldn't be using the dev server unless you want to lose all your data every time you reboot) and have given biweeklybudget access to a valid token stored in a file somewhere, you'll need to ensure that your username and password data is stored in Vault in the proper format (`username` and `password` keys). If you happen to use LastPass to store your passwords, you may find my lastpass2vault.py helpful; run it as `./lastpass2vault.py -vv -f PATH_TO_VAULT_TOKEN LASTPASS_USERNAME` and it will copy all of your credentials from LastPass to Vault, preserving the folder structure.

Configuring Accounts for Downloading with ofxclient

1. Use the `ofxclient` CLI to configure and test your account, according to the upstream documentation.
2. Store the username and password for your account in Vault, as `username` and `password` keys, respectively, of the same secret (path).
3. Convert ~/ofxclient.ini to JSON (this will look something like the example below), removing the `institution.username` and `institution.password` keys (these will be read from Vault at runtime).
4. If there is no sensitive information in the resulting JSON, store the JSON string in the *`ofxgetter_config_json`* attribute of the appropriate *`Account`* object. This can be done via the `/accounts` view in the Web UI. If there *is* sensitive information in the ofxclient configuration JSON, you can store the entire JSON configuration in an additional key on the Vault secret, and then set the `ofxgetter_config_json` attribute to `{"key": "NameOfVaultKeyWithJSON"}`.

A working configuration for a Bank account might look something like this:

```
{
    "routing_number": "012345678",
    "account_type": "CHECKING",
    "description": "Checking",
    "number": "111222333",
    "local_id": "f0a14074d33cdf83b4a099bc322dbe2fe19680ca1719425b33de5022",
    "institution": {
        "client_args": {
            "app_version": "2200",
            "app_id": "QWIN",
            "ofx_version": "103",
            "id": "f87217350cc341e2ba7407cf99dcdede"
        },
        "description": "MyBank",
        "url": "https://ofx.MyBank.com",
        "local_id": "e51fb78f88580a1c2e3bb65bd59495384388abda8796c9bf06dcf",
        "broker_id": "",
        "org": "ORG",
        "id": "98765"
    }
}
```

Configuring Accounts for Downloading with Selenium

In your *customization package <_getting_started.customization>*, subclass `ScreenScraper`. Override the constructor to take whatever keyword arguments are required, and add those to your account's `ofxgetter_config_json` as shown below. `OfxGetter` will instantiate the class passing it the specified keyword arguments in addition to `username`, `password` and `savedir` keyword arguments. `savedir` is the directory under `STATEMENTS_SAVE_PATH` where the account's OFX statements should be saved. After instantiating the class, `ofxgetter` will call the class's `run()` method with no arguments, and expect to receive an OFX statement string back.

If cookies are a concern, be aware that saving and loading cookies is broken in PhantomJS 2.x. If you need to persist cookies across sessions, look into the `ScreenScraper` class' `load_cookies()` and `save_cookies()` methods.

```
{
    "class_name": "MyScraper",
    "module_name": "budget_customization.myscraper",
    "institution": {},
    "kwargs": {
        "acct_num": "1234"
    }
}
```

This JSON configuration will have the username and password from Vault interpolated as keyword arguments, similar to how they will be added to `institution` for ofxclient accounts. As described in ofxclient accounts #4, above, you can also store the entire JSON configuration in Vault if desired.

Here's a simple, contrived example of such a class:

```
import logging
import time
import codecs
from datetime import datetime

from selenium.common.exceptions import NoSuchElementException

from biweeklybudget.screenscraper import ScreenScraper

logger = logging.getLogger(__name__)

# suppress selenium logging
selenium_log = logging.getLogger("selenium")
selenium_log.setLevel(logging.WARNING)
selenium_log.propagate = True


class MyScraper(ScreenScraper):

    def __init__(self, username, password, savedir='./',
                 acct_num=None, screenshot=False):
        """
        :param username: username
        :type username: str
        :param password: password
        :type password: str
        :param savedir: directory to save OFX in
        :type savedir: str
        :param acct_num: last 4 of account number, as shown on homepage
```

```
        :type acct_num: str
        """
        super(MyScraper, self).__init__(
            savedir=savedir, screenshot=screenshot
        )
        self.browser = self.get_browser('phantomjs')
        self.username = username
        self.password = password
        self.acct_num = acct_num

    def run(self):
        """ download the transactions, return file path on disk """
        logger.debug("running, username={u}".format(u=self.username))
        logger.info('Logging in...')
        try:
            self.do_login(self.username, self.password)
            logger.info('Logged in; sleeping 2s to stabilize')
            time.sleep(2)
            self.do_screenshot()
            self.select_account()
            act = self.get_account_activity()
        except Exception:
            self.error_screenshot()
            raise
        return act

    def do_login(self, username, password):
        self.get_page('http://example.com')
        raise NotImplementedError("login to your bank here")

    def select_account(self):
        self.get_page('http://example.com')
        logger.debug('Finding account link...')
        link = self.browser.find_element_by_xpath(
            '//a[contains(text(), "%s")]' % self.acct_num
        )
        logger.debug('Clicking account link: %s', link)
        link.click()
        self.wait_for_ajax_load()
        self.do_screenshot()

    def get_account_activity(self):
        # some bank-specific stuff here, then we POST to get OFX
        post_list = self.xhr_post_urlencoded(
            post_url, post_data, headers=post_headers
        )
        if not post_list.startswith('OFXHEADER'):
            self.error_screenshot()
            with codecs.open('result', 'w', 'utf-8') as fh:
                fh.write(post_list)
            raise SystemExit("Got non-OFX response")
        return post_list
```

OFX Related Account Settings

The following attributes on the `Account` model effect OFX downloads and how OFX statements are handled:

- *ofxgetter_config_json* - Stores the configuration required for ofxclient- or Selenium-based OFX downloads. See above. This is exposed as the "OFXGetter Config (JSON)" form field when adding or editing accounts through the UI.
- *ofx_cat_memo_to_name* - This is exposed as the "OFX Cat Memo to Name" checkbox when adding or editing accounts through the UI.
- *negate_ofx_amounts* - This is exposed as the "Negate OFX Amounts" checkbox when adding or editing accounts through the UI.

Getting Help

Bugs and Feature Requests

Bug reports and feature requests are happily accepted via the GitHub Issue Tracker. Pull requests are welcome. Issues that don't have an accompanying pull request will be worked on as my time and priority allows.

Development

To install for development:

1. Fork the biweeklybudget repository on GitHub
2. Create a new branch off of master in your fork.

```
$ virtualenv biweeklybudget
$ cd biweeklybudget && source bin/activate
$ pip install -e git+git@github.com:YOURNAME/biweeklybudget.git@BRANCHNAME
↪#egg=biweeklybudget
$ cd src/biweeklybudget
```

The git clone you're now in will probably be checked out to a specific commit, so you may want to `git checkout BRANCHNAME`.

Guidelines

- pep8 compliant with some exceptions (see pytest.ini)
- 100% test coverage with pytest (with valid tests)

Loading Data

The sample data used for acceptance tests is defined in `biweeklybudget/tests/fixtures/sampledata.py`. This data can be loaded by *setting up the environment <_getting_started.setup>* and then using the `loaddata` entrypoint (the following values for options are actually the defaults, but are shown for clarity):

```
loaddata -m biweeklybudget.tests.fixtures.sampledata -c SampleDataLoader
```

This entrypoint will **drop all tables and data** and then load fresh data from the specified class.

If you wish, you can copy `biweeklybudget/tests/fixtures/sampledata.py` to your *customization package <_getting_started.customization>* and edit it to load your own custom data. This should only be required if you plan on dropping and reinitializing the database often.

Testing

Testing is done via pytest, driven by tox.

- testing is as simple as:
  - `pip install tox`
  - `tox`
- If you want to pass additional arguments to pytest, add them to the tox command line after "–". i.e., for verbose pytext output on py27 tests: `tox -e py27 -- -v`

For rapid iteration on tests, you can either use my toxit script to re-run the test commands in an existing tox environment, or you can use the `bin/t` and `bin/ta` scripts to run unit or acceptance tests, respectively, on only one module.

Unit Tests

There are minimal unit tests, really only some examples and room to test some potentially fragile code. Run them via the `^py\d+` tox environments.

Integration Tests

There's a pytest marker for integration tests, effectively defined as anything that might use either a mocked/in-memory DB or the flask test client, but no HTTP server and no real RDBMS. Run them via the `integration` tox environment. But there aren't any of them yet.

Acceptance Tests

There are acceptance tests, which use a real MySQL DB (see the connection string in `tox.ini` and `conftest.py`) and a real Flask HTTP server, and selenium. Run them via the `acceptance` tox environment.

The acceptance tests connect to a local MySQL database using a connection string specified by the `DB_CONNSTRING` environment variable, or defaulting to a DB name and user/password that can be seen in `conftest.py`. Once connected, the tests will drop all tables in the test DB, re-create all models/tables, and then load sample data. After the DB is initialized, tests will run the local Flask app on a random port, and run Selenium backed by PhantomJS.

If you want to run the acceptance tests without dumping and refreshing the test database, export the `NO_REFRESH_DB` environment variable. Setting the `NO_CLASS_REFRESH_DB` environment variable will prevent refreshing the DB after classes that manipulate data; this will cause subsequent tests to fail but can be useful for debugging.

Alembic DB Migrations

This project uses Alembic for DB migrations:

- To generate migrations, run `alembic -c biweeklybudget/alembic/alembic.ini revision --autogenerate -m "message"` and examine/edit then commit the resulting file(s). This must be run *before* the model changes are applied to the DB. If adding new models, make sure to import the model class in `models/__init__.py`.
- To apply migrations, run `alembic -c biweeklybudget/alembic/alembic.ini upgrade head`.
- To see the current DB version, run `alembic -c biweeklybudget/alembic/alembic.ini current`.

- To see migration history, run `alembic -c biweeklybudget/alembic/alembic.ini history`.

Database Debugging

If you set the `SQL_ECHO` environment variable to "true", all SQL run by SQLAlchemy will be logged at INFO level.

To get an interactive Python shell with the database initialized, use `python -i bin/db_tester.py`.

Docker Image Build

Use the `docker` tox environment. See the docstring at the top of `biweeklybudget/tests/docker_build.py` for further information.

Frontend / UI

The UI is based on BlackrockDigital's startbootstrap-sb-admin-2, currently as of the 3.3.7-1 GitHub release. It is currently not modified at all, but should it need to be rebuilt, this can be done with: `pushd biweeklybudget/flaskapp/static/startbootstrap-sb-admin-2 && gulp`

Sphinx also generates documentation for the custom javascript files. This must be done manually on a machine with jsdoc installed, via: `tox -e jsdoc`.

Vendored Requirements

A number of this project's dependencies are or were seemingly abandoned, and weren't responding to bugfix pull requests or weren't pushing new releases to PyPI. This made the installation process painful, as it required `pip install -r requirements.txt` to pull in git requirements.

In an attempt to make installation easier, we've vendored any git requirements in to this repository under `biweeklybudget/vendored/`. The intent is to move these back to `setup.py` requirements when each project includes the fixes we need in its official release on PyPI.

To updated the vendored projects:

1. Update `biweeklybudget/vendored/vendored_requirements.txt`
2. Run `cd biweeklybudget/vendored && install_vendored.sh`
3. Ensure that our main `setup.py` includes all dependencies of the vendored projects.

Release Checklist

1. Open an issue for the release; cut a branch off master for that issue.
2. Verify whether or not DB migrations are needed. If they are, ensure they've been created, tested and verified.
3. Confirm that there are CHANGES.rst entries for all major changes.
4. Rebuild documentation and javascript documentation locally: `tox -e jsdoc,docs`. Commit any changes.
5. Run the Docker image build and tests locally: `tox -e docker`.
6. Ensure that Travis tests passing in all environments.
7. Ensure that test coverage is no less than the last release, and that there are acceptance tests for any non-trivial changes.

8. If there have been any major visual or functional changes to the UI, regenerate screenshots via `tox -e screenshots`.
9. Increment the version number in biweeklybudget/version.py and add version and release date to CHANGES.rst, then push to GitHub.
10. Confirm that README.rst renders correctly on GitHub.
11. Upload package to testpypi:
    - Make sure your ~/.pypirc file is correct (a repo called `test` for https://testpypi.python.org/pypi)
    - `rm -Rf dist`
    - `python setup.py sdist bdist_wheel`
    - `twine upload -r test dist/*`
    - Check that the README renders at https://testpypi.python.org/pypi/biweeklybudget
12. Create a pull request for the release to be merged into master. Upon successful Travis build, merge it.
13. Tag the release in Git, push tag to GitHub:
    - tag the release. for now the message is quite simple: `git tag -a X.Y.Z -m 'X.Y.Z released YYYY-MM-DD'`
    - push the tag to GitHub: `git push origin X.Y.Z`
14. Upload package to live pypi:
    - `twine upload dist/*`
15. Build and push the new Docker image:
    - Check out the git tag: `git checkout X.Y.Z`
    - Build the Docker image: `DOCKER_BUILD_VER=X.Y.Z tox -e docker`
    - Follow the instructions from that script to push the image to the Docker Hub and tag a "latest" version.
16. make sure any GH issues fixed in the release were closed.
17. Log in to readthedocs.org and enable building of the release tag. You may need to re-run another build to get the tag to be picked up.

