Acorn Accounting Documentation

Release 0.12.0

Pavan Rikhi

September 20, 2018

Contents

1 Introduction **1**
- 1.1 Documentation Overview . . . 1
- 1.2 Program Overview . . . 1

2 Installation & Configuration **3**
- 2.1 Downloading . . . 3
- 2.2 Install Prerequisites . . . 3
- 2.3 Configuration . . . 4
- 2.4 Deployment . . . 4
- 2.5 Building the Documentation . . . 7

3 User Guide **9**
- 3.1 Tips . . . 9

4 Development Standards and Contributing **11**
- 4.1 Bug Reports and Feature Requests . . . 11
- 4.2 Development Quickstart . . . 12
- 4.3 Code Conventions . . . 12
- 4.4 Version Control . . . 13
- 4.5 Version Numbers . . . 16
- 4.6 Tests . . . 16
- 4.7 Documentation . . . 17
- 4.8 Specifications . . . 17

5 Specification Explanation **19**
- 5.1 Use Cases . . . 19
- 5.2 Use Case Diagrams . . . 19
- 5.3 Screens and Conditions . . . 19
- 5.4 Activity Diagrams . . . 20

6 Design Specifications **21**
- 6.1 Use Cases . . . 21
- 6.2 Screen Descriptions . . . 22
- 6.3 Future Features . . . 43

7 Technical Specifications **47**
- 7.1 core Package . . . 47
- 7.2 accounts Package . . . 50
- 7.3 entries Package . . . 53

7.4 fiscalyears Package . . . 61
7.5 events Package . . . 63
7.6 reports Package . . . 64
7.7 creditcards Package . . . 65
7.8 trips Package . . . 67
7.9 bank_import Package . . . 68

8 Appendix: Installlation & Deployment Guides **73**
8.1 Slackware Linux . . . 73

9 Glossary **81**

10 Indices and tables **83**

Python Module Index **85**

CHAPTER 1

Introduction

The purpose of this Documentation is to define what this application does in a simple language. If you have used www.google.com, you should have enough knowledge to understand this document. If you are having trouble, it's not because you lack technical experience but rather because the author has failed to express themselves clearly.

The authors beg you to please inform us when you find any confusing or unclear passages so they may be reviewed.

If Specifications and Documentation scare you, you may want to first read our *Specification Explanation*.

Documentation Overview

This Document defines the AcornAccounting Application, and provides guides for users and contributers.

The End-User experience is defined by the *Design Specifications*. The *User Guide* will help new users become acquainted with interacting with the application by providing an overview of all possible actions and effects.

Developers and Contributers should reference the *Development Standards and Contributing* and *Technical Specifications* Sections for information on Best Coding Practices and the current public *API*.

Program Overview

AcornAccounting is an Open Source, Web-based, Double Entry Accounting System for (Egalitarian) Communities.

Major Features

AcornAccounting has these notable features:

- Trip Entry Form - Tripper-friendly entry form with Accountant approval
- People Tracking - Count the number of Intern/Visitor days, automatically pay monthly and yearly stipends
- Online Bank Statements - Download statements to make entry easy
- Open Source - Easily add custom reports or entry types

Goals

AcornAccounting strives to be both Accountant and Communard friendly.

Entry and Reporting should be streamlined for Accountants and the UI and workflow should be accessible to Communards. Automation will be used to lighten repetitive workloads for Accountants.

Non-accountants should feel comfortable checking their Account balances and Project budgets.

Non-Goals

AcornAccounting does **not** try to be:

1. An international Accounting system(only English and USD is currently supported).
2. Customer Relationship Management Software
3. Enterprise Resource Planning Software
4. Payroll Accounting Software

Technology

AcornAccounting is written in Python, using the Django Web Framework. The UI is built using the Bootstrap CSS and Javascript GUI Framework. Clientside validation will be performed by custom Javascript and Parsley.js.

Django contains many helper apps/plugins. Some apps we use include:

- Cache Machine for caching querys and objects.
- Djanjo-parsley for integration with Parsley.js.
- South to automate database changes/migrations.
- MPTT for handling object hierarchies.

Version Control is handled with Git. The automated deployment script uses Fabric.

Documentation and *API* Specifications will be written in reStructuredText for use with Sphinx. UML diagrams will be generated from the documentation using plantUML.

CHAPTER 2

Installation & Configuration

Downloading

Pip Install Directions

v1.0.0 should be hosted on PyPi so installing is as easy as:

```
$ pip install acornaccounting
```

See The Hitchhiker's Guide to Packaging for information on how to get that set up.

Git Clone Directions

Developers will want to clone the current source code repository:

```
$ git clone git@aphrodite.acorn:~/Acorn-Accounting.git/
```

The current public release is located on the `master` branch while new development occurs on the `develop` branch.

See *Development Standards and Contributing* for more information.

Install Prerequisites

You should install python 2 and pip via your package manager.

On Arch Linux:

```
$ sudo pacman -S python2 python2-pip
```

On Slackware:

```
$ sudo /usr/sbin/slackpkg install python
$ wget https://raw.github.com/pypa/pip/master/contrib/get-pip.py
$ sudo python get-pip.py
```

On Debian/Ubuntu:

```
$ sudo apt-get install python-pip
```

Optionally you may want to install virtualenv and virtualenvwrapper to manage and isolate the python dependencies.

```
$ sudo pip install virtualenv virtualenvwrapper
```

Make sure to do the initial setup for virtualenv:

```
$ export WORKON_HOME=~/.virtualenv/
$ mkdir -p $WORKON_HOME
$ source virtualenvwrapper.sh
```

Then you may create an environment for AcornAccounting:

```
$ mkvirtualenv AcornAccounting
```

You may then install dependencies into this virtual environment. There are multiple tiers of dependencies:

- `base` - minimum requirements needed to run the application
- `test` - requirements necessary for running the test suite
- `local` - development prerequisites such as the debug toolbar and documentation builders
- `production` - all packages required for real world usage

A set of dependencies may be installed via pip:

```
$ workon AcornAccounting
$ pip install -r requirements/develop.txt
```

Configuration

Some settings are set through environmental variables instead of files. These include settings with sensitive information, and allows us to keep the information out of version control.

You may set these variables directly in the terminal or add them to your virtualenv's `activate` script:

```
$ DB_USER='prikhi' DB_NAME='DjangoAccounting' ./manage.py runserver
$ export DB_NAME='DjangoAccounting'
$ ./manage.py runserver
```

The required environmental variables are `DJANGO_SECRET_KEY`, `DB_NAME` and `DB_USER`.

Deployment

It is recommended to use uWSGI for serving the dynamic pages and either Apache or Nginx for serving your static files.

Create and Initialize the Database

You'll need a new Database and User, if you use PostgreSQL you may run:

```
su - postgres
createuser -DERPS accounting
createdb accounting -O accounting
```

Set configuration values for the account you just created:

```
export DJANGO_SETTINGS_MODULE=accounting.settings.production
export DB_USER=accounting
export DB_PASSWORD=<accounting user password>
export DB_NAME=accounting
```

Then create the initial schema and migrate any database changes:

```
cd acornaccounting
python manage.py syncdb
python manage.py migrate
```

Collect Static Files

Next collect all the static files into the directory you will serve them out of:

```
python manage.py collectstatic
```

Configure uWSGI

You can use the following ini file to setup the uWSGI daemon:

```
[uwsgi]
uid = <your accounting user>
gid = %(uid)
chdir = <acornaccounting project root>

plugin = python
pythonpath = %(chdir)
virtualenv = </path/to/virtualenv/>
module = django.core.handlers.wsgi:WSGIHandler()

socket = 127.0.0.1:3031
master = true
workers = 10
max-requests = 5000
vacuum = True

daemonize = /var/log/accounting/uwsgi.log
pidfile = /var/run/accounting.pid
touch-reload = /tmp/accounting.touch

env = DJANGO_SETTINGS_MODULE=accounting.settings.production
env = DB_NAME=<database name
env = DB_USER=<database user>
env = DB_PASSWORD=<database password>
env = DB_HOST=
env = DB_PORT=
env = DJANGO_SECRET_KEY=<your unique secret key>
env = CACHE_LOCATION=127.0.0.1:11211
```

Make sure to review this and replace the necessary variables.

Note: If you do not have a secure, unique secret key, you may generate one by running the following in the Python interpreter:

```
import random
print(''.join(
    [random.SystemRandom().choice(
        'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)')
     for i in range(50)])
)
```

Depending on your OS, you may need to put this file in `/etc/uwsgi/apps-available` then link it to `/etc/uwsgi/apps-enabled/`. Or you may need to write an rc.d or init.d startup script:

```
#!/bin/bash
#
# Start/Stop/Restart the Accounting uWSGI server
#
# To make the server start at boot make this file executable:
#
#       chmod 755 /etc/rc.d/rc.accounting

INIFILE=/etc/uwsgi/accounting.ini
PIDFILE=/var/run/accounting.pid

case "$1" in
    'start')
        echo "Starting the Accounting uWSGI Process."
        uwsgi -i $INIFILE
        ;;
    'stop')
        echo "Stopping the Accounting uWSGI Process."
        uwsgi --stop $PIDFILE
        rm $PIDFILE
        ;;
    'restart')
        echo "Restarting the Accounting uWSGI Process."
        if [ -f $PIDFILE ]; then
            uwsgi --reload $PIDFILE
        else
            echo "Error: No Accounting uWSGI Process Found."
        fi
        ;;
    'status')
        if [ -f $PIDFILE ] && [ "$(ps -o comm= "$(cat $PIDFILE)")" = uwsgi ]; then
            echo "Accounting uWSGI Process is running."
        else
            echo "Accounting uWSGI Process is not running."
        fi
        ;;
    *)
        echo "Usage: /etc/rc.d/rc.accounting {start|stop|restart|status}"
        exit 1
        ;;
esac

exit 0
```

Apache VirtualHost

The Virtual Host should redirect every request, except those to `/static`, to the uWSGI handler:

```
<VirtualHost *:80>
    ServerName accounting.yourdomain.com
    DocumentRoot "/srv/accounting/"
    Alias /static /srv/accounting/static/
    <Directory "/srv/accounting/">
```

```
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Require all granted
    </Directory>
    <Location />
        Options FollowSymLinks Indexes
        SetHandler uwsgi-handler
        uWSGISocket 127.0.0.1:3031
    </Location>
    <Location /static>
        SetHandler none
    </Location>
    ErrorLog "/var/log/httpd/accounting-error_log"
    CustomLog "/var/log/httpd/accounting-access_log" common
</VirtualHost>
```

Note that in the above setup, `/srv/accounting/` is linked to the Django project's root directory `acornaccounting`.

1-Step Deployment

1-step deploy script and indepth instuctions, with example apache and uwsgi configs.

Look into fabric for automated deployment. Deploying Django with Fabric

Ideally we would be able to run something like `fab deploy_initial` and `fab deploy`.

We can use fab templates, putting samples/templates in the `/conf/` directory.

v1.0.0 should include a 1-step build/deployment file.

Building the Documentation

pip may be used to install most prerequisites required:

```
$ pip install -r requirements/local.txt
```

Java is optional, but required to create the plantUML images. You can install it via your package manager.

On Arch Linux:

```
$ sudo pacman -S jre7-openjdk
```

On Debian:

```
$ sudo apt-get install default-jdk
```

On Slackware you must manually download the source from Oracle, available here. You may then use the slackbuild at http://slackbuilds.org/repository/14.1/development/jdk/ to install the package:

```
$ wget http://slackbuilds.org/slackbuilds/14.0/development/jdk.tar.gz
$ tar xfz jdk.tar.gz
$ cd jdk
$ mv ~/jdk-7*-linux-*.tar.gz .
$ ./jdk.SlackBuild
$ sudo installpkg /tmp/jdk-7u45-x86_64-1_SBo.tgz
# Add java to your $PATH:
$ sudo ln -s /usr/lib64/java/jre/bin/java /usr/bin/java
```

You can now build the full documentation in HTML or PDF format:

```
$ cd docs/
$ make html
$ make latexpdf
```

The output files will be located in `docs/build/html` and `docs/build/latex`.

CHAPTER 3

User Guide

Something goes here _eventually_.