Changelog

0.5.0 (2017-10-28)

This release includes database migrations.

- Issue #118 - PR to fix bugs in the wishlist dependency package, and vendor that patched version in under `biweeklybudget.vendored.wishlist`.
- Issue #113 - vendor in other git requirements (ofxclient and ofxparse) that seem unmaintained or inactive, so we can install via `pip`.
- Issue #115 - In Transactions view, add ability to filter by budget.
- Change `BiweeklyPayPeriod` class to never convert to floats (always use decimal.Decimal types).

- Issue #124 - Major changes to the `ofxgetter` and `ofxbackfiller` console scripts; centralize all database access in them to the new `biweeklybudget.ofxapi.local.OfxApiLocal` class and allow these scripts to function remotely, interacting with the ReST API instead of requiring direct database access.
- Issue #123 - Modify the Credit Payoffs view to allow removal of Increase and Onetime Payment settings lines.
- Issue #131 - Add better example data for screenshots.
- Issue #117 and #133 - Implement and then revert out a failed attempt at automatic balancing of budgets in the previous pay period.
- Issue #114
  - Add `transfer_id` field and `transfer` relationship to Transaction model, to link the halves of budget transfer transactions in the database. The alembic migration for this release iterates all Transactions in the database, and populates these links based on inferences of the description, date, account_id and notes fields of sequential pairs of Transactions. (Note: this migration would likely miss some links if two transfers were created simultaneously, and ended up with the Transaction IDs interleaved).
  - Identify transfer Transactions on the Edit Transaction modal, and provide link to the matching Transaction.
  - Add graph of spending by budget to Budgets view.
- Issue #133 - Change BiweeklyPayPeriod model to only use actual spent amount when creating remaining amount on payperiods in the past. Previously, all pay periods calculated the overall "remaining" amount as income minus the greater of `allocated` or `spent`; this resulted in pay periods in the past still including allocated-but-not-spent amounts counted against "remaining".

0.4.0 (2017-08-22)

- Have `ofxgetter` enable ofxclient logging when running at DEBUG level (`-vv`).
- Bump ofxclient requirement to my vanguard-fix branch for PR #47.
- Issue #101 - Fix static example amounts on `/projects` view.
- Issue #103 - Show most recent MPG in notification box after adding fuel fill.
- Issue #97 - Fix integration tests that are date-specific and break on certain dates (run all integration tests as if it were a fixed date).
- Issue #104 - Relatively major changes to add calculation of Credit account payoff times and amounts.
- Issue #107 - Fix bug where Budget Transfer modal dialog would always default to current date, even when viewing past or future pay periods.
- Issue #48 - UI support for adding and editing accounts.

0.3.0 (2017-07-09)

- Issue #88 - Add tracking of cost for Projects and Bills of Materials (BoM) for them.
- Add script / entry point to sync Amazon Wishlist with a Project.
- Issue #74 - Another attempt at working over-balance notification.

0.2.0 (2017-07-02)

- Fix `/pay_period_for` redirect to be a 302 instead of 301, add redirect logging, remove some old debug logging from that view.

- Fix logging exception in db_event_handlers on initial data load.
- Switch ofxparse requirement to use upstream repo now that https://github.com/jseutter/ofxparse/pull/127 is merged.
- Issue #83 - Fix 500 error preventing display of balance chart on `/` view when an account has a None ledger balance.
- Issue #86 - Allow budget transfers to periodic budgets.
- Issue #74 - Warning notification for low balance should take current pay period's overall allocated sum, minus reconciled transactions, into account.
- Fix some template bugs that were causing HTML to be escaped into plaintext.
- Issue #15 - Add pay period totals table to index page.
- Refactor form generation in UI to use new FormBuilder javascript class (DRY).
- Fix date-sensitive acceptance test.
- Issue #87 - Add fuel log / fuel economy tracking.

0.1.2 (2017-05-28)

- Minor fix to instructions printed after release build in `biweeklybudget/tests/docker_build.py`
- Issue #61 - Document running `ofxgetter` in the Docker container.
- fix ReconcileRule repr for uncommited (id is None)
- Issue #67 - ofxgetter logging - suppress DB and Alembic logging at INFO and above; log number of inserted and updated transactions.
- Issue #71 - Fix display text next to prev/curr/next periods on `/payperiod/YYYY-mm-dd` view; add 6 more future pay periods to the `/payperiods` table.
- Issue #72 - Add a built-in method for transferring money from periodic (per-pay-period) to standing budgets; add budget Transfer buttons on Budgets and Pay Period views.
- Issue #75 - Add link on payperiod views to skip a ScheduledTransaction instance this period.
- Issue #57 - Ignore future transactions from unreconciled transactions list.
- Transaction model - fix default for `date` field to actually be just a date; previously, Transactions with `date` left as default would attempt to put a full datetime into a date column, and throw a data truncation warning.
- Transaction model - Fix `__repr__` to not throw exception on un-persisted objects.
- When adding or updating the `actual_amount` of a Transaction against a Standing Budget, update the `current_balance` of the budget.
- Fix ordering of Transactions table on Pay Period view, to properly sort by date and then amount.
- Numerous fixes to date-sensitive acceptance tests.
- Issue #79 - Update `/pay_period_for` view to redirect to current pay period when called with no query parameters; add bookmarkable link to current pay period to Pay Periods view.

0.1.1 (2017-05-20)

- Improve ofxgetter/ofxupdater error handling; catch OFX files with error messages in them.

- Issue #62 - Fix phantomjs in Docker image. * Allow docker image tests to run against an existing image, defined by `DOCKER_TEST_TAG`. * Retry MySQL DB creation during Docker tests until it succeeds, or fails 10 times. * Add testing of PhantomJS in Docker image testing; check version and that it actually works (GET a page). * More reliable stopping and removing of Docker containers during Docker image tests.
- Issue #63 - Enable gunicorn request logging in Docker container.
- Switch to my fork of ofxclient in requirements.txt, to pull in ofxclient PR #41
- Issue #64 - Fix duplicate/multiple on click event handlers in UI that were causing duplicate transactions.

0.1.0 (2017-05-07)

- Initial Release

biweeklybudget

biweeklybudget package

Subpackages

biweeklybudget.flaskapp package

Subpackages

biweeklybudget.flaskapp.views package

Submodules

biweeklybudget.flaskapp.views.accounts module

biweeklybudget.flaskapp.views.budgets module

biweeklybudget.flaskapp.views.credit_payoffs module

biweeklybudget.flaskapp.views.example module

biweeklybudget.flaskapp.views.formhandlerview module

biweeklybudget.flaskapp.views.fuel module

biweeklybudget.flaskapp.views.help module

biweeklybudget.flaskapp.views.index module

biweeklybudget.flaskapp.views.ofx module

biweeklybudget.flaskapp.views.payperiods module

biweeklybudget.flaskapp.views.projects module

biweeklybudget.flaskapp.views.reconcile module

biweeklybudget.flaskapp.views.scheduled module

biweeklybudget.flaskapp.views.searchableajaxview module

biweeklybudget.flaskapp.views.transactions module

biweeklybudget.flaskapp.views.utils module

Submodules

biweeklybudget.flaskapp.app module

biweeklybudget.flaskapp.cli_commands module

`biweeklybudget.flaskapp.cli_commands.`**`template_paths`**`()`

Return a list of all Flask app template paths, to auto-reload on change.

from http://stackoverflow.com/a/41666467/211734

Returns list of all template paths

Return type list

biweeklybudget.flaskapp.context_processors module

biweeklybudget.flaskapp.filters module

biweeklybudget.flaskapp.jinja_tests module

biweeklybudget.flaskapp.jsonencoder module

class `biweeklybudget.flaskapp.jsonencoder.`**`MagicJSONEncoder`**`(`*skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, encoding='utf-8', default=None*`)`

Bases: `json.encoder.JSONEncoder`

Customized JSONEncoder class that uses `as_dict` properties on objects to encode them.

`default``(`*o*`)`

biweeklybudget.flaskapp.notifications module

class `biweeklybudget.flaskapp.notifications.`**`NotificationsController`**

Bases: `object`

static **`budget_account_sum`**(*sess=None*)

Return the sum of current balances for all is_budget_source accounts.

Returns Combined balance of all budget source accounts

Return type float

static **`budget_account_unreconciled`**(*sess=None*)

Return the sum of unreconciled txns for all is_budget_source accounts.

Returns Combined unreconciled amount of all budget source accounts

Return type float

static **`get_notifications`**()

Return all notifications that should be displayed at the top of pages, as a list in the order they should appear. Each list item is a dict with keys "classes" and "content", where classes is the string that should appear in the notification div's "class" attribute, and content is the string content of the div.

static **`num_stale_accounts`**(*sess=None*)

Return the number of accounts with stale data.

@TODO This is a hack because I just cannot figure out how to do this natively in SQLAlchemy.

Returns count of accounts with stale data

Return type int

static **`num_unreconciled_ofx`**(*sess=None*)

Return the number of unreconciled OFXTransactions.

Returns number of unreconciled OFXTransactions

Return type int

static **`pp_sum`**(*sess=None*)

Return the overall allocated sum for the current payperiod minus the sum of all reconciled Transactions for the pay period.

Returns overall allocated sum for the current pay period minus the sum of all reconciled Transactions for the pay period.

Return type float

static **`standing_budgets_sum`**(*sess=None*)

Return the sum of current balances of all standing budgets.

Returns sum of current balances of all standing budgets

Return type float

biweeklybudget.models package

Submodules

biweeklybudget.models.account module

class `biweeklybudget.models.account.`**`Account`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

`_sa_class_manager` **= <ClassManager of <class 'biweeklybudget.models.account.Account'> at 7fad57e62ab8>**

`acct_type`

Type of account (Enum *`AcctType`*)

`all_statements`

Relationship to all *`OFXStatement`* for this Account

`apr`

Finance rate (APR) for credit accounts

`balance`

Return the latest AccountBalance object for this Account.

Returns latest AccountBalance for this Account

Return type *biweeklybudget.models.account_balance.AccountBalance*

`credit_limit`

credit limit, for credit accounts

`description`

description

`effective_apr`

Return the effective APR for a credit account. If *`prime_rate_margin`* is not Null, return that added to the current US Prime Rate. Otherwise, return *`apr`*.

Returns Effective account APR

Return type decimal.Decimal

`for_ofxgetter`

Return whether or not this account should be handled by ofxgetter.

Returns whether or not ofxgetter should run for this account

Return type bool

`id`

Primary Key

`interest_class_name`

Name of the *`biweeklybudget.interest._InterestCalculation`* subclass used to calculate interest for this account.

`is_active`

whether or not the account is active and can be used, or historical

`is_budget_source`

Return whether or not this account should be considered a funding source for Budgets.

Returns whether or not this account is a Budget funding source

Return type bool

`is_stale`

Return whether or not there is stale data for this account.

Returns whether or not data for this account is stale

Return type bool

min_payment_class_name
Name of the `biweeklybudget.interest._MinPaymentFormula` subclass used to calculate minimum payments for this account.

name
name for the account

negate_ofx_amounts
For use in reconciling our `Transaction` entries with the account's `OFXTransaction` entries, whether or not to negate the OfxTransaction amount. We enter Transactions with income as negative amounts and expenses as positive amounts, but most bank OFX statements will show the opposite.

ofx_cat_memo_to_name
whether or not to concatenate the OFX memo text onto the OFX name text; for banks like Chase that use the memo for run-on from the name

ofx_statement
Return the latest OFXStatement for this Account.

Returns latest OFXStatement for this Account

Return type *biweeklybudget.models.ofx_statement.OFXStatement*

ofxgetter_config
Return the deserialized ofxgetter_config_json dict.

Returns ofxgetter config

Return type dict

ofxgetter_config_json
JSON-encoded ofxgetter configuration

prime_rate_margin
Margin added to the US Prime Rate to determine APR, for credit accounts.

re_fee
regex for matching transactions as fees

re_interest_charge
regex for matching transactions as interest charges

re_interest_paid
regex for matching transactions as interest paid

re_payment
regex for matching transactions as payments

reconcile_trans
Include Transactions and OFXTransactions from this account when reconciling. Set to False to exclude accounts that are investment, payment only, or otherwise won't have a matching Transaction for each OFXTransaction.

set_balance(***kwargs*)
Create an AccountBalance object for this account and associate it with the account. Add it to the current session.

set_ofxgetter_config(*config*)
Set ofxgetter configuration.

Parameters **config** (*dict*) – ofxgetter configuration

unreconciled
Return a query to match all unreconciled Transactions for this account.

Parameters **db** (*sqlalchemy.orm.session.Session*) – active database session to use for queries

Returns query to match all unreconciled Transactions

Return type sqlalchemy.orm.query.Query

unreconciled_sum
Return the sum of all unreconciled transaction amounts for this account.

Returns sum of amounts of all unreconciled transactions

Return type float

vault_creds_path
path in Vault to read the credentials from

class biweeklybudget.models.account.**AcctType**
Bases: enum.Enum

Bank = 1

Cash = 4

Credit = 2

Investment = 3

Other = 5

_member_map_ = OrderedDict([('Bank', <AcctType.Bank: 1>), ('Credit', <AcctType.Credit: 2>), ('Investment', <AcctT

_member_names_ = ['Bank', 'Credit', 'Investment', 'Cash', 'Other']

_member_type_
alias of object

_value2member_map_ = {1: <AcctType.Bank: 1>, 2: <AcctType.Credit: 2>, 3: <AcctType.Investment: 3>, 4: <AcctTyp

as_dict

biweeklybudget.models.account_balance module

class biweeklybudget.models.account_balance.**AccountBalance**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.account_balance.AccountBalance'> at 7fad5

account
Relationship to *Account* this balance is for

account_id
ID of the account this balance is for

avail
Available balance

avail_date
as-of date for the available balance

id
Primary Key

ledger
Ledger balance, or investment account value, or credit card balance

ledger_date
as-of date for the ledger balance

overall_date
overall balance as of DateTime

biweeklybudget.models.base module

class `biweeklybudget.models.base.`**`ModelAsDict`**
Bases: `object`

as_dict
Return a dict representation of the model.

Returns model's variables/attributes

Return type dict

biweeklybudget.models.budget_model module

class `biweeklybudget.models.budget_model.`**`Budget`**`(`***kwargs*`)`
Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.budget_model.Budget'> at 7fad57e62e30>

current_balance
current balance for standing budgets

description
description

id
Primary Key

is_active
whether active or historical

is_income
whether this is an Income budget (True) or expense (False).

is_periodic
Whether the budget is standing (long-running) or periodic (resets each pay period or budget cycle)

name
name of the budget

starting_balance
starting balance for periodic budgets

biweeklybudget.models.dbsetting module

class biweeklybudget.models.dbsetting.**DBSetting**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.dbsetting.DBSetting'> at 7fad57e623c8>

default_value
Default value - usually JSON

is_json
Whether setting is JSON, or plain text

name
Primary Key

value
Setting value - usually JSON

biweeklybudget.models.fuel module

class biweeklybudget.models.fuel.**FuelFill**(***kwargs*)

Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_previous_entry()
Get the previous fill for this vehicle by odometer reading, or None.

Returns the previous fill for this vehicle, by odometer reading, or None.

Return type *biweeklybudget.models.fuel.FuelFill*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.fuel.FuelFill'> at 7fad57e34740>

calculate_mpg()
Calculate calculated_mpg field.

Returns True if recalculate, False if unable to calculate

Return type bool

calculated_miles
Number of miles actually traveled since the last fill.

calculated_mpg
Calculated MPG, based on last fill

cost_per_gallon
Fuel cost per gallon

date
date of the fill

fill_location
Location of fill - usually a gas station name/address

gallons
Total amount of fuel (gallons)

id
Primary Key

level_after
Fuel level after fill, as a percentage (Integer 0-100)

level_before
Fuel level before fill, as a percentage (Integer 0-100)

notes
Notes

odometer_miles
Odometer reading of the vehicle, in miles

reported_miles
Number of miles the vehicle thinks it's traveled since the last fill.

reported_mpg
MPG as reported by the vehicle itself

total_cost
Total cost of fill

validate_gallons(_, *value*)

validate_odometer_miles(_, *value*)

vehicle
The vehicle

vehicle_id
ID of the vehicle

class biweeklybudget.models.fuel.**Vehicle**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.fuel.Vehicle'> at 7fad57e342a0>

id
Primary Key

is_active
whether active or historical

name
Name of vehicle

biweeklybudget.models.ofx_statement module

class biweeklybudget.models.ofx_statement.**OFXStatement**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.ofx_statement.OFXStatement'> at 7fad57e34

account
Relationship to the *Account* this statement is for

account_id
Foreign key - Account.id - ID of the account this statement is for

acct_type
Textual account type, from the bank (i.e. "Checking")

acctid
Institution's account ID

as_of
Last OFX statement datetime

avail_bal
Available balance

avail_bal_as_of
as-of date for the available balance

bankid
FID of the Institution

brokerid
BrokerID, for investment accounts

currency
Currency definition ("USD")

file_mtime
File mtime

filename
Filename parsed from

id
Unique ID

ledger_bal
Ledger balance, or investment account value

ledger_bal_as_of
as-of date for the ledger balance

routing_number
Routing Number

type
Account Type, string corresponding to ofxparser.ofxparser.AccountType

biweeklybudget.models.ofx_transaction module

class `biweeklybudget.models.ofx_transaction.`**`OFXTransaction`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

`_sa_class_manager` = <ClassManager of <class 'biweeklybudget.models.ofx_transaction.OFXTransaction'> at 7fad57

`account`
Account this transaction is associated with

`account_amount`
Return the amount of the transaction, appropriately negated if the *`Account`* for this transaction has *`negate_ofx_amounts`* True.

Returns amount, negated as appropriate

Return type decimal.Decimal

account_id
Account ID this transaction is associated with

amount
OFX - Amount

checknum
OFX - Checknum

date_posted
OFX - Date Posted

description
Description

first_statement_by_date
Return the first OFXStatement on or after *self.date_posted*.

Returns first OFXStatement on or after *self.date_posted*

Return type *biweeklybudget.models.ofx_statement.OFXStatement*

fitid
OFX - FITID

is_interest_charge
Account's `re_interest_charge` matched

is_interest_payment
Account's `re_interest_paid` matched

is_late_fee
Account's `re_late_fee` matched

is_other_fee
Account's `re_fee` matched

is_payment
Account's `re_payment` matched

mcc
OFX - MCC

memo
OFX - Memo

name
OFX - Name

notes
Notes

static **params_from_ofxparser_transaction**(*t*, *acct_id*, *stmt*, *cat_memo=False*)
Given an ofxparser.ofxparser.Transaction object, generate and return a dict of kwargs to create a new OFXTransaction.

Parameters

- **t** (`ofxparser.ofxparser.Transaction`) – ofxparser transaction
- **acct_id** (*int*) – OFXAccount ID
- **stmt** (`biweeklybudget.models.ofx_statement.OFXStatement`) – OFXStatement this transaction was on

- **cat_memo** (*bool*) – whether or not to concatenate OFX Memo to Name

Returns dict of kwargs to create an OFXTransaction

Return type dict

reconcile_id
The reconcile_id for the OFX Transaction

sic
OFX - SIC

statement
OFXStatement this transaction was last seen in

statement_id
OFXStatement ID this transaction was last seen in

trans_type
OFX - Transaction Type

static **unreconciled**(*db*)
Return a query to match all unreconciled OFXTransactions.

Parameters **db** (*sqlalchemy.orm.session.Session*) – active database session to use for queries

Returns query to match all unreconciled OFXTransactions

Return type sqlalchemy.orm.query.Query

biweeklybudget.models.projects module

class biweeklybudget.models.projects.**BoMItem**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.projects.BoMItem'> at 7fad57df3ab8>

id
Primary Key

is_active
whether active or historical

line_cost
The total cost for this BoM Item, unit_cost times quantity

Returns total line cost

Return type decimal.Decimal

name
Name of item

notes
Notes / Description

project
Relationship to the *Project* this item is for

project_id
Project ID

quantity
Quantity Required

unit_cost
Unit Cost / Cost Each

url
URL

class biweeklybudget.models.projects.**Project**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.projects.Project'> at 7fad57df3618>

id
Primary Key

is_active
whether active or historical

name
Name of project

notes
Notes / Description

remaining_cost
Return the remaining cost of all line items (*BoMItem*) for this project which are still active

Returns remianing cost of this project

Return type float

total_cost
Return the total cost of all line items (*BoMItem*) for this project.

Returns total cost of this project

Return type float

biweeklybudget.models.reconcile_rule module

class biweeklybudget.models.reconcile_rule.**ReconcileRule**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Base, *biweeklybudget.models.base.ModelAsDict*

_sa_class_manager = <ClassManager of <class 'biweeklybudget.models.reconcile_rule.ReconcileRule'> at 7fad57df3

id
Primary Key

is_active
whether the rule is enabled or disabled

name
Name of the rule

biweeklybudget.models.scheduled_transaction module

class `biweeklybudget.models.scheduled_transaction.`**`ScheduledTransaction`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

`_sa_class_manager` = **<ClassManager of <class 'biweeklybudget.models.scheduled_transaction.ScheduledTransactio**

`account`
Relationship - *`Account`* the transaction is against

`account_id`
ID of the account the transaction is against

`amount`
Amount of the transaction

`budget`
Relationship - *`Budget`* the transaction is against

`budget_id`
ID of the budget the transaction is against

`date`
Denotes a scheduled transaction that will happen once on the given date

`day_of_month`
Denotes a scheduled transaction that happens on the same day of each month

`description`
description

`id`
Primary Key

`is_active`
whether the scheduled transaction is enabled or disabled

`notes`
notes

`num_per_period`
Denotes a scheduled transaction that happens N times per pay period

`recurrence_str`
Return a string describing the recurrence interval. This is a string of the format `YYYY-mm-dd`, `N per period` or `N(st|nd|rd|th)` where `N` is an integer.

Returns string describing recurrence interval

Return type str

`schedule_type`
Return a string describing the type of schedule; one of `date` (a specific Date), `per period` (a number per pay period)'' or `monthly` (a given day of the month).

Returns string describing type of schedule

Return type str

`validate_day_of_month`(*_*, *value*)

`validate_num_per_period`(*_*, *value*)

biweeklybudget.models.transaction module

class `biweeklybudget.models.transaction.`**`Transaction`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

`_sa_class_manager` **= <ClassManager of <class 'biweeklybudget.models.transaction.Transaction'> at 7fad57ef9ab8>**

`account`
Relationship - *`Account`* this transaction is against

`account_id`
ID of the account this transaction is against

`actual_amount`
Actual amount of the transaction

`budget`
Relationship - the *`Budget`* this transaction is against

`budget_id`
ID of the Budget this transaction is against

`budgeted_amount`
Budgeted amount of the transaction

`date`
date of the transaction

`description`
description

`id`
Primary Key

`notes`
free-form notes

`scheduled_trans`
Relationship - the *`ScheduledTransaction`* this Transaction was created from; set when a scheduled transaction is converted to a real one

`scheduled_trans_id`
ID of the ScheduledTransaction this Transaction was created from; set when a scheduled transaction is converted to a real one

`transfer`
Relationship - the *`Transaction`* that makes up the other half/side of a transfer, if this transaction was for a transfer.

`transfer_id`
If the transaction is one half of a transfer, the Transaction ID of the other half/side of the transfer.

static `unreconciled`(*db*)
Return a query to match all unreconciled Transactions.

Parameters **`db`** (*`sqlalchemy.orm.session.Session`*) – active database session to use for queries

Returns query to match all unreconciled Transactions

Return type sqlalchemy.orm.query.Query

biweeklybudget.models.txn_reconcile module

class `biweeklybudget.models.txn_reconcile.`**`TxnReconcile`**(***kwargs*)

Bases: `sqlalchemy.ext.declarative.api.Base`, *`biweeklybudget.models.base.ModelAsDict`*

`_sa_class_manager` = **<ClassManager of <class 'biweeklybudget.models.txn_reconcile.TxnReconcile'> at 7fad57db4b**

`id`
Primary Key

`note`
Notes

`ofx_account_id`
OFX Transaction Account ID

`ofx_fitid`
OFX Transaction FITID

`ofx_trans`
Relationship - *`OFXTransaction`*

`reconciled_at`
time when this reconcile was made

`rule`
Relationship - *`ReconcileRule`* that created this reconcile, if any.

`rule_id`
ReconcileRule ID; set if this reconcile was created by a rule

`transaction`
Relationship - *`Transaction`*

`txn_id`
Transaction ID

biweeklybudget.models.utils module

`biweeklybudget.models.utils.`**`do_budget_transfer`**(*db_sess*, *txn_date*, *amount*, *account*, *from_budget*, *to_budget*, *notes=None*)

Transfer a given amount from `from_budget` to `to_budget` on `txn_date`. This method does NOT commit database changes. There are places where we rely on this function not committing changes.