Maybe something like:

1. Configuration
2. Your First 5 Minutes
3. Headers
4. Accounts
5. Events
6. Entrys
7. Fiscal Years
8. Reports

Each section should talk about creating, editing and viewing details. Maybe sections for various pages:

1. The Bank Register
2. The General Ledger
3. The Chart of Accounts

This is very low priority

Tips

While the user guide is incomplete, this list will exist for important tips that maybe not be immediately obvious:

- No dates require leading zeros. *1/5/15* is the same as *01/05/2015*.
- You can press *Enter* in the tables of the *Add/Edit Entry* pages to go down to the next row.
- For Bank Spending & Receiving Entries, you can leave the memo blank & it will be automatically filled with the first Account you select.
- Chrome does odd things like popping open any dropdown when you hit *Spacebar*. Using Firefox is highly recommended.

CHAPTER 4

Development Standards and Contributing

Bug Reports and Feature Requests

Bugs and New Features should be reported on either the internal Redmine bug tracker or on the public Github Page.

All tickets should have descriptive titles.

Bugs should contain at least three sections: Steps to Reproduce, Results, and Expected Results. Accurate and Concise reproduction steps reduces the amount of time spent reproducing/debugging and increases the proportion of time spent fixing the bug.

An example of an ideal bug report:

```
Title: Account's Reconciled Balance Changes After Creating 2nd Fiscal Year

Steps to Reproduce:
1. Create Fiscal Year
2. Reconcile Account w/ Statement in Current Fiscal Year
3. Create New Fiscal Year
4. Visit Reconcile Page for Account

Result:
Reconciled Balance is different from Statement Balance entered in step 2

Expected Result:
Reconciled Balance should be same as last reconciled Statement Balance

Notes:
Bug is caused by accounts/views.py#L289 on commit 1d7762a0

Currently Sums all reconciled Transactions to get the reconciled balance.
Instead, create a "Reconciled Balance" field on the Account model to store
the Reconciled Balance after reconciling the Account.
```

Your bug report does not need to be anywhere close to this ideal, but the more that you can incorporate, the faster a fix can be developed. The most important parts to include are accurate and minimal steps to reproduce the bug and the expected and actual results.

Features can have simple descriptions, but if Requests for Information from developers are not replied to, then Feature Specifications will be determined by the developer. To expedite development of a feature, it is recommended to submit a *Specification* with the Ticket.

For example, an ideal feature request for a new Report would include:

- The name of the report

- How to reach the new report
- What the report should display(in all possible states)
- Optionally, how to calculate that information
- Any page interactions possible(changing the date range, sorting by column, etc.)
- Any new ways to exit the page(links or forms)
- Any behind the scenes changes(i.e. database changes)

A *Screen Description* and *wireframe* will be created by the requester and the task assignee.

Examples of complete specifications can be found in the *Design Specifications*.

Development Quickstart

First, start by cloning the source code repository:

```
$ git clone git@aphrodite.acorn:~/Acorn-Accounting.git
$ cd Acorn-Accounting
```

Create a new Python Virtual Environment:

```
$ mkvirtualenv AcornAccounting -p python2
```

Install the development prerequisites:

```
$ pip install -r requirements/local.txt
```

Setup the database and migrations, note that you must export a `DJANGO_SETTINGS_MODULE` or specify the settings module:

```
$ cd acornaccounting/
$ ./manage.py syncdb --settings=accounting.settings.local
$ ./manage.py migrate --settings=accounting.settings.local
```

Run the development server:

```
$ ./manage.py runserver localhost:8000 --settings=accounting.settings.local
```

You should now have a working copy of the application on your workstation, accessible at http://localhost:8000/.

To allow the application to be served to other computers, bind the server to all available IP addresses instead of `localhost`:

```
$ ./manage.py runserver 0.0.0.0:8000 --settings=accounting.settings.local
```

Code Conventions

The **PEP 8** is our baseline for coding style.

In short we use:

- 4 spaces per indentation
- 79 characters per line
- One import per line, grouped in the following order: standard library, 3rd party imports, local application imports

- One statement per line
- Docstrings for all public modules, functions, classes and methods.

The following naming conventions should be followed:

- Class names use CapitalWords
- Function names are lowercase, with words separated by underscores
- Use `self` and `cls` for first argument to instance and class methods, respectively.
- Non-public methods and variables should be prefixed with an underscore
- Constants in all uppercase.

Code should attempt to be idiomatic/pythonic, for example:

- Use list, dict and set comprehensions.
- Test existence in a sequence with `in`.
- Use `enumerate` instead of loop counters.
- Use `with ... as ...` for context managers.
- Use `is` to compare against `None` instead of `==`.
- Use parenthesis instead of backslashes for line continuations.

For more information and full coverage of conventions, please read **PEP 8**, **PEP 257**, **PEP 20** and the Django Coding Style Documentation.

There are tools available to help assess compliance to these conventions, such as `pep8` and `pylint`. Both of these tools are installed via `pip`:

```
$ pip install pep8
$ pip install pylint
```

You may then run `pep8` on files to determine their compliance:

```
$ pep8 accounts/signals.py
accounts/signals.py:26:80: E501 line too long (116 > 79 characters)
```

Pylint may be used to show compliance to best practices and give your code a generalized score. It is recommended to run pylint with some files and warnings ignored, to reduce the amount of clutter and false positives:

```
$ pylint --ignore=tests.py,migrations,wsgi.py,settings.py   \
    -d R0904,R0903,W0232,E1101,E1103,W0612,W0613,R0924
```

Version Control

AcornAccounting uses Git as a Version Control System.

Branches

We have 2 long-term public branches:

- `master` - The latest stable release. This branch should be tagged with a new version number every time a branch is merged into it.
- `develop` - The release currently in development. New features and releases originate from this branch.

There are also multiple short-term supporting branches:

- `hotfix` - Used for immediate changes that need to be pushed out into production. These branches should originate from `master` and be merged into `master` and either the `develop` or current `release` if one exists.
- `feature` - Used for individual features and bug fixes, these branches are usually kept on local development machines. These should originate from and be merged back into `develop`.
- `release` - Used for preparing the `develop` branch for merging into `master`, creating a new release. These branches should originate from `develop` and be merged back into `develop` and `master`. Releases should be created when all new features for a version are finished. Any new commits should only contain code refactoring and bug fixes.

This model is adapted from A Successful Git Branching Model, however we use a linear history instead of a branching history, so the `--no-ff` option should be omitted during merges.

Commit Messages

Commit messages should follow the format described in A Note About Git Commit Messages. They should generally follow the following format:

```
[TaskID#] Short 50 Char or Less Title

Explanatory text or summary describing the feature or bugfix, capped
at 72 characters per line, written in the imperative.

Bullet points are also allowed:

* Add method `foo` to `Bar` class
* Modify `Base` class to be abstract
* Remove `foobaz` method from `Bar` class
* Refactor `bazfoo` function

Refs/Closes/Fixes #TaskID: Task Name in Bug Tracker
```

For example:

```
[#142] Add Account History

* Add `HistoricalAccount` model to store archived Account information
* Add `show_account_history` view to display Historical Accounts by
  month
* Add Account History template and Sidebar link to Account History Page

Closes #142: Add Historical Account Record
```

Workflow

The general workflow we follow is based on A Git Workflow for Agile Teams.

Work on a new task begins by branching from `develop`. Feature branch names should be in the format of `tasknumber-short-title-or-name`:

```
$ git checkout -b 142-add-account-history develop
```

Commits on this branch should be early and often. These commit messages are not permanent and do not have to use the format specified above.

You should fetch and rebase against the upstream repository often in order to prevent merging conflicts:

```
$ git fetch origin develop
$ git rebase origin/develop
```

When work is done on the task, you should rebase and squash your many commits into a single commit:

```
$ git rebase -i origin/develop
```

You may then choose which commits to reorder, squash or reword.

> **Warning:** Only rebase commits that have not been published to public branches. Otherwise problems will arise in every other user's local repository. NEVER rewrite public branches and NEVER force a push unless you know EXACTLY what you are doing, and have preferably backed up the upstream repository.

Afterwards, merge your changes into `develop` and push your changes to the upstream repository:

```
$ git checkout develop
$ git merge 142-add-account-history
$ git push origin develop
```

Preparing a Release

A Release should be forked off of develop into its own branch once all associated features are completed. A `release` branch should contain only bugfixes and version bumps.

1. Fork the `release` branch off of the `develop` branch:

   ```
   $ git checkout -b release-1.2.0 develop
   ```

2. Branch, Fix and Merge any existing bugs.
3. Bump the version number and year in `setup.py` and `docs/source/conf.py`.
4. Commit the version changes:

   ```
   $ git commit -a -m "Prepare v1.2.0 Release"
   ```

5. Create a new annotated and signed Tag for the commit:

   ```
   $ git tag -s -a v1.2.0
   ```

   The annotation should contain the release's name and number, and optionally a short description:

   ```
   Version 1.2.0 Release - Trip Entry Form

   This release adds a Trip Entry form for Communards and a Trip Approving
   form for Accountants.
   ```

6. Merge the branch into the `master` branch and push it upstream:

   ```
   $ git checkout master
   $ git merge release-1.2.0
   $ git push origin master
   $ git push --tags origin master
   ```

7. Make sure to merge back into the `develop` branch as well:

```
$ git checkout develop
$ git merge release-1.2.0
$ git push origin develop
```

8. You can then remove the `release` branch from your local repository:

```
$ git branch -d release-1.2.0
```

Version Numbers

Each release will be tagged with a version number, using the MAJOR.MINOR.PATCH Semantic Versioning format and specifications.

These version numbers indicate the changes to the public *API*.

The PATCH number will be incremented if a new version contains only backwards-compatible bug fixes.

The MINOR number is incremented for new, backwards-compatible functionality and marking any new deprecations. Increasing the MINOR number should reset the PATCH number to 0.

The MAJOR number is incremented if ANY backwards incompatible changes are introduced to the public *API*. Increasing the MAJOR number should reset the MINOR and PATCH numbers to 0.

Pre-release versions may have additional data appended to the version, e.g. `1.0.1-alpha` or `2.1.0-rc`.

The first stable release will begin at version 1.0.0, any versions before this are for initial development and should be not be considered stable.

For more information, please review the Semantic Versioning Specification.

Tests

AcornAccounting is developed using Test-Driven Development, meaning tests are written **before** any application code.

Features should be written incrementally alongside tests that define the feature's requirements.

When fixing bugs, a test proving the bug's existence should first be written. This test should fail initially and pass when the fix is implemented. This ensures that the bug does not reappear in future versions.

All tests must pass before any branch is merged into the public branches `master` and `develop`.

Our goal is to achieve 100% test coverage. Any code that does not have tests written for it should be considered bugged.

Test coverage will be monitored, and no commits that reduce the Test Coverage will be merged into the main branches. The pytest and coverage packages are recommended for monitoring test coverage. These packages are included in the `test` requirements file, which can be installed by running:

```
$ pip install -r requirements/test.txt
```

You can then check a branch's Test Coverage by running:

```
$ cd acornaccunting
$ py.test
```

To clear the coverage history, remove the `.coverage` file:

```
$ rm .coverage
$ py.test
```

You can generate an html report of the coverage by adding the `--cov-report html` flag:

```
$ py.test --cov-report html
```

You can also specify a single file to run:

```
$ py.test accounts/tests.py
```

You can re-run tests when files have changed:

```
$ ptw
$ ptw accounts
$ ptw entries
```

Documentation

Documentation for AcornAccounting is written in reStructuredText and created using the Sphinx Documentation Generator. Sphinx's `autodoc` module is used to create the API specifications of the application by scraping docstrings(**PEP 257**).

Each class, function, method and global should have an accurate docstring for Sphinx to use.

Each feature or bug fix should include all applicable documentation changes such as changes in *Screen Descriptions* or the *API*.

To build the Documentation, install the prerequisites then run the make command to generate either html or pdf output:

```
$ pip install -r requirements/local.txt
$ cd docs/
$ make html; make latexpdf
```

The output files will be located in the `docs/build` directory.

Specifications

Technical Specifications and Documentation should exist in the docstrings(**PEP 257**) of the respective class, method, function, etc.

Design Specifications will be written for every usecase and screen in the application. Wireframes will be created for each screen. And each screen's Entry, Initial, Intermediate and Final Conditions should be clearly defined(see *Screens and Conditions*).

For Usecases and Complex Screens, UML models such as Activity Diagrams will be created using plantUML and Sphinx.