Parameters

- **`db_sess`** (*`sqlalchemy.orm.session.Session`*) – active database session to use for queries
- **`txn_date`** (*`datetime.date`*) – date to make the transfer Transactions on
- **`amount`** (*`float`*) – amount of money to transfer
- **`account`** (`biweeklybudget.models.account.Account`) –
- **`from_budget`** (`biweeklybudget.models.budget_model.Budget`) –
- **`to_budget`** (`biweeklybudget.models.budget_model.Budget`) –
- **`notes`** (*`str`*) – Notes to add to the Transaction

Returns list of Transactions created for the transfer

Return type `list` of *`Transaction`* objects

biweeklybudget.ofxapi package

`biweeklybudget.ofxapi.`**`apiclient`**(*api_url=None*, *ca_bundle=None*, *client_cert=None*, *client_key=None*)

Submodules

biweeklybudget.ofxapi.exceptions module

exception `biweeklybudget.ofxapi.exceptions.`**`DuplicateFileException`**(*acct_id*, *filename*, *stmt_id*)

Bases: `exceptions.Exception`

Exception raised when trying to parse a file that has already been parsed for the Account (going by the OFX signon date).

biweeklybudget.ofxapi.local module

class `biweeklybudget.ofxapi.local.`**`OfxApiLocal`**(*db_sess*)

Bases: `object`

`_create_statement`(*acct*, *ofx*, *mtime*, *filename*)

Create an OFXStatement for this OFX file. If one already exists with the same account and filename, raise DuplicateFileException.

Parameters

- **`acct`** (`biweeklybudget.models.account.Account`) – the Account this statement is for
- **`ofx`** (`ofxparse.ofxparse.Ofx`) – Ofx instance for parsed file
- **`mtime`** (*`datetime.datetime`*) – OFX file modification time (or current time)
- **`filename`** (*`str`*) – OFX file name

Returns the OFXStatement object

Return type *biweeklybudget.models.ofx_statement.OFXStatement*

Raises DuplicateFileException

`_new_updated_counts`()

Return integer counts of the number of *`OFXTransaction`* objects that have been created and updated.

Returns 2-tuple of new OFXTransactions created, OFXTransactions updated

Return type tuple

`_update_bank_or_credit`(*acct*, *ofx*, *stmt*)

Update a single OFX file for this Bank or Credit account.

Parameters

- **`acct`** (`biweeklybudget.models.account.Account`) – the Account this statement is for

- **ofx** (`ofxparse.ofxparse.Ofx`) – Ofx instance for parsed file
- **stmt** (`biweeklybudget.models.ofx_statement.OFXStatement`) – the OFXStatement for this statement

Returns the OFXStatement object

Return type *biweeklybudget.models.ofx_statement.OFXStatement*

_update_investment(*acct*, *ofx*, *stmt*)

Update a single OFX file for this Investment account.

Parameters

- **acct** (`biweeklybudget.models.account.Account`) – the Account this statement is for
- **ofx** (`ofxparse.ofxparse.Ofx`) – Ofx instance for parsed file
- **stmt** (`biweeklybudget.models.ofx_statement.OFXStatement`) – the OFXStatement for this statement

Returns the OFXStatement object

Return type *biweeklybudget.models.ofx_statement.OFXStatement*

get_accounts()

Query the database for all *ofxgetter-enabled Accounts* that have a non-empty *biweeklybudget.models.account.Account.ofxgetter_config* and a non-None *biweeklybudget.models.account.Account.vault_creds_path*. Return a dict of string *Account name* to dict with keys:

- `vault_path` - *vault_creds_path*
- `config` - *ofxgetter_config*
- `id` - *id*
- `cat_memo` - *ofx_cat_memo_to_name*

Returns dict of account names to configuration

Return type dict

update_statement_ofx(*acct_id*, *ofx*, *mtime=None*, *filename=None*)

Update a single statement for the specified account, from an OFX file.

Parameters

- **acct_id** (*int*) – Account ID that statement is for
- **ofx** (`ofxparse.ofxparse.Ofx`) – Ofx instance for parsed file
- **mtime** (*datetime.datetime*) – OFX file modification time (or current time)
- **filename** (*str*) – OFX file name

Returns 3-tuple of the int ID of the *OFXStatement* created by this run, int count of new *OFXTransaction* created, and int count of *OFXTransaction* updated

Return type tuple

Raises `RuntimeError` on error parsing OFX or unknown account type; *DuplicateFileException* if the file (according to the OFX signon date/time) has already been recorded.

biweeklybudget.ofxapi.remote module

class `biweeklybudget.ofxapi.remote.`**`OfxApiRemote`**(*api_base_url*, *ca_bundle=None*, *client_cert_path=None*, *client_key_path=None*)

Bases: `object`

Remote OFX API client, used by ofxgetter/ofxbackfiller when running on a remote system.

`get_accounts`()

Query the database for all *ofxgetter-enabled Accounts* that have a non-empty *biweeklybudget.models.account.Account.ofxgetter_config* and a non-None *biweeklybudget.models.account.Account.vault_creds_path*. Return a dict of string *Account name* to dict with keys:

- `vault_path` - *vault_creds_path*
- `config` - *ofxgetter_config*
- `id` - *id*
- `cat_memo` - *ofx_cat_memo_to_name*

Returns dict of account names to configuration

Return type dict

`update_statement_ofx`(*acct_id*, *ofx*, *mtime=None*, *filename=None*)

Update a single statement for the specified account, from an OFX file.

Parameters

- **`acct_id`** (*int*) – Account ID that statement is for
- **`ofx`** (`ofxparse.ofxparse.Ofx`) – Ofx instance for parsed file
- **`mtime`** (*datetime.datetime*) – OFX file modification time (or current time)
- **`filename`** (*str*) – OFX file name

Returns 3-tuple of the int ID of the *OFXStatement* created by this run, int count of new *OFXTransaction* created, and int count of *OFXTransaction* updated

Return type tuple

Raises `RuntimeError` on error parsing OFX or unknown account type; *DuplicateFileException* if the file (according to the OFX signon date/time) has already been recorded.

Submodules

biweeklybudget.backfill_ofx module

class `biweeklybudget.backfill_ofx.`**`OfxBackfiller`**(*client*, *savedir*)

Bases: `object`

Class to backfill OFX in database from files on disk.

`_do_account_dir`(*acct_id*, *path*)

Handle all OFX statements in a per-account directory.

Parameters

- **acct_id** (*int*) – account database ID
- **path** (*str*) – absolute path to per-account directory

_do_one_file(*acct_id*, *path*)
Parse one OFX file and use OFXUpdater to upsert it into the DB.

Parameters

- **acct_id** (*int*) – Account ID number
- **path** (*str*) – absolute path to OFX/QFX file

run()
Main entry point - run the backfill.

biweeklybudget.backfill_ofx.**main**()
Main entry point - instantiate and run *OfxBackfiller*.

biweeklybudget.backfill_ofx.**parse_args**()
Parse command-line arguments.

biweeklybudget.biweeklypayperiod module

class biweeklybudget.biweeklypayperiod.**BiweeklyPayPeriod**(*start_date*, *db_session*)
Bases: object

This object contains all logic related to working with pay periods, specifically finding a pay period for a given data, and figuring out the start and end dates of pay periods. Sure, the app is called "biweeklybudget" but there's no reason to hard-code logic all over the place that's this simple.

_data
Return the object-local data cache dict. Built it if not already present.

Returns object-local data cache

Return type dict

_dict_for_sched_trans(*t*)
Return a dict describing the ScheduledTransaction t. Called from *_trans_dict()*.

The resulting dict will have the following layout:

- type (**str**) "Transaction" or "ScheduledTransaction"
- id (**int**) the id of the object
- date (**date**) the date of the transaction, or None for per-period ScheduledTransactions
- sched_type (**str**) for ScheduledTransactions, the schedule type ("monthly", "date", or "per period")
- sched_trans_id **None**
- description (**str**) the transaction description
- amount (**Decimal.decimal**) the transaction amount
- budgeted_amount **None**
- account_id (**int**) the id of the Account the transaction is against.
- account_name (**str**) the name of the Account the transaction is against.
- budget_id (**int**) the id of the Budget the transaction is against.

- `budget_name` (**str**) the name of the Budget the transaction is against.
- `reconcile_id` (**int**) the ID of the TxnReconcile, or None

Parameters **t** (`ScheduledTransaction`) – ScheduledTransaction to describe

Returns common-format dict describing `t`

Return type dict

_dict_for_trans(*t*)

Return a dict describing the Transaction t. Called from *_trans_dict()*.

The resulting dict will have the following layout:

- `type` (**str**) "Transaction" or "ScheduledTransaction"
- `id` (**int**) the id of the object
- `date` (**date**) the date of the transaction, or None for per-period ScheduledTransactions
- `sched_type` (**str**) for ScheduledTransactions, the schedule type ("monthly", "date", or "per period")
- `sched_trans_id` (**int**) for Transactions, the ScheduledTransaction `id` that it was created from, or None.
- `description` (**str**) the transaction description
- `amount` (**Decimal.decimal**) the transaction amount
- `budgeted_amount` (**Decimal.decimal**) the budgeted amount. This may be None.
- `account_id` (**int**) the id of the Account the transaction is against.
- `account_name` (**str**) the name of the Account the transaction is against.
- `budget_id` (**int**) the id of the Budget the transaction is against.
- `budget_name` (**str**) the name of the Budget the transaction is against.
- `reconcile_id` (**int**) the ID of the TxnReconcile, or None

Parameters **t** (`Transaction`) – transaction to describe

Returns common-format dict describing `t`

Return type dict

_income_budget_ids

Return a list of all *Budget* IDs for Income budgets.

Returns list of income budget IDs

Return type list

_make_budget_sums()

Find the sums of all transactions per periodic budget ID ; return a dict where keys are budget IDs and values are per-budget dicts containing:

- `budget_amount` *(Decimal.decimal)* - the periodic budget *starting_balance*.
- `allocated` *(Decimal.decimal)* - sum of all *ScheduledTransaction* and *Transaction* amounts against the budget this period. For actual transactions, we use the *budgeted_amount* if present (not None).

- `spent` *(Decimal.decimal)* - the sum of all actual *`Transaction`* amounts against the budget this period.
- `trans_total` *(Decimal.decimal)* - the sum of spent amounts for Transactions that have them, or allocated amounts for ScheduledTransactions.
- `remaining` *(Decimal.decimal)* - the remaining amount in the budget. This is `budget_amount` minus the greater of `allocated` or `trans_total`. For income budgets, this is always positive.

Returns dict of dicts, transaction sums and amounts per budget

Return type dict

`_make_combined_transactions`()

Combine all Transactions and ScheduledTransactions from `self._data_cache` into one ordered list of similar dicts, adding dates to the monthly ScheduledTransactions as appropriate and excluding ScheduledTransactions that have been converted to real Transactions. Store the finished list back into `self._data_cache`.

`_make_overall_sums`()

Return a dict describing the overall sums for this pay period, namely:

- `allocated` *(Decimal.decimal)* total amount allocated via *`ScheduledTransaction`*, *`Transaction`* (counting the *`budgeted_amount`* for Transactions that have one), or *`Budget`* (not counting income budgets).
- `spent` *(Decimal.decimal)* total amount actually spent via *`Transaction`*.
- `income` *(Decimal.decimal)* total amount of income allocated this pay period. Calculated value (from *`_make_budget_sums()`* / `self._data_cache['budget_sums']`) should be negative, but is returned as its positive inverse (absolute value).
- `remaining` *(Decimal.decimal)* income minus the greater of `allocated` or `spent` for current or future pay periods, or minus `spent` for pay periods ending in the past (*`is_in_past`*)

Returns dict describing sums for the pay period

Return type dict

`_scheduled_transactions_date`()

Return a Query for all *`ScheduledTransaction`* defined by date (schedule_type == "date") for this pay period.

Returns Query matching all ScheduledTransactions defined by date, for this pay period.

Return type sqlalchemy.orm.query.Query

`_scheduled_transactions_monthly`()

Return a Query for all *`ScheduledTransaction`* defined by day of month (schedule_type == "monthly") for this pay period.

Returns Query matching all ScheduledTransactions defined by day of month (monthly) for this period.

Return type sqlalchemy.orm.query.Query

`_scheduled_transactions_per_period`()

Return a Query for all *`ScheduledTransaction`* defined by number per period (schedule_type == "per period") for this pay period.

Returns Query matching all ScheduledTransactions defined by number per period, for this pay period.

Return type sqlalchemy.orm.query.Query

_trans_dict(*t*)

Given a Transaction or ScheduledTransaction, return a dict of a common format describing the object.

The resulting dict will have the following layout:

- `type` (**str**) "Transaction" or "ScheduledTransaction"
- `id` (**int**) the id of the object
- `date` (**date**) the date of the transaction, or None for per-period ScheduledTransactions
- `sched_type` (**str**) for ScheduledTransactions, the schedule type ("monthly", "date", or "per period")
- `sched_trans_id` (**int**) for Transactions, the ScheduledTransaction `id` that it was created from, or None.
- `description` (**str**) the transaction description
- `amount` (**Decimal.decimal**) the transaction amount
- `budgeted_amount` (**Decimal.decimal**) the budgeted amount. This may be None.
- `account_id` (**int**) the id of the Account the transaction is against.
- `account_name` (**str**) the name of the Account the transaction is against.
- `budget_id` (**int**) the id of the Budget the transaction is against.
- `budget_name` (**str**) the name of the Budget the transaction is against.
- `reconcile_id` (**int**) the ID of the TxnReconcile, or None

Parameters **t** (*Transaction* or *ScheduledTransaction*) – the object to return a dict for

Returns dict describing `t`

Return type dict

_transactions()

Return a Query for all *Transaction* for this pay period.

Returns Query matching all Transactions for this pay period

Return type sqlalchemy.orm.query.Query

budget_sums

Return a dict of budget sums; the return value of *_make_budget_sums()*.

Returns dict of dicts, transaction sums and amounts per budget

Return type dict

clear_cache()

Clear the cached transaction, budget and sum data stored in *self._data_cache* and returned by *_data*.

end_date

Return the date of the last day in this pay period. The pay period is generally considered to end at the last instant (i.e. 23:59:59) of this date.

Returns last date in the pay period

Return type datetime.date

filter_query (*query*, *date_prop*)

Filter `query` for `date_prop` in this pay period. Returns a copy of the query.

e.g. to filter an existing query of *`OFXTransaction`* for the BiweeklyPayPeriod starting on 2017-01-14:

```
q = # some query here
p = BiweeklyPayPeriod(date(2017, 1, 14))
q = p.filter_query(q, OFXTransaction.date_posted)
```

Parameters

- **query** (`sqlalchemy.orm.query.Query`) – The query to filter
- **date_prop** – the Model's date property, to filter on.

Returns the filtered query

Return type `sqlalchemy.orm.query.Query`

is_in_past

next

Return the BiweeklyPayPeriod following this one.

Returns next BiweeklyPayPeriod after this one

Return type *BiweeklyPayPeriod*

overall_sums

Return a dict of overall sums; the return value of *`_make_overall_sums()`*.

Returns dict describing sums for the pay period

Return type dict

static **period_for_date** (*dt*, *db_session*)

Given a datetime, return the BiweeklyPayPeriod instance describing the pay period containing this date.

Todo

This is a very naive, poorly-performing implementation.

Parameters

- **dt** (`datetime` or `date`) – datetime or date to find the pay period for
- **db_session** (*`sqlalchemy.orm.session.Session`*) – active database session to use for queries

Returns BiweeklyPayPeriod containing the specified date

Return type *`BiweeklyPayPeriod`*

period_interval

Return the interval between BiweeklyPayPeriods as a timedelta.

Returns interval between BiweeklyPayPeriods

Return type datetime.timedelta

period_length

Return the length of a BiweeklyPayPeriod; this is calculated as *`period_interval`* minus one second.

Returns length of one BiweeklyPayPeriod

Return type datetime.timedelta

previous

Return the BiweeklyPayPeriod preceding this one.

Returns previous BiweeklyPayPeriod before this one

Return type *BiweeklyPayPeriod*

start_date

Return the starting date for this pay period. The period is generally considered to start at midnight (00:00) of this date.

Returns start date for pay period

Return type datetime.date

transactions_list

Return an ordered list of dicts, each representing a transaction for this pay period. Dicts have keys and values as described in `_trans_dict()`.

Returns ordered list of transaction dicts

Return type list

biweeklybudget.cliutils module

biweeklybudget.cliutils.**set_log_debug**(*logger*)

set logger level to DEBUG, and debug-level output format, via `set_log_level_format()`.

biweeklybudget.cliutils.**set_log_info**(*logger*)

set logger level to INFO via `set_log_level_format()`.

biweeklybudget.cliutils.**set_log_level_format**(*logger*, *level*, *format*)

Set logger level and format.

Parameters

- **logger** (*logging.Logger*) – the logger object to set on
- **level** (*int*) – logging level; see the `logging` constants.
- **format** (*str*) – logging formatter format string

biweeklybudget.db module

biweeklybudget.db.**_alembic_get_current_rev**(*config*, *script*)

Works sorta like alembic.command.current

Parameters **config** – alembic Config

Returns current revision

Return type str

biweeklybudget.db.**cleanup_db**()

This must be called from all scripts, using

atexit.register(cleanup_db)

biweeklybudget.db.**db_session** = **<sqlalchemy.orm.scoping.scoped_session object>**
sqlalchemy.orm.scoping.scoped_session session

biweeklybudget.db.**engine** = **Engine(sqlite:///:memory:)**
The database engine object; return value of sqlalchemy.create_engine().

biweeklybudget.db.**init_db**()
Initialize the database; call sqlalchemy.schema.MetaData.create_all() on the metadata object.

biweeklybudget.db.**upsert_record**(*model_class*, *key_fields*, ***kwargs*)
Upsert a record in the database.

key_fields is either a string primary key field name (a key in the kwargs dict) or a list or tuple of string primary key field names, for compound keys.

If a record can be found matching these keys, it will be updated and committed. If not, a new one will be inserted. Either way, the record is returned.

sqlalchemy.orm.session.Session.commit() is **NOT** called.

Parameters

- **model_class** (biweeklybudget.models.base.ModelAsDict) – the class of model to insert/update
- **key_fields** – The field name(s) (keys in kwargs) that make up the primary key. This can be a single string, or a list or tuple of strings for compound keys. The values for these key fields MUST be included in kwargs.
- **kwargs** (*dict*) – arguments to provide to the model class constructor, or to update if there is an existing record matching the key.

Returns inserted or updated record; type is an instance of model_class

biweeklybudget.db_event_handlers module

biweeklybudget.db_event_handlers.**handle_before_flush**(*session*, *flush_context*, *instances*)
Hook into before_flush (sqlalchemy.orm.events.SessionEvents.before_flush()) on the DB session, to handle updates that need to be made before persisting data. Currently, this method just calls a number of other methods to handle specific cases:

- *handle_new_transaction()*

Parameters

- **session** (*sqlalchemy.orm.session.Session*) – current database session
- **flush_context** (*sqlalchemy.orm.session.UOWTransaction*) – internal SQLAlchemy object
- **instances** – deprecated

biweeklybudget.db_event_handlers.**handle_new_transaction**(*session*)
before_flush event handler (sqlalchemy.orm.events.SessionEvents.before_flush()) on the DB session, to handle creation of *new* Transactions. For updates to existing Transactions, we rely on *handle_trans_amount_change()*.

If the Transaction's *budget* is a *Budget* with *is_periodic* False (i.e. a standing budget), update the Budget's *current_balance* for this transaction.

Parameters **session** (*sqlalchemy.orm.session.Session*) – current database session

biweeklybudget.db_event_handlers.**handle_trans_amount_change**(***kwargs*)

Handle change of `Transaction.actual_amount` for existing instances (`id` is not None). For new instances, we rely on `handle_new_transaction()` called via `handle_before_flush()`.

If the Transaction's `budget` is a `Budget` with `is_periodic` `False` (i.e. a standing budget), update the Budget's `current_balance` for this transaction.

See: `sqlalchemy.orm.events.AttributeEvents.set()`

Parameters **kwargs** (*dict*) – keyword arguments

biweeklybudget.db_event_handlers.**init_event_listeners**(*db_session*)

Initialize/register all SQLAlchemy event listeners.

See http://docs.sqlalchemy.org/en/latest/orm/events.html

Parameters **db_session** (*sqlalchemy.orm.session.Session*) – the Database Session

biweeklybudget.initdb module

biweeklybudget.initdb.**main**()

biweeklybudget.initdb.**parse_args**()

biweeklybudget.interest module

class biweeklybudget.interest.**AdbCompoundedDaily**(*apr*)

Bases: *biweeklybudget.interest._InterestCalculation*

Average Daily Balance method, compounded daily (like American Express).

calculate(*principal*, *first_d*, *last_d*, *transactions={}*)

Calculate compound interest for the specified principal.

Parameters

- **principal** (*decimal.Decimal*) – balance at beginning of statement period
- **first_d** (*datetime.date*) – date of beginning of statement period
- **last_d** (*datetime.date*) – last date of statement period
- **transactions** (*dict*) – dict of datetime.date to float amount adjust the balance by on the specified dates.

Returns dict describing the result: end_balance (float), interest_paid (float)

Return type dict

description = 'Average Daily Balance Compounded Daily (AmEx)'

Human-readable string name of the interest calculation type.

class biweeklybudget.interest.**CCStatement**(*interest_cls*, *principal*, *min_payment_cls*, *billing_period*, *transactions={}*, *end_balance=None*, *interest_amt=None*)

Bases: `object`

Represent a credit card statement (one billing period).

apr

billing_period

Return the Billing Period for this statement.

Returns billing period for this statement

Return type *_BillingPeriod*

end_date

interest

minimum_payment

Return the minimum payment for the next billing cycle.

Returns minimum payment for the next billing cycle

Return type decimal.Decimal

next_with_transactions(*transactions={}*)

Return a new CCStatement reflecting the next billing period, with a payment of *amount* applied to it.

Parameters **transactions** (*dict*) – dict of transactions, *datetime.date* to *Decimal*

Returns next period statement, with transactions applied

Return type *CCStatement*

pay(*amount*)

Return a new CCStatement reflecting the next billing period, with a payment of *amount* applied to it at the middle of the period.

Parameters **amount** (*decimal.Decimal*) – amount to pay during the next statement period

Returns next period statement, with payment applied

Return type *CCStatement*

principal

start_date

class biweeklybudget.interest.**FixedPaymentMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)

Bases: *biweeklybudget.interest._PayoffMethod*

TESTING ONLY - pay the same amount on every statement.

description = 'TESTING ONLY - Fixed Payment for All Statements'

find_payments(*statements*)

Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = False

class biweeklybudget.interest.**HighestBalanceFirstMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)

Bases: *biweeklybudget.interest._PayoffMethod*

Pay statements off from highest to lowest balance.

description = 'Highest to Lowest Balance'

find_payments(*statements*)

Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = True

class `biweeklybudget.interest.`**`HighestInterestRateFirstMethod`**(*max_total_payment=None, increases={}, onetimes={}*)

Bases: *`biweeklybudget.interest._PayoffMethod`*

Pay statements off from highest to lowest interest rate.

description = 'Highest to Lowest Interest Rate'

find_payments(*statements*)

Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*`list`*) – statements to pay, list of *`CCStatement`*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = True

`biweeklybudget.interest.`**INTEREST_CALCULATION_NAMES = {'AdbCompoundedDaily': {'doc': 'Average Daily Bala**

Dict mapping interest calculation class names to their description and docstring.

class `biweeklybudget.interest.`**`InterestHelper`**(*db_sess, increases={}, onetimes={}*)

Bases: `object`

_calc_payoff_method(*cls*)

Calculate payoffs using one method.

Parameters **cls** (`biweeklybudget.interest._PayoffMethod`) – payoff method class

Returns Dict with integer *account_id* as the key, and values are dicts with keys "payoff_months" (int), "total_payments" (Decimal) and "total_interest" (Decimal).

Return type dict

_get_credit_accounts()

Return a dict of *account_id* to *`Account`* for all Credit type accounts with OFX data present.

Returns dict of account_id to Account instance

Return type dict

_make_statements(*accounts*)

Make *`CCStatement`* instances for each account; return a dict of *account_id* to CCStatement instance.

Parameters **accounts** (*`dict`*) – dict of (int) account_id to Account instance

Returns dict of (int) account_id to CCStatement instance

Return type dict

accounts

Return a dict of *account_id* to *`Account`* for all Credit type accounts with OFX data present.

Returns dict of account_id to Account instance

Return type dict

calculate_payoffs()

Calculate payoffs for each account/statement.

Returns dict of payoff information. Keys are payoff method names. Values are dicts, with keys "description" (str description of the payoff method), "doc" (the docstring of the class), and "results". The "results" dict has integer *account_id* as the key, and values are dicts with keys "payoff_months" (int), "total_payments" (Decimal) and "total_interest" (Decimal).

Return type dict

min_payments

Return a dict of *account_id* to minimum payment for the latest statement, for each account.

Returns dict of *account_id* to minimum payment (Decimal)

Return type dict

class biweeklybudget.interest.**LowestBalanceFirstMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)

Bases: *biweeklybudget.interest._PayoffMethod*

Pay statements off from lowest to highest balance, a.k.a. the "snowball" method.

description = 'Lowest to Highest Balance (a.k.a. Snowball Method)'

find_payments(*statements*)

Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = True

class biweeklybudget.interest.**LowestInterestRateFirstMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)

Bases: *biweeklybudget.interest._PayoffMethod*

Pay statements off from lowest to highest interest rate.

description = 'Lowest to Highest Interest Rate'

find_payments(*statements*)

Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = True

biweeklybudget.interest.**MIN_PAYMENT_FORMULA_NAMES = {'MinPaymentCiti': {'doc': "Greater of:\n - $25;\n - Th**

Dict mapping Minimum Payment Formula class names to their description and docstring.

class biweeklybudget.interest.**MinPaymentAmEx**

Bases: *biweeklybudget.interest._MinPaymentFormula*

Interest on last statement plus 1% of balance, or $35 if balance is less than $35.

calculate(*balance*, *interest*)

Calculate the minimum payment for a statement with the given balance and interest amount.

Parameters

- **balance** (*decimal.Decimal*) – balance amount for the statement

- **interest** (*decimal.Decimal*) – interest charged for the statement period

Returns minimum payment for the statement

Return type decimal.Decimal

description = 'AmEx - Greatest of Interest Plus 1% of Principal, or $35'
human-readable string description of the formula

class biweeklybudget.interest.**MinPaymentCiti**
Bases: *biweeklybudget.interest._MinPaymentFormula*

Greater of: - $25; - The new balance, if it's less than $25; - 1 percent of the new balance, plus the current statement's interest charges or minimum interest charges, plus late fees; - 1.5% of the new balance, rounded to the nearest dollar amount.