Design Specifications written for new features should include all the *Screen Conditions*.

CHAPTER 5

Specification Explanation

Use Cases

Use Cases describe interactions with the application by multiple roles of users(accountants, communards, administrators, budgeters, etc.). They are also known as "User Stories".

In general, they describe "who uses this site" and "why?".

Use Cases include diagrams and short narratives. Each Use Case will direct the reader to any relevant Screens.

Use Case Diagrams

Diagrams will be used to display the use case associated with each role. Each role is represented by a figure, known as an Actor. Use Cases are enclosed in circles and similar Use Cases are grouped together by rectangles.

Lines between an Actor and a Use Case indicates the actor utilizes the Use Case. A hollow arrow between two actors means that the actor is based off of the actor the arrow is pointed at, and inherits the other actors Use Cases. A dashed arrow from one Use Case to another indicates the source Use Case depends on or includes the Use Case being pointed at.

The following example shows the use cases for an application where an `admin` creates Accounts and both `admin` and `viewer` view them:

Screens and Conditions

A `screen` is a specific page in the application.

Every screen in the specification contains a description of the following Conditions:

- Entry Conditions - The action required to get to the screen
- Initial Conditions - The initial appearance and state of the screen
- Intermediate Conditions - All possible changes in the screen before the screen is replaced
- Final Conditions - All possible exits from the page and any related behind-the-scenes actions

Each screen will contain a *wireframe*, depicting the page's layout. The wireframe is not meant to depict the final appearance of a page, but acts as a rough guide or prototype.

Activity Diagrams

Complex Screens may include flowcharts or activity diagrams. To read these, start at the top circle, following the arrows until you reach diamonds, which represent decisions. Continue down only one path from a decision point. Repeat until you reach the end, represented by the bottom circle.

CHAPTER 6

Design Specifications

Use Cases

Communards

Communards will use the application to enter Trip and Credit Card purchases, and check balances for Accounts such as their Stipend, Deposited Asssets and any related Budgeted Projects.

Communards will take out Cash Advances to make Community and Personal Purchases in town, known as Trips. Communards will enter their Trips through the *Trip Entry Form*. Trips allow Communards to associate Purchases with their relevant Accounts.

Communards with Personal Credit Cards will use the application enter their Purchases and associate Purchases with an Account. This will be done with the help of the *Credit Card Entry Form*.

A Communard may want to check their Stipend or Deposited Assets Balances, or review the Purchases and Balance of any Projects they manage. They may check the current Balance of any Account via the *Chart of Accounts* and all Debits/Credits to a specific Account via the *Account's Detail Page*.

Accountants

Accountants are Communards but with more responsibilities. They will use the application like Communards, but will also be involved in additional Entry and Administration Tasks.

Accountants will use the *Add Entry Pages* to enter new Forms and Statements such as Bank and Credit Card Statements or Internal Transfers. Accountants are responsible for approving Credit Card and Trip Entries submitted by Communards.

Accountants will also reconcile an Account's balance against a Statement's balance using the *Reconcile Account Page*.

Occasionally they will change an Account, Entry or Event's details and create new Accounts, Headers and Events. This can be as simple as fixing a spelling error or as destructive as deleting an Account. Accounts and Events are editable through their respective *Admin Pages*. Entries are created and edited through their respective *Add Entry Page*.

Once a Year, Accountants will start a new Fiscal Year. Fiscal Years allow Accountants to archive a Year's data, removing old entries and resetting Account balances in order to track spending and income on a yearly basis. Fiscal Year creation will be handled by the *Add Fiscal Year Page*.

Budgeter

Budgeters are Communards who are also responsible for analyzing and planning spending.

Budgeters may access the Profit & Loss for a specific date range in the current year through the *Profit & Loss Reports* or the Trial Balance through the *Trial Balance Report*. They may reference the historical Balances for Asset, Liability and Equity Accounts and the historical Profit & Loss amounts for Income and Expense Accounts via the *Account History Page*.

Budgeters can view an overview of all Events via the *Event Reports Page* which shows information such as each Event's location and Net Profit.

Screen Descriptions

AcornAccounting is web-based and each distinct web page is considered a screen. Each screen is generated by the Django back-end framework and will use the Twitter Bootstrap front-end framework.

The following *wireframes* may not be rendered/styled with the front-end elements and are meant to convey the layout of the application, not it's final appearance or style.

The Wireframes are built using Pencil and the Pencil Document is `available for download`.

Global Design

The following design features will be implemented consistently throughout the application:

- All Date fields will require the input to be in some combination of `MM/DD/YYYY` or `M/D/YY` format, such as `01/2/13` or `3/20/12`. Clicking a Date field will pop-up a calendar, allowing the User to select a date, instead of having to enter one.
- Table rows will change their font/background colors to a unique set when moused over.
- Negative balances and amounts will be surrounded by parenthesis, `($22.00)`, instead of having minus signs, `-$22.00`.

General Layout

The general layout of the website will follow the basic Header-Content-Footer model:

- Header - Company Logo/Title, Navigation
- Content - Each screens unique content
- Footer - Copyright, Additional Links(feedback, bug report, documentation)

There will be no global sidebars.

Navigation

The Navigation will be placed in the `Header` section of the *General Layout*.

The navigation will contain multiple dropdown-menus and autocomplete inputs:

- A Logo and/or Company Name(hyperlinked to the *Home Page*)
- An *Accounts Sub-menu*
- An *Entry Sub-menu*
- A *Reports Sub-menu*
- An *Accounts Autocomplete Input*

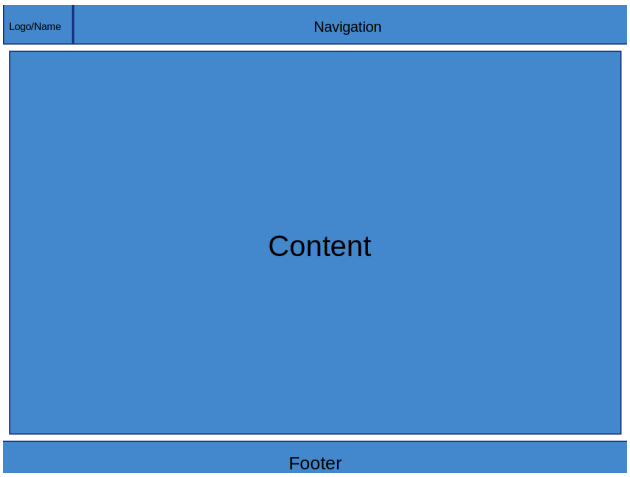

Figure 6.1: The general layout of the application.

- A *Bank Journals Autocomplete Input*
- A *Admin Sub-menu*

Figure 6.2: The Base Navigation Menu.

Accounts Sub-Menu

The `Accounts` Sub-Menu will contain the following Items and Links:

- *Chart of Accounts*
- *General Ledger*
- *Budgeted Accounts*
- Member Stipends - Linking to the `Member's Stipends` *Chart of Accounts Page*
- Member Deposited Assets - Linking to the `Member's Deposited Assets` *Chart of Accounts Page*
- *Account History Page*

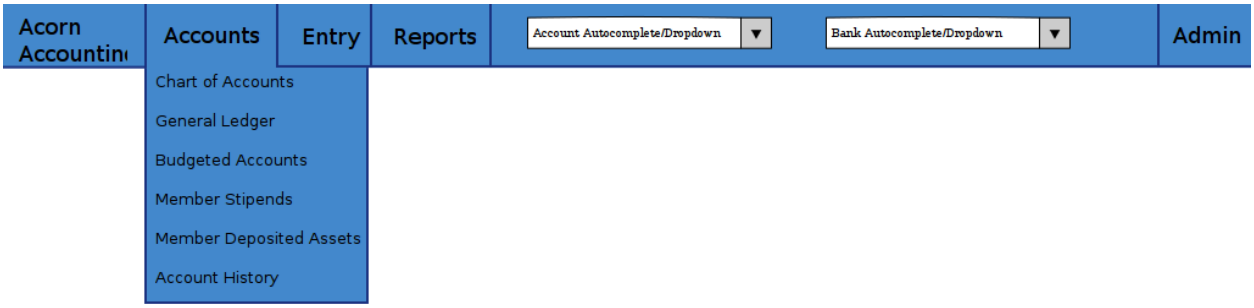

Figure 6.3: The Accounts Sub-Menu.

Entry Sub-Menu

The `Entry` Sub-Menu will contain the following Items and Links:

- *General Entry*
- *Transfer*
- *Bank Spending*
- *Bank Receiving*

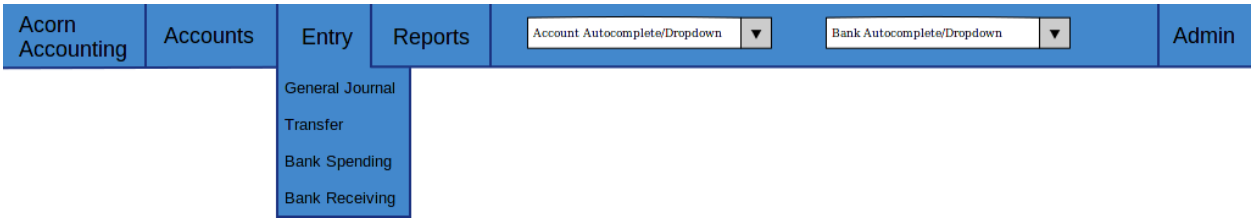

Figure 6.4: The Entry Sub-Menu.

Reports Sub-Menu

The `Reports` Sub-Menu will contain the following Items and Links:

- *Events*
- *Profit & Loss*
- *Trial Balance*

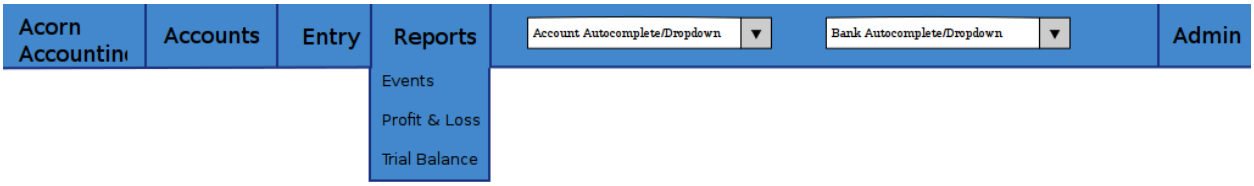

Figure 6.5: The Reports Sub-Menu.

Admin Sub-Menu

The `Admin` Sub-Menu will contain the following Items and Links:

- *Add Header*
- *Add Account*
- *Add Event*
- *New Fiscal Year*

Autocomplete Inputs

An `Autocomplete` Input Widget allows users to enter filtering text and select items from a dropdown box. The Inputs will be pre-populated will all respective items. When the user enters text, a dropdown will appear with suggested

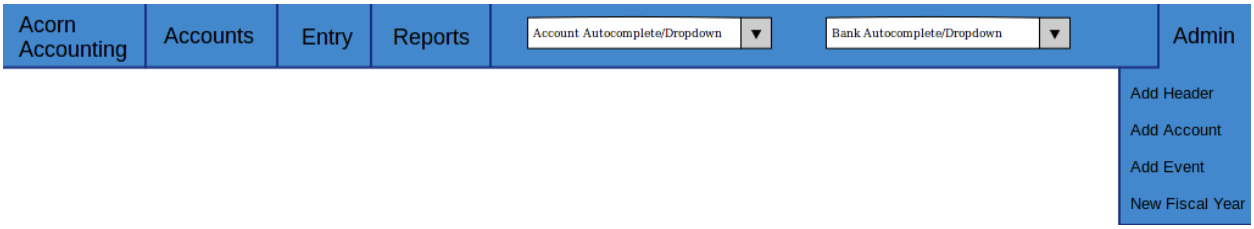

Figure 6.6: The Admin Sub-Menu.

completions that the user can select. The user can also click the Input Widget, causing it to display a selectable list of all items.

Upon item selection, the application will redirect the user to a specific page:

- The item's *Account Detail Page* for the `Accounts` Autocomplete Input
- The item's *Bank Journal Page* for the `Bank Journal` Autocomplete Input

For a working example of this functionality, see the Select2 Widget.