In all cases, add past fees and finance charges due, plus any amount in excess of credit line.

calculate(*balance*, *interest*)
Calculate the minimum payment for a statement with the given balance and interest amount.

Parameters

- **balance** (*decimal.Decimal*) – balance amount for the statement
- **interest** (*decimal.Decimal*) – interest charged for the statement period

Returns minimum payment for the statement

Return type decimal.Decimal

description = 'Citi - Greatest of 1.5% of Principal, or 1% of Principal plus interest and fees, or $25, or Principal'
human-readable string description of the formula

class biweeklybudget.interest.**MinPaymentDiscover**
Bases: *biweeklybudget.interest._MinPaymentFormula*

Greater of: - $35; or - 2% of the New Balance shown on your billing statement; or - $20, plus any of the following charges as shown on your billing statement: fees for any debt protection product that you enrolled in on or after 2/1/2015; Interest Charges; and Late Fees.

calculate(*balance*, *interest*)
Calculate the minimum payment for a statement with the given balance and interest amount.

Parameters

- **balance** (*decimal.Decimal*) – balance amount for the statement
- **interest** (*decimal.Decimal*) – interest charged for the statement period

Returns minimum payment for the statement

Return type decimal.Decimal

description = 'Discover - Greatest of 2% of Principal, or $20 plus Interest, or $35'
human-readable string description of the formula

class biweeklybudget.interest.**MinPaymentMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)
Bases: *biweeklybudget.interest._PayoffMethod*

Pay only the minimum on each statement.

description = 'Minimum Payment Only'

find_payments(*statements*)
Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

show_in_ui = True

`biweeklybudget.interest.`**PAYOFF_METHOD_NAMES = {'HighestInterestRateFirstMethod': {'doc': 'Pay statements off**

Dict mapping Payoff Method class names to their description and docstring.

class `biweeklybudget.interest.`**SimpleInterest**(*apr*)

Bases: *biweeklybudget.interest._InterestCalculation*

Simple interest, charged on balance at the end of the billing period.

calculate(*principal*, *first_d*, *last_d*, *transactions={}*)

Calculate compound interest for the specified principal.

Parameters

- **principal** (*decimal.Decimal*) – balance at beginning of statement period
- **first_d** (*datetime.date*) – date of beginning of statement period
- **last_d** (*datetime.date*) – last date of statement period
- **transactions** (*dict*) – dict of datetime.date to float amount adjust the balance by on the specified dates.

Returns dict describing the result: end_balance (float), interest_paid (float)

Return type dict

description = 'Interest charged once on the balance at end of period.'

Human-readable string name of the interest calculation type.

class `biweeklybudget.interest.`**_BillingPeriod**(*end_date*, *start_date=None*)

Bases: `object`

description = None

human-readable string description of the billing period type

end_date

next_period

Return the next billing period after this one.

Returns next billing period

Return type *_BillingPeriod*

payment_date

prev_period

Return the previous billing period before this one.

Returns previous billing period

Return type *_BillingPeriod*

start_date

class `biweeklybudget.interest.`**_InterestCalculation**(*apr*)

Bases: `object`

apr

calculate(*principal*, *first_d*, *last_d*, *transactions={}*)
Calculate compound interest for the specified principal.

Parameters

- **principal** (*decimal.Decimal*) – balance at beginning of statement period
- **first_d** (*datetime.date*) – date of beginning of statement period
- **last_d** (*datetime.date*) – last date of statement period
- **transactions** (*dict*) – dict of datetime.date to float amount adjust the balance by on the specified dates.

Returns dict describing the result: end_balance (float), interest_paid (float)

Return type dict

description = None
Human-readable string name of the interest calculation type.

class biweeklybudget.interest.**_MinPaymentFormula**
Bases: object

calculate(*balance*, *interest*)
Calculate the minimum payment for a statement with the given balance and interest amount.

Parameters

- **balance** (*decimal.Decimal*) – balance amount for the statement
- **interest** (*decimal.Decimal*) – interest charged for the statement period

Returns minimum payment for the statement

Return type decimal.Decimal

description = None
human-readable string description of the formula

class biweeklybudget.interest.**_PayoffMethod**(*max_total_payment=None*, *increases={}*, *onetimes={}*)
Bases: object

A payoff method for multiple cards; a method of figuring out how much to pay on each card, each month.

description = None
human-readable string name of the payoff method

find_payments(*statements*)
Given a list of statements, return a list of payment amounts to make on each of the statements.

Parameters **statements** (*list*) – statements to pay, list of *CCStatement*

Returns list of payment amounts to make, same order as `statements`

Return type list

max_total_for_period(*period*)
Given a *_BillingPeriod*, calculate the maximum total payment for that period, including both *self._max_total* and the increases and onetimes specified on the class constructor.

Parameters **period** (*_BillingPeriod*) – billing period to get maximum total payment for

Returns maximum total payment for the period

Return type decimal.Decimal

`biweeklybudget.interest.`**`calculate_payoffs`**(*payment_method*, *statements*)

Calculate the amount of time (in years) and total amount of money required to pay off the cards associated with the given list of statements. Return a list of (*float* number of years, *decimal.Decimal* amount paid) tuples for each item in *statements*.

Parameters

- **payment_method** (`_PayoffMethod`) – method used for calculating payment amount to make on each statement; subclass of _PayoffMethod
- **statements** (`list`) – list of `CCStatement` objects to pay off.

Returns list of (*float* number of billing periods, *decimal.Decimal* amount paid) tuples for each item in *statements*

Return type list

`biweeklybudget.interest.`**`subclass_dict`**(*klass*)

biweeklybudget.load_data module

`biweeklybudget.load_data.`**`main`**()

`biweeklybudget.load_data.`**`parse_args`**()

biweeklybudget.ofxgetter module

class `biweeklybudget.ofxgetter.`**`OfxGetter`**(*client*, *savedir='./'*)

Bases: `object`

`_get_ofx_scraper`(*account_name*, *days=30*)

Get OFX via a ScreenScraper subclass.

Parameters

- **account_name** (`str`) – account name
- **days** (`int`) – number of days of data to download

Returns OFX string

Return type str

`_ofx_to_db`(*account_name*, *fname*, *ofxdata*)

Put OFX Data to the DB

Parameters

- **account_name** (`str`) – account name to download
- **ofxdata** (`str`) – raw OFX data
- **fname** (`str`) – filename OFX was written to

`_write_ofx_file`(*account_name*, *ofxdata*)

Write OFX data to a file.

Parameters

- **account_name** (`str`) – account name
- **ofxdata** (`str`) – raw OFX data string

Returns name of the file that was written

Return type str

static accounts(*client*)

Return a dict of account information of ofxgetter-enabled accounts, str account name to dict of information about the account.

Parameters **client** (Instance of `OfxApiLocal` or `OfxApiRemote`) – API client

Returns dict of account information; see `get_accounts()` for details.

Return type dict

get_ofx(*account_name*, *write_to_file=True*, *days=30*)

Download OFX from the specified account. Return it as a string.

Parameters

- **account_name** (`str`) – account name to download
- **write_to_file** (`bool`) – if True, also write to a file named "<account_name>_<date stamp>.ofx"
- **days** (`int`) – number of days of data to download

Returns OFX string

Return type str

biweeklybudget.ofxgetter.**main**()

biweeklybudget.ofxgetter.**parse_args**()

biweeklybudget.prime_rate module

class biweeklybudget.prime_rate.**PrimeRateCalculator**(*db_session*)

Bases: `object`

_get_prime_rate()

Get the US Prime Rate from MarketWatch; update the DB and return the value.

Returns current US Prime Rate

Return type decimal.Decimal

_rate_from_marketwatch()

calculate_apr(*margin*)

Calculate an APR based on the prime rate.

Parameters **margin** (`decimal.Decimal`) – margin added to Prime Rate to get APR

Returns effective APR

Return type decimal.Decimal

prime_rate

Return the current US Prime Rate

Returns current US Prime Rate

Return type decimal.Decimal

biweeklybudget.screenscraper module

class `biweeklybudget.screenscraper.`**`ScreenScraper`**(*savedir='./'*, *screenshot=False*)

Bases: `object`

Base class for screen-scraping bank/financial websites.

`do_screenshot`()

take a debug screenshot

`doc_readystate_is_complete`(*foo*)

return true if document is ready/complete, false otherwise

`error_screenshot`(*fname=None*)

`get_browser`(*browser_name*)

get a webdriver browser instance

`jquery_finished`(*foo*)

return true if jQuery.active == 0 else false

`load_cookies`(*cookie_file*)

Load cookies from a JSON cookie file on disk. This file is not the format used natively by PhantomJS, but rather the JSON-serialized representation of the dict returned by `selenium.webdriver.remote.webdriver.WebDriver.get_cookies()`.

Cookies are loaded via `selenium.webdriver.remote.webdriver.WebDriver.add_cookie()`

Parameters **`cookie_file`** (*`str`*) – path to the cookie file on disk

`save_cookies`(*cookie_file*)

Save cookies to a JSON cookie file on disk. This file is not the format used natively by PhantomJS, but rather the JSON-serialized representation of the dict returned by `selenium.webdriver.remote.webdriver.WebDriver.get_cookies()`.

Parameters **`cookie_file`** (*`str`*) – path to the cookie file on disk

`wait_for_ajax_load`(*timeout=20*)

Function to wait for an ajax event to finish and trigger page load, like the Janrain login form.

Pieced together from http://stackoverflow.com/a/15791319

timeout is in seconds

`xhr_get_url`(*url*)

use JS to download a given URL, return its contents

`xhr_post_urlencoded`(*url*, *data*, *headers={}*)

use JS to download a given URL, return its contents

biweeklybudget.settings module

`biweeklybudget.settings.`**`BIWEEKLYBUDGET_TEST_TIMESTAMP`** **= None**

int - FOR ACCEPTANCE TESTS ONLY - This is used to "fudge" the current time to the specified integer timestamp. Used for acceptance tests only. Do NOT set this outside of acceptance testing.

`biweeklybudget.settings.`**`DB_CONNSTRING`** **= 'sqlite:///:memory:'**

string - SQLAlchemy database connection string. See the SQLAlchemy Database URLS docs for further information.

`biweeklybudget.settings.`**DEFAULT_ACCOUNT_ID = 1**
int - Account ID to show first in dropdown lists. This must be the database ID of a valid account.

`biweeklybudget.settings.`**FUEL_BUDGET_ID = 1**
int - Budget ID to select as default when inputting Fuel Log entries. This must be the database ID of a valid budget.

`biweeklybudget.settings.`**PAY_PERIOD_START_DATE = datetime.date(2017, 3, 17)**
`datetime.date` - The starting date of one pay period (generally the first pay period represented in data in this app). The dates of all pay periods will be determined based on an interval from this date. This must be specified in Y-m-d format (i.e. parsable by `datetime.datetime.strptime()` with `%Y-%m-%d` format).

`biweeklybudget.settings.`**RECONCILE_BEGIN_DATE = datetime.date(2017, 1, 1)**
`datetime.date` - When listing unreconciled transactions that need to be reconciled, any transaction before this date will be ignored. This must be specified in Y-m-d format (i.e. parsable by `datetime.datetime.strptime()` with `%Y-%m-%d` format).

`biweeklybudget.settings.`**STALE_DATA_TIMEDELTA = datetime.timedelta(2)**
`datetime.timedelta` - Time interval beyond which OFX data for accounts will be considered old/stale. This must be specified as a number (integer) that will be converted to a number of days.

`biweeklybudget.settings.`**STATEMENTS_SAVE_PATH = '/home/docs/ofx'**
string - *(optional)* Filesystem path to download OFX statements to, and for backfill_ofx to read them from.

`biweeklybudget.settings.`**TOKEN_PATH = 'vault_token.txt'**
string - *(optional)* Filesystem path to read Vault token from, for OFX credentials.

`biweeklybudget.settings.`**VAULT_ADDR = 'http://127.0.0.1:8200'**
string - *(optional)* Address to connect to Vault at, for OFX credentials.

biweeklybudget.settings_example module

`biweeklybudget.settings_example.`**DB_CONNSTRING = 'sqlite:///:memory:'**
SQLAlchemy database connection string. Note that the value given in generated documentation is the value used in TravisCI, not the real default.

`biweeklybudget.settings_example.`**DEFAULT_ACCOUNT_ID = 1**
Account ID to show first in dropdown lists

`biweeklybudget.settings_example.`**FUEL_BUDGET_ID = 1**
int - Budget ID to select as default when inputting Fuel Log entries. This must be the database ID of a valid budget.

`biweeklybudget.settings_example.`**PAY_PERIOD_START_DATE = datetime.date(2017, 3, 17)**
The starting date of one pay period. The dates of all pay periods will be determined based on an interval from this date.

`biweeklybudget.settings_example.`**RECONCILE_BEGIN_DATE = datetime.date(2017, 1, 1)**
When listing unreconciled transactions that need to be reconciled, any *`OFXTransaction`* before this date will be ignored.

`biweeklybudget.settings_example.`**STALE_DATA_TIMEDELTA = datetime.timedelta(2)**
`datetime.timedelta` beyond which OFX data will be considered old

`biweeklybudget.settings_example.`**STATEMENTS_SAVE_PATH = '/home/docs/ofx'**
Path to download OFX statements to, and for backfill_ofx to read them from

`biweeklybudget.settings_example.`**TOKEN_PATH = 'vault_token.txt'**
Path to read Vault token from, for OFX credentials

`biweeklybudget.settings_example.`**`VAULT_ADDR`** **= 'http://127.0.0.1:8200'**
Address to connect to Vault at, for OFX credentials

biweeklybudget.utils module

exception `biweeklybudget.utils.`**`SecretMissingException`**(*path*)
Bases: `exceptions.Exception`

class `biweeklybudget.utils.`**`Vault`**(*addr='http://127.0.0.1:8200'*, *token_path='vault_token.txt'*)
Bases: `object`

Provides simpler access to Vault

`read`(*secret_path*)
Read and return a secret from Vault. Return only the data portion.

Parameters **`secret_path`** (*`str`*) – path to read in Vault

Returns secret data

Return type dict

`biweeklybudget.utils.`**`date_suffix`**(*n*)
Given an integer day of month (1 <= n <= 31), return that number with the appropriate suffix (st|nd|rd|th).

From: http://stackoverflow.com/a/5891598/211734

Parameters **`n`** (*`int`*) – Integer day of month

Returns n with the appropriate suffix

Return type str

`biweeklybudget.utils.`**`decode_json_datetime`**(*d*)
Return a datetime.datetime for a datetime that was serialized with *`MagicJSONEncoder`*.

Parameters **`d`** (*`dict`*) – dict from deserialized JSON

Returns datetime represented by dict

Return type datetime.datetime

`biweeklybudget.utils.`**`dtnow`**()
Return the current datetime as a timezone-aware DateTime object in UTC.

Returns current datetime

Return type datetime.datetime

`biweeklybudget.utils.`**`fix_werkzeug_logger`**()
Remove the werkzeug logger StreamHandler (call from `app.py`).

With Werkzeug at least as of 0.12.1, werkzeug._internal._log sets up its own StreamHandler if logging isn't already configured. Because we're using the `flask` command line wrapper, that will ALWAYS be imported (and executed) before we can set up our own logger. As a result, to fix the duplicate log messages, we have to go back and remove that StreamHandler.

`biweeklybudget.utils.`**`in_directory`**(**args*, ***kwds*)

biweeklybudget.version module

biweeklybudget.wishlist2project module

class `biweeklybudget.wishlist2project.`**`WishlistToProject`**

Bases: `object`

`_do_project`(*list_url*, *project*)

Update a project with information from its wishlist.

Parameters

- **`list_url`** (*`str`*) – Amazon wishlist URL
- **`project`** (`Project`) – the project to update

Returns whether or not the update was successful

Return type bool

`_get_wishlist_projects`()

Find all projects with descriptions that begin with a wishlist URL.

Returns list of (url, Project object) tuples

Return type list

`_project_items`(*proj*)

Return all of the BoMItems for the specified project, as a dict of URL to BoMItem.

Parameters **`proj`** (`Project`) – the project to get items for

Returns item URLs to BoMItems

Return type dict

static **`_url_is_wishlist`**(*url*)

Determine if the given string or URL matches a wishlist.

Parameters **`url`** (*`str`*) – URL or string to test

Returns whether url is a wishlist URL

Return type bool

`_wishlist_items`(*list_url*)

Get the items on the specified wishlist.

Parameters **`list_url`** (*`str`*) – wishlist URL

Returns dict of item URL to item details dict

Return type dict

`run`()

Run the synchronization.

Returns 2-tuple; count of successful syncs, total count of projects with associated wishlists

Return type tuple

`biweeklybudget.wishlist2project.`**`main`**()

`biweeklybudget.wishlist2project.`**`parse_args`**()

UI JavaScript Docs

Files

jsdoc.accounts_modal

File: `biweeklybudget/flaskapp/static/js/accounts_modal.js`

accountModal(*id*, *dataTableObj*)

Show the modal popup, populated with information for one account. Uses `accountModalDivFillAndShow()` as ajax callback.

Arguments

- **id** (*number*) – the ID of the account to show modal for, or null to show a modal to add a new account.
- **dataTableObj** (*Object|null*) – passed on to `handleForm()`

accountModalDivFillAndShow(*msg*)

Ajax callback to fill in the modalDiv with data on a account. Callback for ajax call in `accountModal()`.

accountModalDivForm()

Generate the HTML for the form on the Modal

accountModalDivHandleType()

Handle change of the "Type" radio buttons on the modal

jsdoc.bom_items

File: `biweeklybudget/flaskapp/static/js/bom_items.js`

reloadProject()

Reload the top-level project information on the page.

jsdoc.bom_items_modal

File: `biweeklybudget/flaskapp/static/js/bom_items_modal.js`

bomItemModal(*id*)

Show the BoM Item modal popup, optionally populated with information for one BoM Item. This function calls `bomItemModalDivForm()` to generate the form HTML, `bomItemModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the BoM Item to show a modal for, or null to show modal to add a new Transaction.

bomItemModalDivFillAndShow(*msg*)

Ajax callback to fill in the modalDiv with data on a BoM Item.

bomItemModalDivForm()

Generate the HTML for the form on the Modal

jsdoc.budget_transfer_modal

File: `biweeklybudget/flaskapp/static/js/budget_transfer_modal.js`

budgetTransferDivForm()
Generate the HTML for the form on the Modal

budgetTransferModal(*txfr_date*)
Show the modal popup for transferring between budgets. Uses *budgetTransferDivForm()* to generate the form.

Arguments

- **txfr_date** (*string*) – The date, as a "yyyy-mm-dd" string, to default the form to. If null or undefined, will default to `BIWEEKLYBUDGET_DEFAULT_DATE`.

jsdoc.budgets_modal

File: `biweeklybudget/flaskapp/static/js/budgets_modal.js`

budgetModal(*id*, *dataTableObj*)
Show the modal popup, populated with information for one Budget. Uses *budgetModalDivFillAndShow()* as ajax callback.

Arguments

- **id** (*number*) – the ID of the Budget to show modal for, or null to show a modal to add a new Budget.
- **dataTableObj** (*Object|null*) – passed on to `handleForm()`

budgetModalDivFillAndShow(*msg*)
Ajax callback to fill in the modalDiv with data on a budget. Callback for ajax call in *budgetModal()*.

budgetModalDivForm()
Generate the HTML for the form on the Modal

budgetModalDivHandleType()
Handle change of the "Type" radio buttons on the modal

jsdoc.credit_payoffs

File: `biweeklybudget/flaskapp/static/js/credit_payoffs.js`

addIncrease(*settings*)
Link handler to add another "starting on, increase payments by" form to the credit payoff page.

addOnetime(*settings*)
Link handler to add another one time payment form to the credit payoff page.

loadSettings()
Load settings from embedded JSON. Called on page load.

nextIndex(*prefix*)
Return the next index for the form with an ID beginning with a given string.

Arguments

- **prefix** (*string*) – The prefix of the form IDs.

Returns **int** – next form index

recalcPayoffs()
Buttom handler to serialize and submit the forms, to save user input and recalculate the payoff amounts.

removeIncrease(*idx*)
Remove the specified Increase form.

removeOnetime(*idx*)
Remove the specified Onetime form.

serializeForms()
Serialize the form data into an object and return it.

Returns **Object** – serialized forms.

setChanged()
Event handler to activate the "Save & Recalculate" button when user input fields have changed.

jsdoc.custom

File: `biweeklybudget/flaskapp/static/js/custom.js`

fmt_currency(*value*)
Format a float as currency

Arguments

- **value** (*number*) – the number to format

Returns **string** – The number formatted as currency

fmt_null(*o*)
Format a null object as " "

Arguments

- **o** (*Object|null*) – input value

Returns **Object|string** – o if not null, ` ` if null

isoformat(*d*)
Format a javascript Date as ISO8601 YYYY-MM-DD

Arguments

- **d** (*Date*) – the date to format

Returns **string** – YYYY-MM-DD

jsdoc.formBuilder

File: `biweeklybudget/flaskapp/static/js/formBuilder.js`

FormBuilder(*id*)
Create a new FormBuilder to generate an HTML form

Arguments

- **id** (*String*) – The form HTML element ID.

FormBuilder.**addCheckbox**(*id*, *name*, *label*, *checked*)
Add a checkbox to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **checked** (*Boolean*) – Whether to default to checked or not

Returns **FormBuilder** – this

`FormBuilder.`**`addCurrency`**(*id*, *name*, *label*, *options*)

Add a text `input` for currency to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **options** (*Object*) –
- **options.htmlClass** (*String*) – The HTML class to apply to the element; defaults to `form-control`.
- **options.helpBlock** (*String*) – Content for block of help text after input; defaults to null.
- **options.groupHtml** (*String*) – Additional HTML to add to the outermost form-group div. This is where we'd usually add a default style/display. Defaults to null.

Returns **FormBuilder** – this

`FormBuilder.`**`addDatePicker`**(*id*, *name*, *label*, *options*)

Add a date picker input to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **options** (*Object*) –
- **options.groupHtml** (*String*) – Additional HTML to add to the outermost

Returns **FormBuilder** – this

`FormBuilder.`**`addHTML`**(*content*)

Add a string of HTML to the form.

Arguments

- **content** (*String*) – HTML

Returns **FormBuilder** – this

`FormBuilder.`**`addHidden`**(*id*, *name*, *value*)

Add a hidden `input` to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element

- **value** (*String*) – The value of the form element

Returns **FormBuilder** – this

`FormBuilder.addLabelToValueSelect`(*id*, *name*, *label*, *selectOptions*, *defaultValue*, *addNone*, *options*)

Add a select element to the form, taking an Object of options where keys are the labels and values are the values. This is a convenience wrapper around `budgetTransferDivForm()`.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **selectOptions** (*Object*) – the options for the select, label to value
- **defaultValue** (*String*) – A value to select as the default
- **addNone** (*Boolean*) – If true, prepend an option with a value of "None" and an empty label.
- **options** (*Object*) – Options for rendering the control. Passed through unmodified to `FormBuilder.addSelect()`; see that for details.

Returns **FormBuilder** – this

`FormBuilder.addP`(*content*)

Add a paragraph (`p` tag) to the form.

Arguments

- **content** (*String*) – The content of the `p` tag.

Returns **FormBuilder** – this

`FormBuilder.addRadioInline`(*name*, *label*, *options*)

Add an inline radio button set to the form.

Options is an Array of Objects, each object having keys `id`, `value` and `label`. Optional keys are `checked` (Boolean) and `onchange`, which will have its value placed literally in the HTML.

Arguments

- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **options** (*Array*) – the options for the select; array of objects each having the following attributes:
- **options.id** (*String*) – the ID for the option
- **options.value** (*String*) – the value for the option
- **options.label** (*String*) – the label for the option
- **options.checked** (*Boolean*) – whether the option should be checked by default *(optional; defaults to false)*
- **options.inputHtml** (*String*) – extra HTML string to include in the actual `input` element *(optional; defaults to null)*

Returns **FormBuilder** – this

`FormBuilder.addSelect`(*id*, *name*, *label*, *selectOptions*, *options*)

Add a select element to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **selectOptions** (*Array*) – the options for the select, array of objects (order is preserved) each having the following attributes:
- **selectOptions.label** (*String*) – the label for the option
- **selectOptions.value** (*String*) – the value for the option
- **selectOptions.selected** (*Boolean*) – whether the option should be the default selected value *(optional; defaults to False)*
- **options** (*Object*) –
- **options.htmlClass** (*String*) – The HTML class to apply to the element; defaults to `form-control`.
- **options.helpBlock** (*String*) – Content for block of help text after input; defaults to null.
- **options.groupHtml** (*String*) – Additional HTML to add to the outermost form-group div. This is where we'd usually add a default style/display. Defaults to null.

Returns **FormBuilder** – this

`FormBuilder.addText`(*id*, *name*, *label*, *options*)

Add a text `input` to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element
- **options** (*Object*) –
- **options.groupHtml** (*String*) – Additional HTML to add to the outermost
- **options.inputHtml** (*String*) – extra HTML string to include in the actual `input` element *(optional; defaults to null)*
- **options.helpBlock** (*String*) – Content for block of help text after input; defaults to null.

Returns **FormBuilder** – this

`FormBuilder.addTextArea`(*id*, *name*, *label*, *options*)

Add a Text Area to the form.

Arguments

- **id** (*String*) – The id of the form element
- **name** (*String*) – The name of the form element
- **label** (*String*) – The label text for the form element

- **options** (*Object*) –
- **options.groupHtml** (*String*) – Additional HTML to add to the outermost
- **options.inputHtml** (*String*) – extra HTML string to include in the actual `input` element *(optional; defaults to null)*
- **options.helpBlock** (*String*) – Content for block of help text after input; defaults to null.