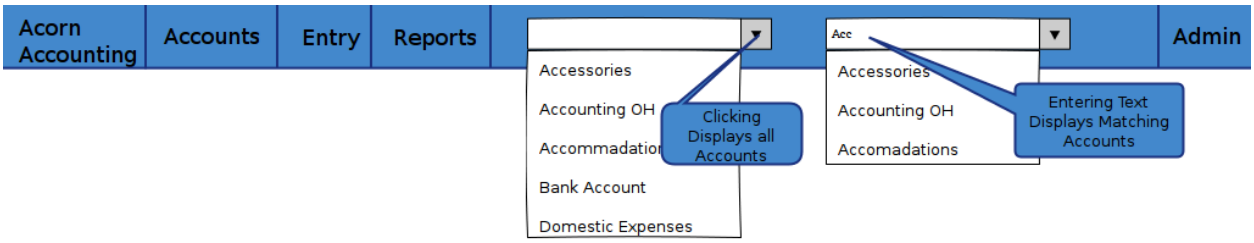

Figure 6.7: An example of the Autocomplete Input Widgets.

Home Page

The Home Page will be the *Chart of Accounts Page*.

In the future, the content may change to show recent activity and priority items(see *Alternate Home Page*).

Entry Conditions

The Home Page will be accessible through the application's base URL(e.g. www.acornaccounting.com/). The *Navigation's* Company Title and Logo will be a hyperlink, directing the User to the Home Page.

Admin Pages

The Application will initially use the default Django admin app for generating administration pages. The admin app will be used for Creating and Editing Headers, Accounts and Events. Eventually custom administration pages will be made for Creating each item, while Edit links will be placed in the *Account Detail Page* and *Event Detail Page*.

Entry Conditions

The admin back-end will be accessible by clicking on the *Admin Sub-Menu*, specific pages will be accessible by clicking on the respective items in the Sub-Menu.

Initial Conditions

If the User is logged in, a table will be displayed, showing links to the Accounts, Headers and Events sub-pages.

Otherwise, a login form containing fields for a Username and Password will be displayed.

Final Conditions

If the User is not logged in, submitting the form will validate the User's Login Credentials, redirecting to the main Admin Page if valid.

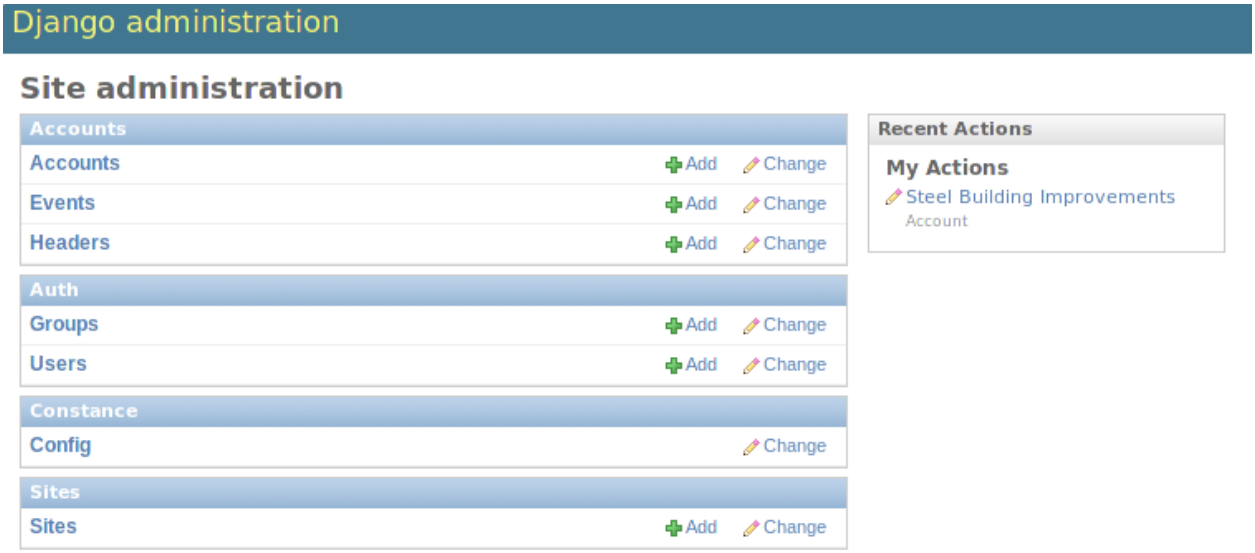

Figure 6.8: The Main Administration Page, created by Django's Admin App.

Headers Admin

Accounts Admin

Events Admin

Chart of Accounts Page

The Chart of Accounts Page displays all Headers and Accounts of the Company, and their respective Balances.

Case 1 describes accessing the global Chart of Accounts. *Case 2* describes accessing the Chart of Accounts recursively, by clicking on a Header row in the Chart of Accounts.

Entry Conditions

Case 1

This screen is accessible through the `Chart of Accounts` link in the *Accounts Sub-Menu*.

It is currently the *Home Page* and therefore clicking the Company Name/Logo to the left of the *Navigation* will redirect the user to this screen.

Case 2

The screen is also recursively accessible, clicking on a Header row in this screen will send the User to a Chart of Accounts Page for that specific Header.

Initial Conditions

Case 1

The page will display a series of tabs above a single table. The Tabs will be named by the different Account Types(Assets, Liability, Equity, etc.) and the currently selected tab will be differentiated by color. By default the table will contain all Headers and Accounts that are Assets and the Asset Tab will be selected.

The Table will have headings for Account Numbers, Names, Descriptions and Current Balances. The Headers' font and background colors will be different from Accounts'. Child Headers and Accounts will have their names indented one level higher than their Parent Header, like in the following example:

```
I am the Highest Header
    I am its child Account
    I am a sub-Header
        I am the sub-Header's child Account
```

Mousing over a row should highlight that row in a unique color, and removing the mouse from a row should revert the effect.

Case 2

If the screen is accessed recursively by a Header row hyperlink, the screen will emulate *Case 1* except the tabs will be replaced with breadcrumb links up the Header's ancestor tree. Instead of showing all Headers and Accounts, only the children of the selected Header will be displayed.

Intermediate Conditions

Case 1

Clicking an Account Type Tab will replace the table, Headers, and Accounts currently displayed with a table containing all Headers and Accounts of the newly selected Account Type.

Final Conditions

Each row of the table will be a hyperlink. Header rows will link back to the Chart of Accounts Page, using the selected Header for *Case 2*. Account rows will link to the Account's *Account Detail Page*.

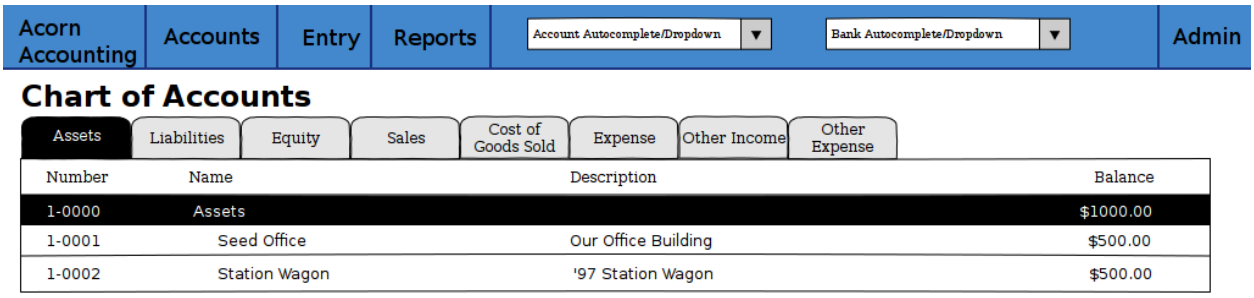

Figure 6.9: A Sample Chart of Accounts Page(*Case 1*).

Account Detail Page

The Account Detail Page will be used to display Transactions associated with a specific Account within a User-specified date range.

Entry Conditions

This screen is accessible by clicking on an Account row on the *Chart of Accounts* or the *General Ledger*, and by selecting an Account from the *Account Autocomplete Input*.

Initial Conditions

The page will contain a Date Range form, a Heading and information, several buttons for administration and a table for displaying the Account's Transactions.

The Date Range form will have 2 Text Input Widgets, labeled `Start Date:` and `End Date:`. They will be used to control the date range of Transactions shown. The `Start Date` will default to the first day of the current month while the `End Date` will default to the current date.

The Heading will display the Account's Number, Name and Current Balance. Additional information should include the statement date of the last reconciliation, only if the Account was previously reconciled.

There will be 2 buttons for Accountants/Administrators, `Edit` for changing the Account's details(name, parent, balance, etc.) and `Reconcile` for reconciling the Account against Bank Statements. If the Account is a Bank Account, there should also be a button linking to it's Bank Journal.

The table will be populated with Transactions within the date range. The table will have headings for the Entry Number, Date, Memo, Transaction Detail, Debit/Credit, Event and Account Balance. The Account Balance will only be shown if the date range is within the current Fiscal Year.

The table will contain a footer that displays the total amounts of Debits and Credits and the Net Change(Credits - Debits) for the date range. If the date range is within the current Fiscal Year, a `Start Balance` and `End Balance` will also be shown.

Final Conditions

The `Edit` Hyperlink will direct the User to the Account's *Edit Page*. The `Reconcile` Hyperlink will direct the User to the Account's *Reconcile Page*.

Clicking on a row in the table will direct the user to the *Entry Detail Page* for the specific Transaction. If an item in the `Event` column is clicked, the User will be directed to the *Event Detail Page* for the Event.

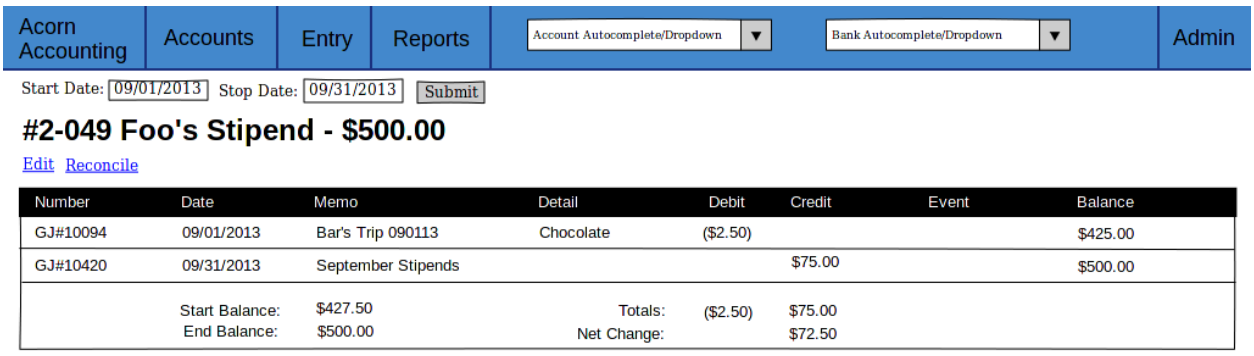

Figure 6.10: A Sample Account Detail Page.

Event Detail Page

The Event Detail Page will be used to display all Transactions associated with a specific event.

Entry Conditions

The Event Detail page may be reached from the *Events Report Page* by clicking a row in the table or from the *General Ledger Page*, *Bank Journal Page* or the *Account Detail Page* by clicking on an Event in the Page's table's `Event` column.

Initial Conditions

The page will contain a Title Heading, Sub-Heading and Table.

The Title Heading will be composed of the Event Name and Number, the Sub-Heading will contain the City, State and Date of the Event.

The table will be populated with Transactions that are associated with the Event. It should have headings for the Transaction's Entry Number, Date, Account, Memo, the Transaction's Detail, and Credit/Debit Amount.

The table will contain a footer that displays the total amounts of Debits and Credits and the Net Change for the Event.

Final Conditions

Clicking a row in the table should direct the User to the *Entry Detail Page* for the selected Transaction/Entry. However, if a row is clicked in the `Account` column the User will be directed to the *Account Detail Page* for the selected Transaction.

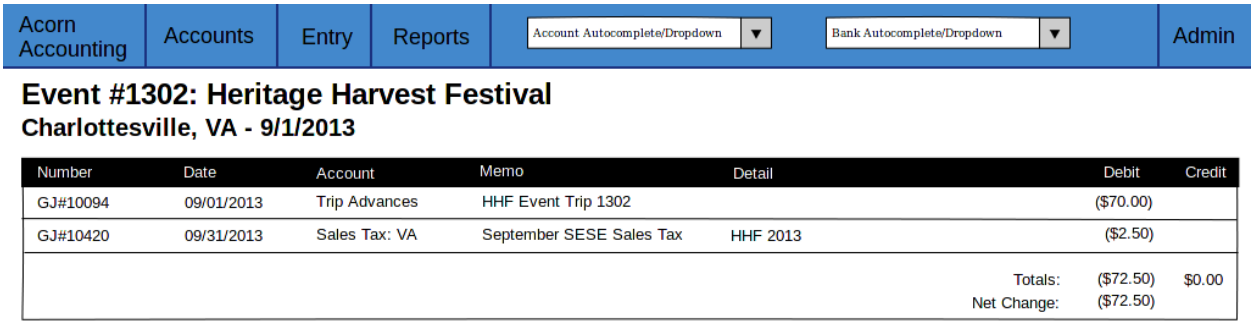

Figure 6.11: A Sample Event Detail Page.

Account History Page

The Account History page is used to show purged data from previous Fiscal Years. The page will show either the Account Balance or Net Change for each month in previous Fiscal Years, depending on the type of the Account.

Note: Historical Accounts are only generated after creating a second Fiscal Year. If Fiscal Years are not used or only one Fiscal Year has been created, there will be no Account History pages.

Entry Conditions

This screen is accessible via the `Account History` link in the *Accounts Sub-Menu*. The default month displayed is the current month in the previous year. Other months may be accessed by `Next` and `Previous` buttons on this screen.

Initial Conditions

The screen contains a Title Heading, buttons for month navigation and tabs above a table.

The Title Heading will contain `Account History` and the month/year being displayed, in `MM/YYYY` format.

There will be `Previous` and `Next` buttons, allowing the User to view the Account History for a different month.

There will be 2 tabs, one labeled `Balances` and one labeled `Profit & Loss`. The currently selected tab should be differentiated by color. By default, the `Balances` tab will be selected. The `Balances` tab will contain Accounts that are Assets, Liabilities and Equities. The `amount` column for these Accounts will represent the month-end balances. The `Profit & Loss` tab will contain all other Account Types(Expenses and Incomes), their `amount` column represents the Net Change in value for the month.

The table will display the Historical Accounts for the selected month and year. The table will have headings for Account Numbers, Names and Amounts.

Intermediate Conditions

Clicking on a tab will toggle which Historical Accounts are shown in the table.

Final Conditions

Clicking on the `Next` or `Previous` buttons will direct the User to the next or previous month's Account History.

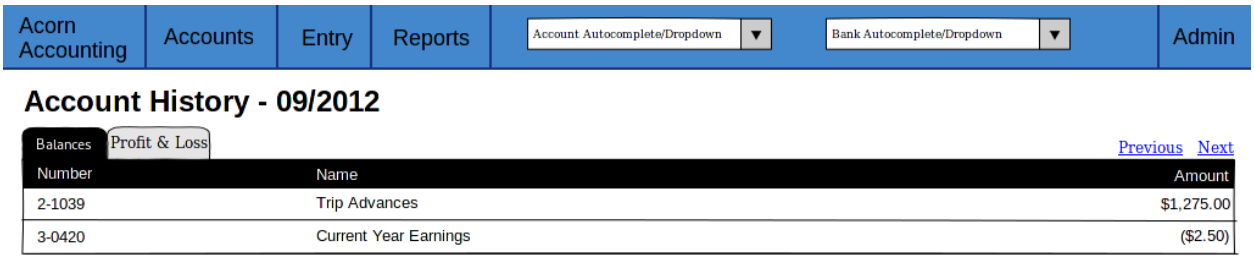

Figure 6.12: A Sample Account History Page showing the Balances of the 2 Asset/Liability/Equity Accounts that existed in the 2012 Fiscal Year. Clicking the Profit & Loss tab would display the Net Changes of all other Account types.

Journal Entry Detail Pages

The Journal Entry Detail pages will be responsible for display information about a specific Journal Entry and it's associated Transactions.

There are three types of Journal Entries:

- *Bank Receiving Entry* - Entries that put money into a Bank Account (checks, *ACH* payments, cash deposits, etc.)
- *Bank Spending Entry* - Entries that take money out of a Bank Account (checks, cash withdrawals, etc.)
- *General Journal Entry* - Entries not related to Bank Accounts (internal transfers, cash drawer withdrawals, etc.)

General Journal Entry

Entry Conditions

The General Entry page will be accessible by clicking an Entry on the *General Ledger Page*, or by clicking a Transaction on the *Account Detail Page* or *Event Detail Page*.

After submitting valid data on the *Add Journal Entry Page*, the User will be redirected to this page if they do not choose to add more Entries.

Initial Conditions

The page will contain Headings, administrative information/buttons, and a table for displaying the Journal Entry's Transactions.

The Heading will display the Entry's date, memo and number.

A `Created` and `Updated` (if applicable) Date should be shown, along with a link for Accountants to Edit the Entry.

The Table will be populated with the Entry's Transactions. It will have headings for the Transaction's Account, Detail, Debit/Credit Amount and Event. The table will have a footer showing the total amount of Credits and Debits.

Final Conditions

The `Edit` Hyperlink will direct the User to the *Add Journal Entry Page* for the Entry(*Case 2*).

Clicking a row in the Transaction table will direct the User to the *Account Detail Page* for the Transaction's Account.

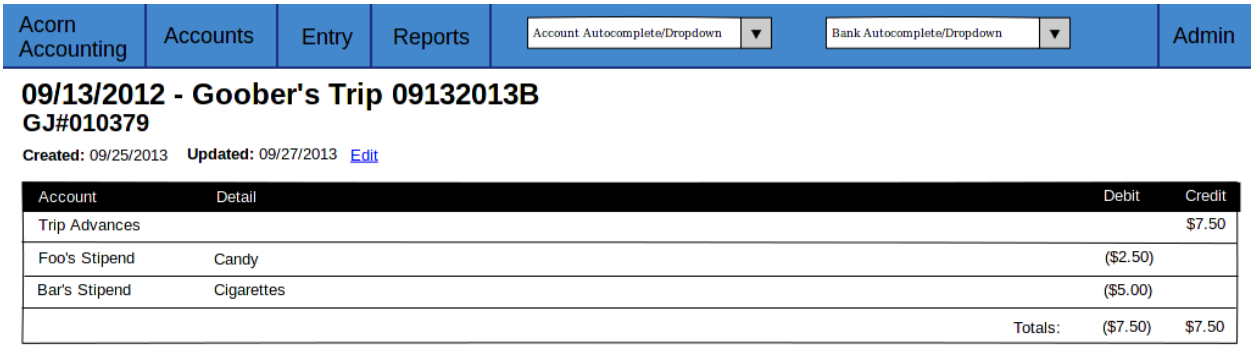

Figure 6.13: Sample Journal Entry Detail Page.

Bank Receiving Entry

Entry Conditions

The Bank Receiving Entry page will be accessible by clicking an Entry on the *Bank Journal Page* or by clicking a Transaction on the *Account Detail Page* or *Event Detail Page*.

The User is redirected to this Page after submitting valid data on the *Add Bank Receiving Entry Page*.

Initial Conditions

The page will contain Headings, administrative information/buttons, and a table for displaying the Bank Receiving Entry's Transactions.

The Headings will display the Entry's date, bank account, memo, entry number, payor and debit amount.

A `Created` and `Updated` (if applicable) Date should be shown, along with a link for Accountants to Edit the Entry.

The Table will be populated with the Bank Receiving Entry's Transactions. It will have headings for the Transaction's Account, Detail, Credit Amount and Event.

Final Conditions

The `Edit` Hyperlink will direct the User to the *Add Bank Receiving Entry Page* for the Entry(*Case 2*).

Clicking a row in the Transaction table will direct the User to the *Account Detail Page* for the Transaction's Account.

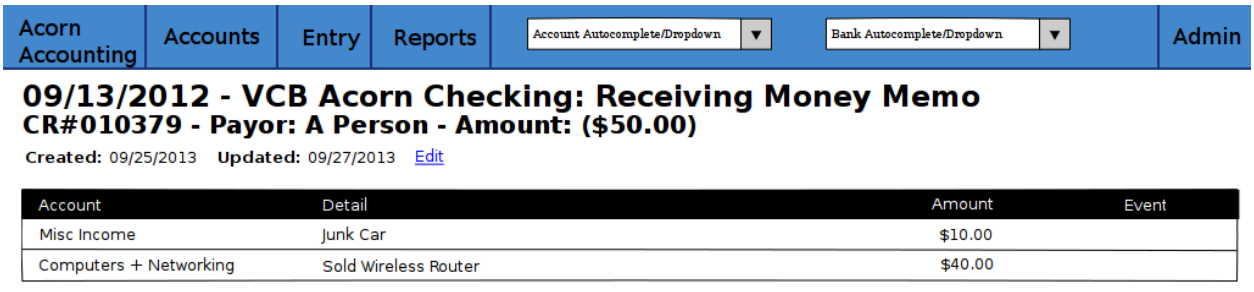

Figure 6.14: Sample Bank Receiving Entry Detail Page.

Bank Spending Entry

Entry Conditions

The Bank Spending Entry page will be accessible by clicking an Entry on the *Bank Journal Page* or by clicking a Transaction on the *Account Detail Page* or *Event Detail Page*.

Initial Conditions

The page will contain Headings, administrative information/buttons, and a table for displaying the Bank Spending Entry's Transactions.

The Headings will display the Entry's Date, Bank Account, Memo, Number, Payee and Credit Amount.

Note: The Entry Number will be `CD# <check_number>` for entries with Check Numbers and `CD###ACH###` for entries that are *ACH* Payments.

A `Created` and `Updated` (if applicable) Date should be shown, along with a link for Accountants to Edit the Entry.

The Table will be populated with the Bank Receiving Entry's Transactions. It will have headings for the Transaction's Account, Detail, Debit Amount and Event.

Final Conditions

The `Edit` Hyperlink will direct the User to the *Add Bank Spending Entry Page* for the Entry(*Case 2*).

Clicking a row in the Transaction table will direct the User to the *Account Detail Page* for the Transaction's Account.

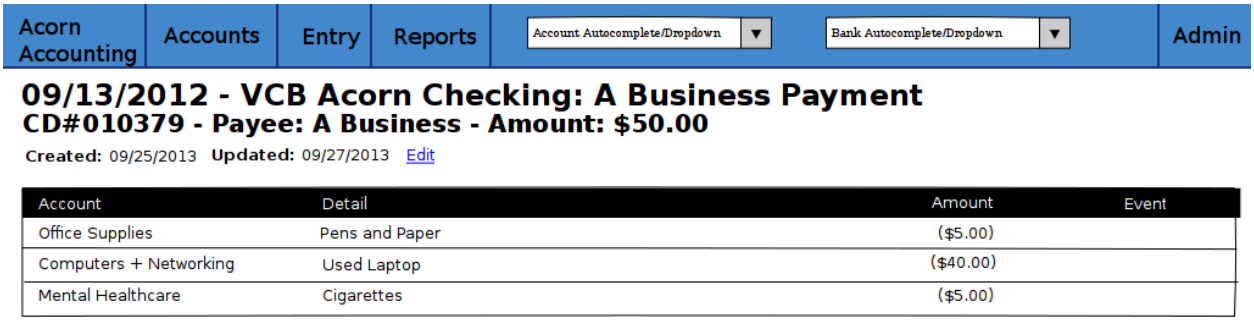

Figure 6.15: Sample Bank Spending Entry Detail Page for an Entry with a `Check Number`.

General Ledger Page

The General Ledger page will be used to display all General Journal Entries and their associated Transactions within a user-specified date range.

Entry Conditions

This screen is only accessible through the `General Ledger` item in the *Navigation's Accounts Sub-Menu*.

Initial Conditions

The page will contain a Date Range form, a Heading, and a table for displaying General Journal Entries within the Date Range.

The Date Range form will have 2 Text Input Widgets, labeled `Start Date:` and `End Date:`. They will be used to control the date range of Journal Entries shown. The `Start Date` will default to the first day of the current month while the `End Date` will default to the current date.

The Heading will display `General Ledger` and the current date range.

The table will be populated with General Journal Entries within the date range. The table will display each Journal Entry's Number, Date and Memo, along with the Accounts, Details, Credits/Debits and Events of the Journal Entry's Transactions.