Returns **FormBuilder** – this

`FormBuilder.render()`

Return complete rendered HTML for the form.

Returns **String** – form HTML

jsdoc.forms

File: `biweeklybudget/flaskapp/static/js/forms.js`

handleForm (*container_id*, *form_id*, *post_url*, *dataTableObj*)

Generic function to handle form submission with server-side validation.

See the Python server-side code for further information.

Arguments

- **container_id** (*string*) – The ID of the container element (div) that is the visual parent of the form. On successful submission, this element will be emptied and replaced with a success message.
- **form_id** (*string*) – The ID of the form itself.
- **post_url** (*string*) – Relative URL to post form data to.
- **dataTableObj** (*Object*) – passed on to `handleFormSubmitted()`

handleFormError (*jqXHR*, *textStatus*, *errorThrown*, *container_id*, *form_id*)

Handle an error in the HTTP request to submit the form.

handleFormSubmitted (*data*, *container_id*, *form_id*, *dataTableObj*)

Handle the response from the API URL that the form data is POSTed to.

This should either display a success message, or one or more error messages.

Arguments

- **data** (*Object*) – response data
- **container_id** (*string*) – the ID of the modal container on the page
- **form_id** (*string*) – the ID of the form on the page
- **dataTableObj** (*Object*) – A reference to the DataTable on the page, that needs to be refreshed. If null, reload the whole page. If a function, call that function. If false, do nothing.

handleInlineForm (*container_id*, *form_id*, *post_url*, *dataTableObj*)

Generic function to handle form submission with server-side validation of an inline (non-modal) form.

See the Python server-side code for further information.

Arguments

- **container_id** (*string*) – The ID of the container element (div) that is the visual parent of the form. On successful submission, this element will be emptied and replaced with a success message.
- **form_id** (*string*) – The ID of the form itself.
- **post_url** (*string*) – Relative URL to post form data to.
- **dataTableObj** (*Object*) – passed on to `handleFormSubmitted()`

handleInlineFormError (*jqXHR*, *textStatus*, *errorThrown*, *container_id*, *form_id*)
Handle an error in the HTTP request to submit the inline (non-modal) form.

handleInlineFormSubmitted (*data*, *container_id*, *form_id*, *dataTableObj*)
Handle the response from the API URL that the form data is POSTed to, for an inline (non-modal) form.

This should either display a success message, or one or more error messages.

Arguments

- **data** (*Object*) – response data
- **container_id** (*string*) – the ID of the modal container on the page
- **form_id** (*string*) – the ID of the form on the page
- **dataTableObj** (*Object*) – A reference to the DataTable on the page, that needs to be refreshed. If null, reload the whole page. If a function, call that function. If false, do nothing.

isFunction (*functionToCheck*)
Return True if `functionToCheck` is a function, False otherwise.

From: http://stackoverflow.com/a/7356528/211734

Arguments

- **functionToCheck** (*Object*) – The object to test.

serializeForm (*form_id*)
Given the ID of a form, return an Object (hash/dict) of all data from it, to POST to the server.

Arguments

- **form_id** (*string*) – The ID of the form itself.

jsdoc.fuel

File: `biweeklybudget/flaskapp/static/js/fuel.js`

fuelLogModal (*dataTableObj*)
Show the modal to add a fuel log entry. This function calls *`fuelModalDivForm()`* to generate the form HTML, *`schedModalDivFillAndShow()`* to populate the form for editing, and *`handleForm()`* to handle the Submit action.

Arguments

- **dataTableObj** (*Object|null*) – passed on to *`handleForm()`*

fuelModalDivForm()
Generate the HTML for the form on the Modal

vehicleModal (*id*)
Show the Vehicle modal popup, optionally populated with information for one Vehicle. This function calls

`vehicleModalDivForm()` to generate the form HTML, `vehicleModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the Vehicle to show a modal for, or null to show modal to add a new Vehicle.

vehicleModalDivFillAndShow(*msg*)
Ajax callback to fill in the modalDiv with data on a Vehicle.

vehicleModalDivForm()
Generate the HTML for the form on the Modal

jsdoc.ofx

File: `biweeklybudget/flaskapp/static/js/ofx.js`

ofxTransModal(*acct_id*, *fitid*)
Show the modal popup, populated with information for one OFX Transaction.

jsdoc.payperiod_modal

File: `biweeklybudget/flaskapp/static/js/payperiod_modal.js`

schedToTransModal(*id*, *payperiod_start_date*)
Show the Scheduled Transaction to Transaction modal popup. This function calls `schedToTransModalDivForm()` to generate the form HTML, `schedToTransModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the ScheduledTransaction to show a modal for.
- **payperiod_start_date** (*string*) – The Y-m-d starting date of the pay period.

schedToTransModalDivFillAndShow(*msg*)
Ajax callback to fill in the modalDiv with data on a budget.

schedToTransModalDivForm()
Generate the HTML for the form on the Modal

skipSchedTransModal(*id*, *payperiod_start_date*)
Show the Skip Scheduled Transaction modal popup. This function calls `skipSchedTransModalDivForm()` to generate the form HTML, `skipSchedTransModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the ScheduledTransaction to show a modal for.
- **payperiod_start_date** (*string*) – The Y-m-d starting date of the pay period.

skipSchedTransModalDivFillAndShow(*msg*)
Ajax callback to fill in the modalDiv with data on a budget.

skipSchedTransModalDivForm()
Generate the HTML for the form on the Modal

jsdoc.projects

File: `biweeklybudget/flaskapp/static/js/projects.js`

activateProject (*proj_id*)
Handler for links to activate a project.

deactivateProject (*proj_id*)
Handler for links to deactivate a project.

handleProjectAdded ()
Handler for when a project is added via the form.

jsdoc.reconcile

File: `biweeklybudget/flaskapp/static/js/reconcile.js`

clean_fitid (*fitid*)
Given an OFXTransaction fitid, return a "clean" (alphanumeric) version of it, suitable for use as an HTML element id.

Arguments

- **fitid** (*String*) – original, unmodified OFXTransaction fitid.

makeTransFromOfx (*acct_id*, *fitid*)
Link function to create a Transaction from a specified OFXTransaction, and then reconcile them.

Arguments

- **acct_id** (*Integer*) – the OFXTransaction account ID
- **fitid** (*String*) – the OFXTransaction fitid

makeTransSaveCallback (*data*, *acct_id*, *fitid*)
Callback for the "Save" button on the Transaction modal created by `makeTransFromOfx()`. Displays the new Transaction at the bottom of the Transactions list, then reconciles it with the original OFXTransaction

Arguments

- **data** (*Object*) – response data from POST to /forms/transaction
- **acct_id** (*Integer*) – the OFXTransaction account ID
- **fitid** (*String*) – the OFXTransaction fitid

reconcileDoUnreconcile (*trans_id*, *acct_id*, *fitid*)
Unreconcile a reconciled OFXTransaction/Transaction. This removes `trans_id` from the `reconciled` variable, empties the Transaction div's reconciled div, and shows the OFX div.

Arguments

- **trans_id** (*Integer*) – the transaction id
- **acct_id** (*Integer*) – the account id
- **fitid** (*String*) – the FITID

reconcileDoUnreconcileNoOfx (*trans_id*)
Unreconcile a reconciled NoOFX Transaction. This removes `trans_id` from the `reconciled` variable and empties the Transaction div's reconciled div.

Arguments

- **trans_id** (*Integer*) – the transaction id

reconcileGetOFX()

Show unreconciled OFX transactions in the proper div. Empty the div, then load transactions via ajax. Uses `reconcileShowOFX()` as the ajax callback.

reconcileGetTransactions()

Show unreconciled transactions in the proper div. Empty the div, then load transactions via ajax. Uses `reconcileShowTransactions()` as the ajax callback.

reconcileHandleSubmit()

Handle click of the Submit button on the reconcile view. This POSTs to `/ajax/reconcile` via ajax. Feedback is provided by appending a div with id `reconcile-msg` to `div#notifications-row/div.col-lg-12`.

reconcileOfxDiv(*trans*)

Generate a div for an individual OFXTransaction, to display on the reconcile view.

Arguments

- **ofxtrans** (*Object*) – ajax JSON object representing one OFXTransaction

reconcileShowOFX(*data*)

Ajax callback handler for `reconcileGetOFX()`. Display the returned data in the proper div.

Arguments

- **data** (*Object*) – ajax response (JSON array of OFXTransaction Objects)

reconcileShowTransactions(*data*)

Ajax callback handler for `reconcileGetTransactions()`. Display the returned data in the proper div.

Sets each Transaction div as `droppable`, using `reconcileTransHandleDropEvent()` as the drop event handler and `reconcileTransDroppableAccept()` to test if a draggable is droppable on the element.

Arguments

- **data** (*Object*) – ajax response (JSON array of Transaction Objects)

reconcileTransDiv(*trans*)

Generate a div for an individual Transaction, to display on the reconcile view.

Arguments

- **trans** (*Object*) – ajax JSON object representing one Transaction

reconcileTransDroppableAccept(*drag*)

Accept function for droppables, to determine if a given draggable can be dropped on it.

Arguments

- **drag** (*Object*) – the draggable element being dropped.

reconcileTransHandleDropEvent(*event*, *ui*)

Handler for Drop events on reconcile Transaction divs. Setup as handler via `reconcileShowTransactions()`. This just gets the draggable and the target from the `event` and `ui`, and then passes them on to `reconcileTransactions()`.

Arguments

- **event** (*Object*) – the drop event
- **ui** (*Object*) – the UI element, containing the draggable

reconcileTransNoOfx (*trans_id*, *note*)
Reconcile a Transaction without a matching OFXTransaction. Called from the Save button handler in `transNoOfx()`.

reconcileTransactions (*ofx_div*, *target*)
Reconcile a transaction; move the divs and other elements as necessary, and updated the `reconciled` variable.

Arguments

- **ofx_div** (*Object*) – the OFXTransaction div element (draggable)
- **target** (*Object*) – the Transaction div (drop target)

transModalOfxFillAndShow (*data*)
Callback for the GET /ajax/ofx/<acct_id>/<fitid> from `makeTransFromOfx()`. Receives the OFXTransaction data and populates it into the Transaction modal form.

Arguments

- **data** (*Object*) – OFXTransaction response data

transNoOfx (*trans_id*)
Show the modal for reconciling a Transaction without a matching OFXTransaction. Calls `transNoOfxDivForm()` to generate the modal form div content. Uses an inline function to handle the save action, which calls `reconcileTransNoOfx()` to perform the reconcile action.

Arguments

- **trans_id** (*number*) – the ID of the Transaction

transNoOfxDivForm (*trans_id*)
Generate the modal form div content for the modal to reconcile a Transaction without a matching OFXTransaction. Called by `transNoOfx()`.

Arguments

- **trans_id** (*number*) – the ID of the Transaction

updateReconcileTrans (*trans_id*)
Trigger update of a single Transaction on the reconcile page.

Arguments

- **trans_id** (*Integer*) – the Transaction ID to update.

jsdoc.reconcile_modal

File: `biweeklybudget/flaskapp/static/js/reconcile_modal.js`

txnReconcileModal (*id*)
Show the TxnReconcile modal popup. This function calls `txnReconcileModalDiv()` to generate the HTML.

Arguments

- **id** (*number*) – the ID of the TxnReconcile to show a modal for.

txnReconcileModalDiv (*msg*)
Ajax callback to generate the modal HTML with reconcile information.

jsdoc.scheduled_modal

File: `biweeklybudget/flaskapp/static/js/scheduled_modal.js`

schedModal(*id*, *dataTableObj*)

Show the ScheduledTransaction modal popup, optionally populated with information for one ScheduledTransaction. This function calls `schedModalDivForm()` to generate the form HTML, `schedModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the ScheduledTransaction to show a modal for, or null to show modal to add a new ScheduledTransaction.
- **dataTableObj** (*Object|null*) – passed on to `handleForm()`

schedModalDivFillAndShow(*msg*)

Ajax callback to fill in the modalDiv with data on a budget.

schedModalDivForm()

Generate the HTML for the form on the Modal

schedModalDivHandleType()

Handle change of the "Type" radio buttons on the modal

jsdoc.transactions_modal

File: `biweeklybudget/flaskapp/static/js/transactions_modal.js`

transModal(*id*, *dataTableObj*)

Show the Transaction modal popup, optionally populated with information for one Transaction. This function calls `transModalDivForm()` to generate the form HTML, `transModalDivFillAndShow()` to populate the form for editing, and `handleForm()` to handle the Submit action.

Arguments

- **id** (*number*) – the ID of the Transaction to show a modal for, or null to show modal to add a new Transaction.
- **dataTableObj** (*Object|null*) – passed on to `handleForm()`

transModalDivFillAndShow(*msg*)

Ajax callback to fill in the modalDiv with data on a Transaction.

transModalDivForm()

Generate the HTML for the form on the Modal

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X