Final Conditions

Submitting a valid `Start` and `End` date will cause the page to reload, filling the table with General Journal Entries from the submitted date range.

Clicking on a Journal Entry row in the table will direct the user to the *Detail Page* for the selected Journal Entry. Clicking on an Account row in the table will direct the user to the *Account Detail Page* for the selected Account. Clicking on an Event in the table will direct the user to that Event's *Detail Page*.

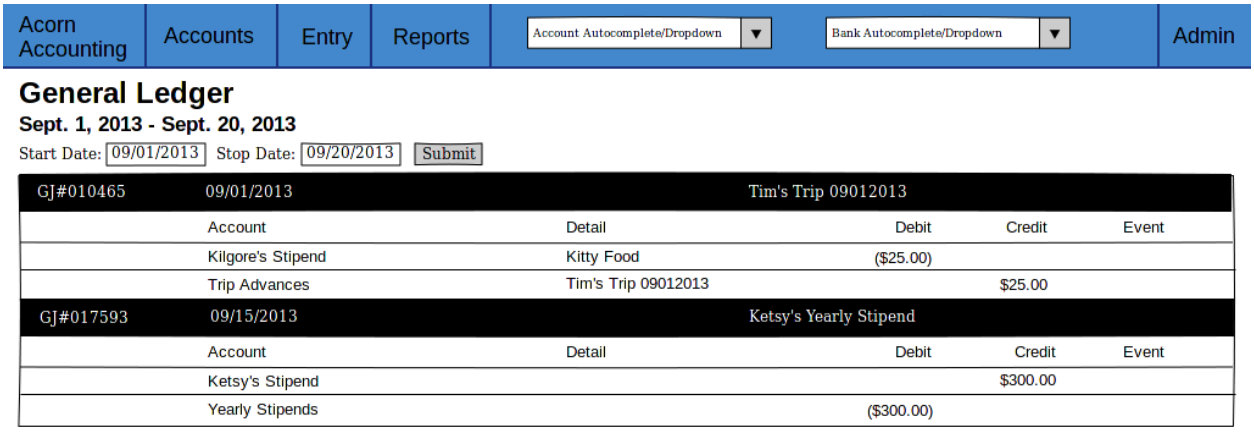

Figure 6.16: A Sample General Ledger Page, showing two General Journal Entries.

Bank Journal Page

The Bank Journal page will be used to display a Bank Account's Bank Spending and Bank Receiving Entries within a user-specified date range.

Note: Accounts are designated as Bank Accounts in the *Account Admin Page*.

Entry Conditions

The Bank Journal page will be accessed through the *Bank Journal Autocomplete Input* in the *Navigation Menu*. Submitting a valid Bank Account will direct the user to the Bank Journal page for that Account.

Initial Conditions

The page will contain a Date Range form, a Heading, and a table for displaying Bank Spending and Receiving Entries within the Date Range.

The Date Range form will have 2 Text Input Widgets, labeled `Start Date:` and `End Date:`. They will be used to control the date range of Bank Entries shown. The `Start Date` will default to the first day of the current month while the `End Date` will default to the current date.

The Heading will display the Account's Name, `Journal` and the current date range.

The table will be populated with Bank Spending and Receiving Entries within the date range. The table will display each Bank Entry's Number, Date and Memo, along with the Accounts, Details, Credits/Debits and Events of the Bank Entry's Transactions.

Final Conditions

Submitting a valid `Start` and `End` date will cause the page to reload, filling the table with Bank Entries associated with the Bank Account within the submitted date range.

Clicking on a Bank Spending Entry row will direct the user to its *Spending Detail Page*, a Bank Receiving Entry to its *Receiving Detail Page*. Clicking on an Account row in the table will direct the user to the *Account Detail Page* for the selected Account. Clicking on an Event in the table will direct the user to that Event's *Detail Page*.

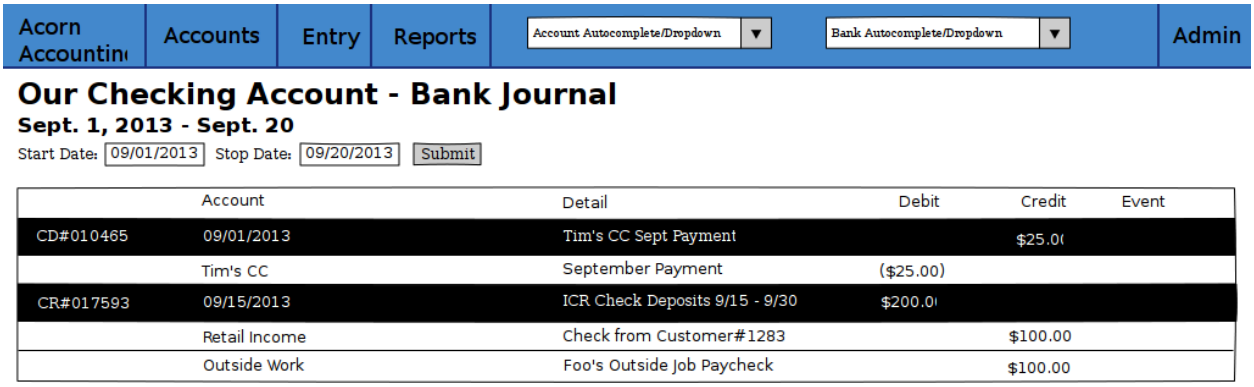

Figure 6.17: A Sample Bank Journal Page, showing both Bank Spending and Bank Receiving Entries.

Events Report Page

The Events Report Page shows an overview of all Events.

Entry Conditions

The page is accessible through the `Events` link in the *Reports Sub-Menu*.

Initial Conditions

The page will contain a Heading and a table for displaying all Events.

The Heading should be `Events Report`.

The table should have headers for each Event's date, name, number, city, state and net change. The net change should be the sum of all credits and debits related to the Event.

Final Conditions

Clicking a row in the Events table will redirect the user to that Event's *Detail Page*.

Profit & Loss Report Page

The Profit & Loss Report shows the Net Profit or Loss over a specified date range.

Entry Conditions

The page is accessible through the `Events` link in the *Reports Sub-Menu*.

Initial Conditions

The page will contain a Heading, a Date Range form and a table containing header totals and summations.

The Heading should be `Profit & Loss Statement` along with the selected Date Range.

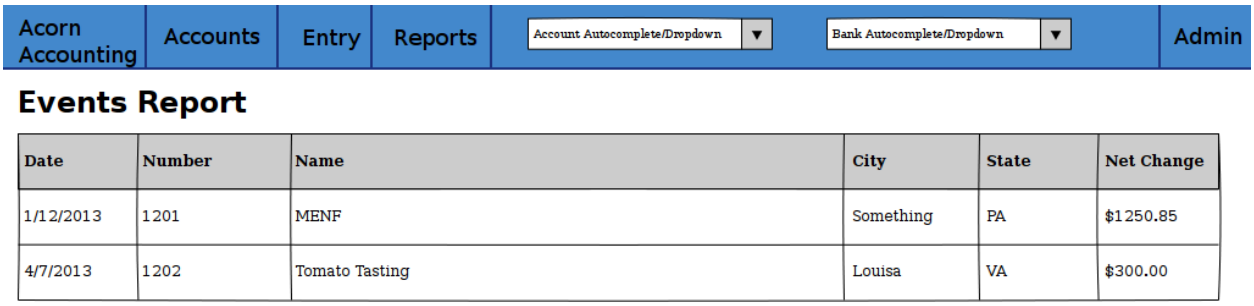

Figure 6.18: A Sample Events Report Page, showing two Events.

The Date Range form will contain two fields, the `Start Date` and `Stop Date`. The default dates should be the first day of the current to the current day.

The table will have no column headings. The table will display header totals and the results of calculations using these totals. There will be a table body for the following headers and calculations:

1. Sales
2. Cost of Goods Sold
3. Gross Profit (Sales - CoGS)
4. Expenses
5. Operating Profit (Gross Profit - Expenses)
6. Other Income
7. Other Expenses
8. Net Profit/(Loss) (Operating Profit + Other Income - Other Expenses)

The table should have 5 columns:

1. Header/Account/Calculation Name
2. Account Credits
3. Account Debits
4. Child Header Totals
5. Root Header/Cacluation Total

Names should be indented by their depth level. Only root Headers, their children and grandchildren should have their totals shown. The general layout will look like:

```
Root Header
  Child Account                        $7.00
  Child Header
    Some Account            ($12.00)
    Grandchild Header                             2.00
      Another Account                    $2.00
  Child Header Total                             ($10.00)
Root Header Total                                            ($3.00)

Some Counter                                                 ($3.00)
```

Final Conditions

Submitting a valid Date Range form will reload the page using the new start and stop dates for the table calculations.

Clicking a row in the table will redirect the user to that Account's *Detail Page*.

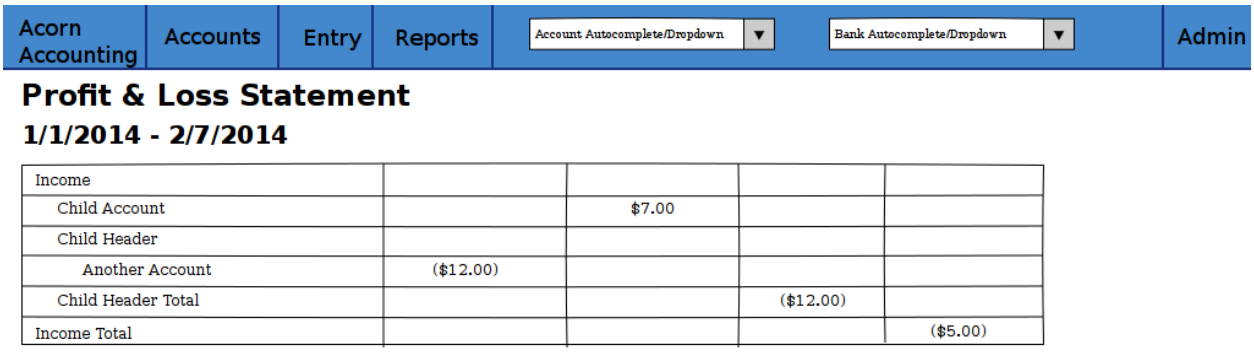

Figure 6.19: A Sample Profit & Loss Report Page, showing just the Income Header.

Trial Balance Report Page

The Trial Balance Report shows the net change and balances of all Accounts over a specified date range.

Entry Conditions

The page is accessible through the `Trial Balance` link in the *Reports Sub-Menu*.

Initial Conditions

The page will contain a Heading, a Date Range form and a table of every Account.

The heading should be `Trial Balance Report` along with the start and stop dates of the period currently being displayed.

The Date Range form will contain two fields, the `Start Date` and `Stop Date`. The default dates should be the first day of the current to the current day.

The table should have headings for each Account's number, name, balance at the beginning of the period, total debits, total credits, net change and the ending balance.

Final Conditions

Submitting a valid Date Range form will reload the page using the new start and stop dates for the table calculations.

Clicking a row in the table will redirect the user to that Account's *Detail Page*.

Reconcile Account Page

The Reconcile Account page will be used to reconcile an Account's Balance and Transactions against a statement.

Case 1 describes the initial Reconcile screen. *Case 2* describes this screen after a Statement Balance and Date are entered.

Entry Conditions

Case 1

The Reconcile Account screen is accessible through the `Reconcile` link on the *Account Detail Page*.

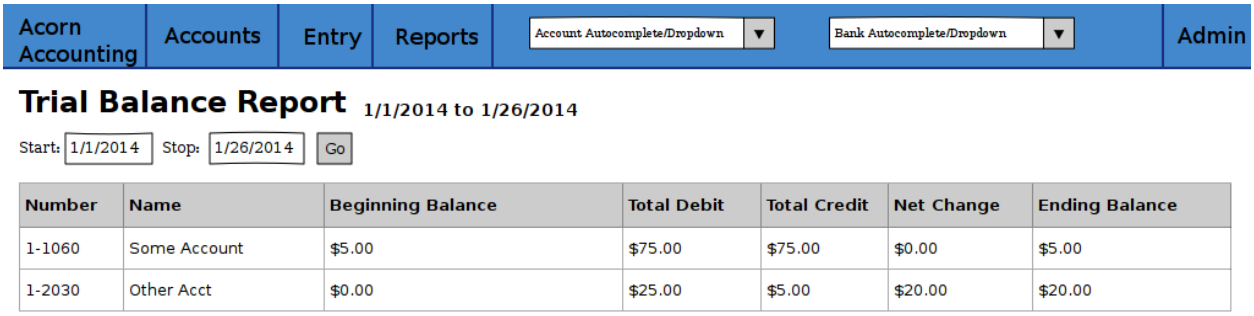

Figure 6.20: A Sample Trial Balance Report Page.

Case 2

Submitting a valid `Statement Balance` and `Statement Date` on the Account Reconcile *Case 1* screen will direct the User to the *Case 2* screen.

Initial Conditions

Case 1

The screen will contain a Heading and a form to retrieve Transactions.

The heading will be `Reconcile` followed by the Account's Name.

The form will have inputs for the Statement Date and Statement Balance. If the Account has previously been reconciled, the previous Statement Date and Statement Balance will also be shown(labeled as the `Last Reconciled Date` and `Reconciled Balance`).

The Statement Date will default to the current date. The Statement Balance will default to the Reconciled Balance.

The form will be submit by a button labeled `Get Transactions`.

Case 2

This screen will contain all elements of *Case 1* with an additional table, displaying all unreconciled Transactions associated with the Account that are dated before the submitted `Statement Date`.

The table will contain checkboxes to toggle whether a Transaction is to be reconciled and each Transaction's information. The table headers will be a checkbox(to toggle all Transactions on or off) and the Transaction's Date, Entry Number, Entry Memo, Debit/Credit Amount, Event.

The Table will have footer that displays the Debit/Credit totals, net change and out of balance amount for all selected Transactions. The out of balance amount is determined by adding the last reconciled balance to the statement balance and subtracting by the net change. It represents the Transaction Credit/Debit total required to make the Reconciled Balance equal to the Statement Balance.

The `Get Transactions` button will resubmit only the Statement Date and Balance, retrieving Transactions from before the new Statement Date.

A `Reconcile Transactions` button will also be present, underneath the Transaction table. This button will be used for final form submission.

Intermediate Conditions

Case 1

The `Statement Balance` will be required to be a valid number.

Case 2

Toggling the checkbox in the table header will cause all listed Transactions to be selected or unselected. Selecting a Transaction's checkbox will update the Debit/Credit totals, the Net Change and Out of Balance Amounts.

The form will only be considered valid if the out of balance amount is equal to `$0.00`, i.e. if `reconciled balance + net change = statement balance`.

Final Conditions

Case 1

Submitting a valid form by using the `Get Transactions` page will return the Screen for *Case 2*.

Case 2

The `Get Transactions` button will refresh the Screen retrieving different Transactions if the `Statement Date` was changed.

Submitting a valid form using the `Reconcile Transactions` button will change the Account's reconciled balance to the `Statement Balance` and mark each selected Transaction as reconciled. The User will be redirected to the *Account's Detail Page*.

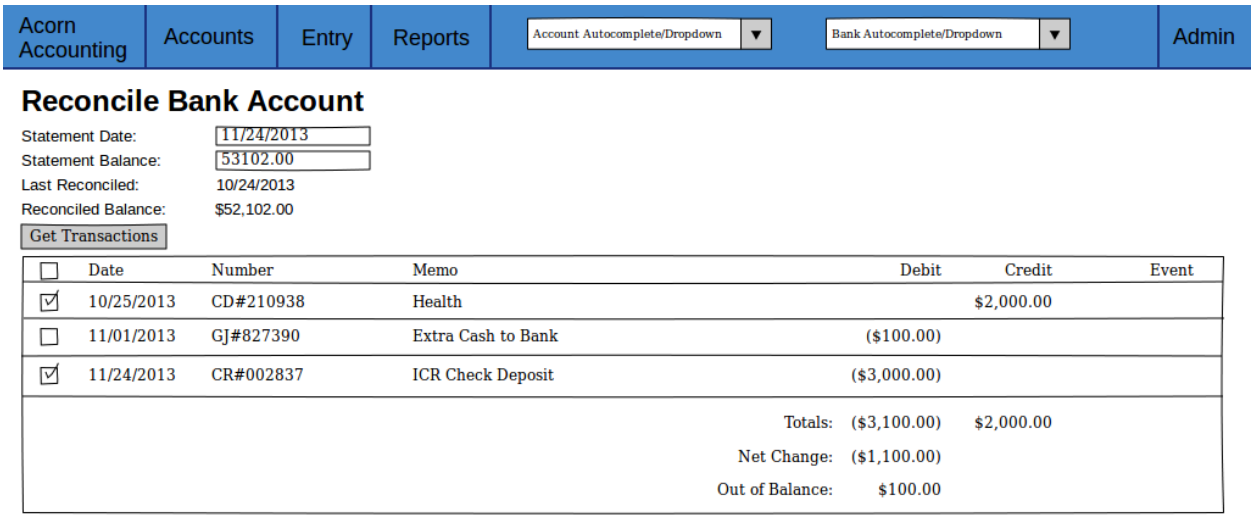

Figure 6.21: A Sample Account Reconciling Page, showing Transactions marked for reconciliation(*Case 2*).

Journal Entry Creation Pages

The Journal Entry Creation Pages will be responsible for creating/editing and validating new Entries and Transactions.

Add Journal Entry

The Add Journal Entry Page will be used to create new General Journal Entries(*Case 1*) and edit existing Entries(*Case 2*).

Entry Conditions

Case 1

This Page is accessible through the `General Journal` link in the *Entry Sub-Menu*.

Case 2

Clicking the `Edit` link in a General Journal Entry's *Detail Page* will direct the User to this Page.

Initial Conditions

The page will contain a Heading and a form to enter Entry details.

The form should contain 2 sections, one for the overall entry details and a table containing each Transaction's details.

The Entry details part of the form will include inputs for the Entry Date, Memo and Additional Comments.

The Transaction table will have headers for each Transaction's Account, Details, Debit/Credit Amount, Event and for deleting the row. There will be a link to add additional rows to the table. The table's footer will show the Debit and Credit totals and the amount the entry is Out of Balance(the Credit total minus the Debit total).

The form will contain two buttons, `Submit` and `Submit & Add More`.

Case 2

All inputs will contain the details of the Journal Entry being edited.

A `Delete` button will be on the Page in addition to the 2 `Submit` buttons.

Intermediate Conditions

The Date and Memo fields, and at least two Transactions, are required.

Only a Debit or Credit may be entered in each row, a Transaction cannot have both.

Changing a Debit or Credit amount will update the Credit/Debit Totals and Out of Balance amounts.

Checking a row's Delete box will remove that row from the table and update the Totals and Out of Balance amounts.

A valid form will have a Debit Total equal to it's Credit Total, resulting in an Out of Balance amount of zero.

Final Conditions

A valid form will create the Journal Entry and the entered Transactions, redirecting the User to the Entry's *Detail Page*.

If the form was submit using the `Submit & Add More` button, the Screen will refresh with a blank form.

Case 2

Submitting the form using the `Delete` button will delete the Entry and its Transactions, redirecting the User to the *General Ledger*.

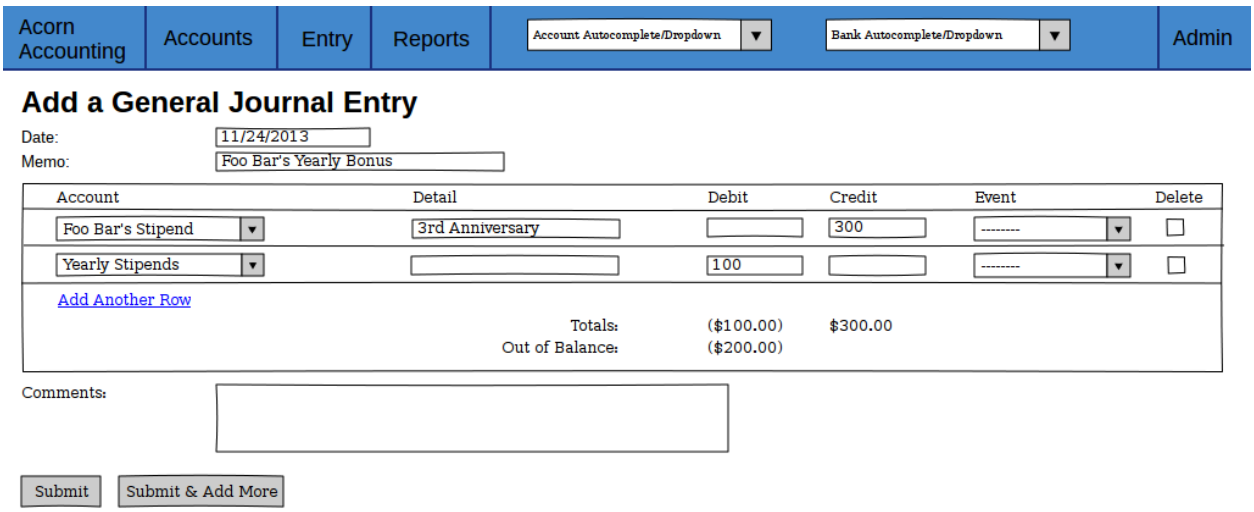

Figure 6.22: A Sample Add Journal Entry Page, showing an Out of Balance entry.

Add Transfer Entry

This Page will be used create Entry's that transfer discrete amounts between two Accounts.

Entry Conditions

This Page is accessible through the `Transfer` link in the *Entry Sub-Menu*.

Initial Conditions

This Page mimics the *Add Journal Entry Page*, except the table contains Source, Destination and Amount columns instead of the Account, Credit and Debit Columns. The table will not contain Total or Out of Balance amounts.

Intermediate Conditions

A source, destination and amount for at least one Transaction is required.

Checking a row's Delete box will remove that row from the table.

Final Conditions

A valid form will create a General Entry and two Transactions for each row in the table, one will debit the Source Account and the other will credit the Destination Account. The User will be redirected to the Entry's *Detail Page*.

If the form was submit using the `Submit & Add More` button, the Screen will refresh with a blank form.

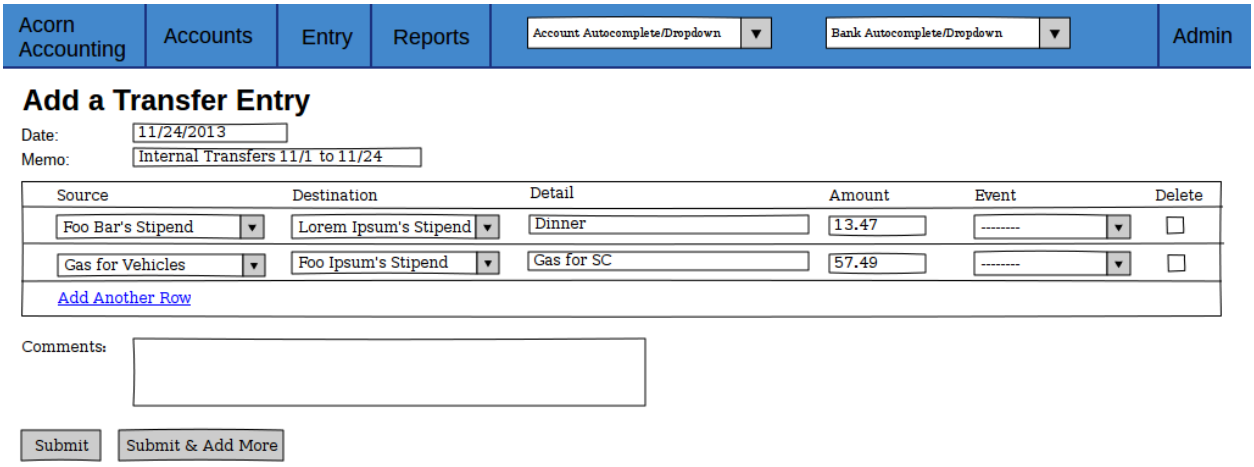

Figure 6.23: A Sample Add Transfer Entry Page, showing 2 rows which will create 4 Transactions.

Add Bank Receiving Entry

This Page is used to enter and update deposits to Bank Accounts.

Entry Conditions

Case 1

This Page is accessible through the `Bank Receiving` link in the *Entry Sub-Menu*.

Case 2

Clicking the `Edit` link in a Bank Receiving Entry's *Detail Page* will direct the User to this Page.

Initial Conditions

This Page mimics the *Add Journal Entry Page*, except for some differences in fields.

The entry fields will also include an Account, Payor and Amount. The Accounts field will be limited to Accounts that are Bank Accounts. The Transaction table will contain a single Amount field instead of Credit and Debit fields.

Case 2

All inputs will contain the details of the Receiving Entry being edited.

A `Delete` button will be on the Page in addition to the 2 `Submit` buttons.

Intermediate Conditions

A Bank Account, Date, Amount, Memo and at least one Transaction is required.

Checking a row's Delete box will remove that row from the table.

Final Conditions

A valid form will create or update the Bank Receiving Entry and the entered Transactions. The User will be redirected to the *Bank Receiving Entry's Detail Page*.

If the form was submit using the `Submit & Add More` button, the Screen will refresh with a blank form.

Case 2

Submitting the form via the `Delete` button will remove the Bank Receiving Entry and it's Transactions, refunding each Account's balance. The User will be redirected to the *Bank Account's Journal*.

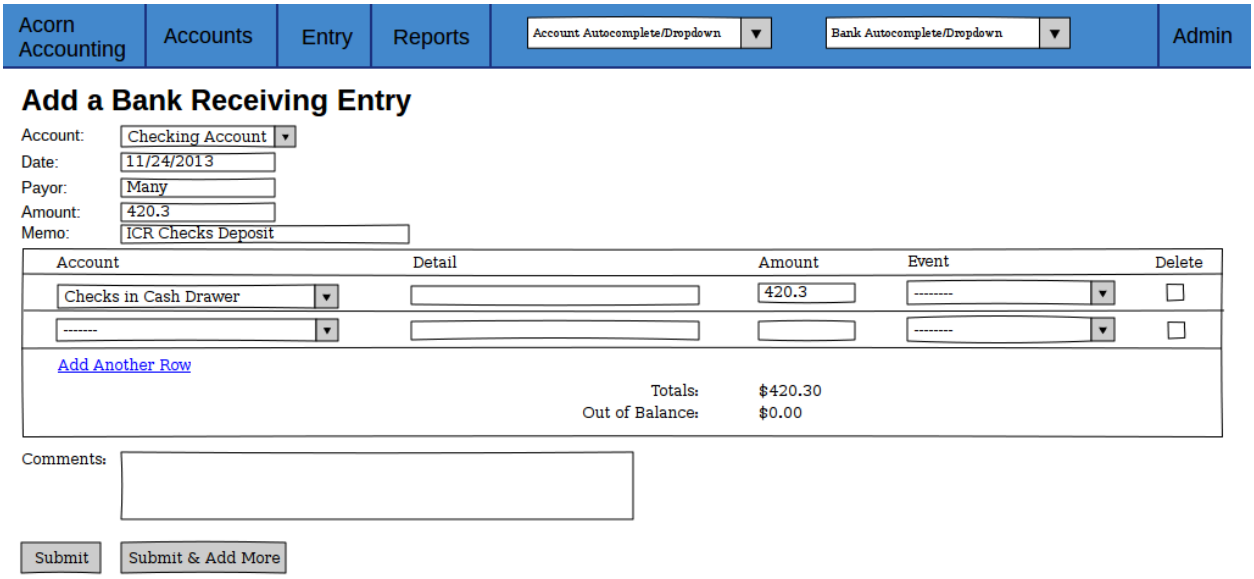

Figure 6.24: A Sample Add Bank Receiving Entry Page, showing a balanced deposit.

Add Bank Spending Entry

This Page is used to enter and update withdrawals from Bank Accounts.

Entry Conditions

Case 1

This Page is accessible through the `Bank Receiving` link in the *Entry Sub-Menu*.

Case 2

Clicking the `Edit` link in a Bank Receiving Entry's *Detail Page* will direct the User to this Page.

Initial Conditions

This Page mimics the *Add Journal Entry Page*, except for some differences in fields.

The entry fields will also include an Account, Payee, *ACH* Payment, Check Number and Amount. The Accounts field will be limited to Accounts that are Bank Accounts. The Transaction table will contain a single Amount field instead of Credit and Debit fields.

Case 2

All inputs will contain the details of the Receiving Entry being edited.

A `Delete` button will be on the Page in addition to the 2 `Submit` buttons.

Intermediate Conditions

A Bank Account, Date, Amount, Memo and at least one Transaction is required. The entry must either be specified as an *ACH* Payment or have a Check Number. Check Numbers must be unique per Bank Account.

Checking a row's Delete box will remove that row from the table.

Final Conditions

A valid form will create or update the Bank Spending Entry and the entered Transactions. The User will be redirected to the *Bank Spending Entry's Detail Page*.

If the Entry is specified as Void, it's Transactions are refunded and it's Amount zeroed.

If the form was submit using the `Submit & Add More` button, the Screen will refresh with a blank form.

Case 2

Submitting the form via the `Delete` button will remove the Bank Spending Entry and it's Transactions, refunding each Account's balance. The User will be redirected to the *Bank Account's Journal*.

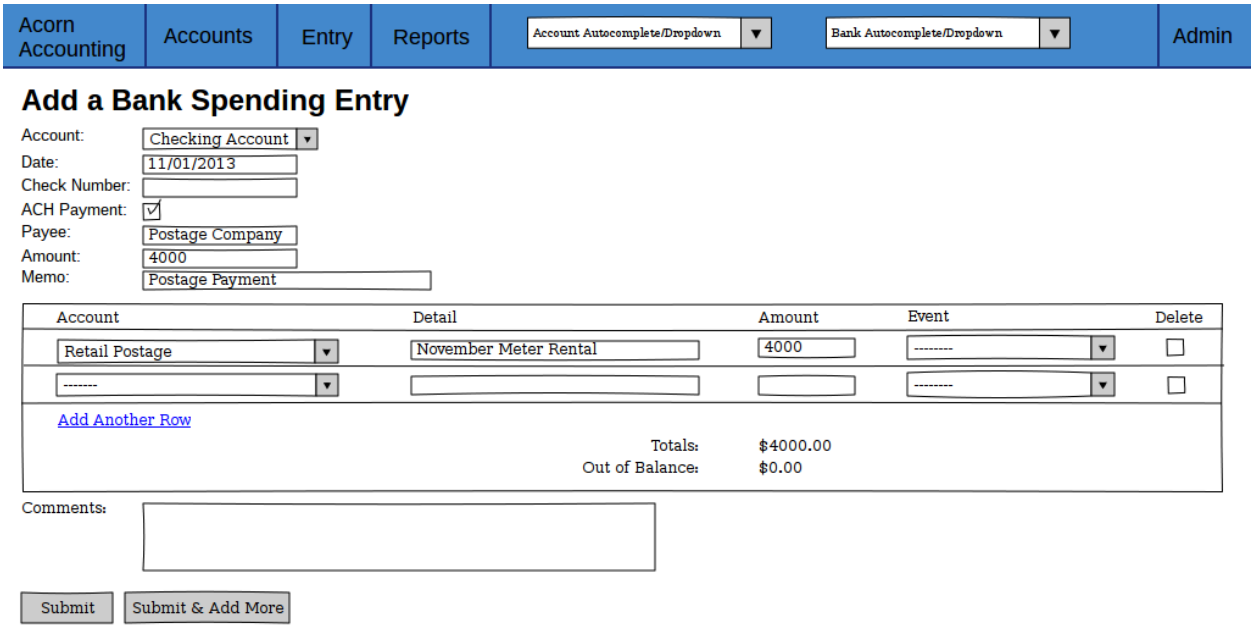

Figure 6.25: A Sample Add Bank Receiving Entry Page, showing a balanced *ACH* withdrawl.

Add Fiscal Year Page

This Page will be used to start new Fiscal Years, purging Transactions from previous years and resetting Income and Expense balances.

Entry Conditions

This Page is accessible through the `New Fiscal Year` link in the *Admin Sub-Menu*.

Initial Conditions

The Page will contain a Heading, explanation text and a form to enter the new Year's details and select Accounts to exclude from purging.

The explanation text should describe the procedure of creating a New Fiscal Year and warnings of the irreversible deletions caused by starting a New Year.

The Fiscal Year Form will have fields for the Year, Ending Month and Period of the new Year. There will be a table containing all Accounts, with columns for checkboxes, the Account Names and Last Reconciled dates. Any Account that has been reconciled previously will default to being excluded.

Intermediate Conditions

If there is a pre-existing Fiscal Year, the new Year must be after the Ending Month and Year of the previous Year. The difference between the previous and new Year's Ending Month and Year cannot be greater than the new Year's period.

The `Current Year Earnings` and `Retained Earnings` Equity Accounts are required to have been created.

Final Conditions

If there is a pre-existing Fiscal Year:

- Historical Accounts will be created for the previous Year
- All Journal Entries that do not have unreconciled Transactions with Accounts selected in the Exclude table will be deleted
- The balance for all Income, Cost of Sales, Expense, Other Income and Other Expense Accounts will be set to zero at the beginning of the new year
- The balance of the Current Year Earnings Account will be transfered to the Retained Earnings Account

The User will be redirected to the *Chart of Accounts*.

Future Features

Trip Entry Form

Create an Entry Form for Communards to easily record any Trips they have taken out. The Entry Fields should include:

- Tripper Name
- Trip Number
- Trip Date
- Cash Advance
- Cash Returned

Transactions should be grouped by Merchant or Store, and totals should be calculated not only for the entire entry, but also for each individual store.

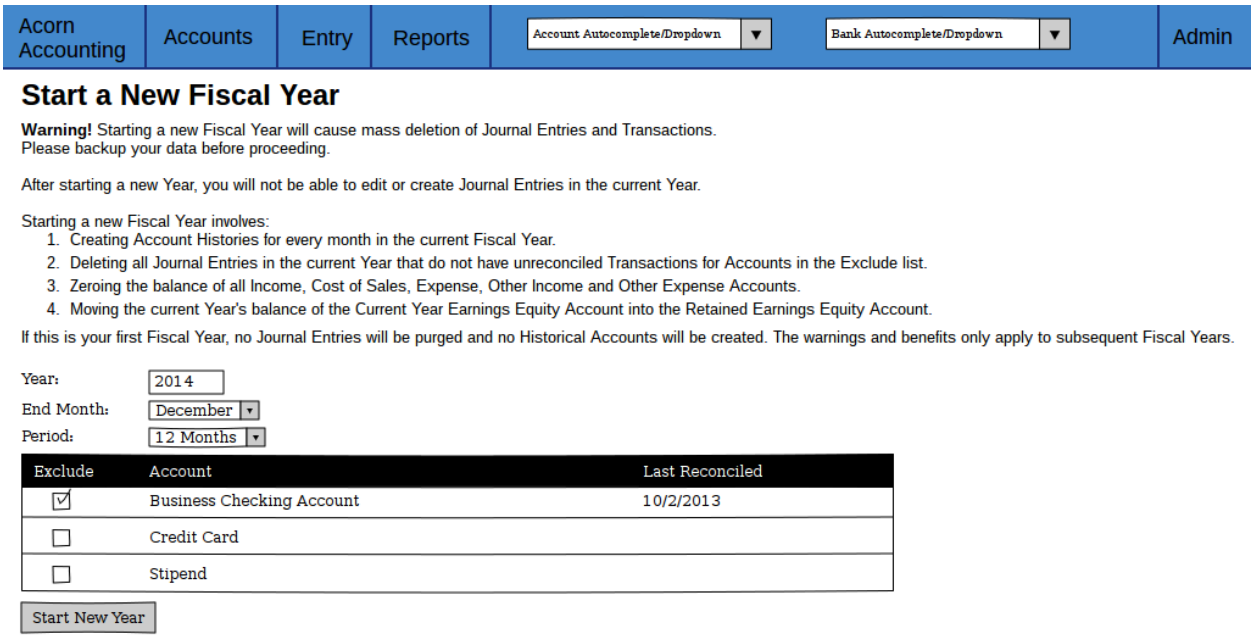

Figure 6.26: A Sample New Fiscal Year Page, showing the default of checking previously reconciled Accounts.

Each Transaction line should have an "Item Price", "Sales Tax" and "Total Price" field, with the Sales Tax defaulting to a "Sales Tax %" Setting. The Total Price should be calculated whenever the Item Price or Sales Tax fields are changed.

A valid form will require that `Cash Advance = Total Prices + Cash Returned`.

A successful submission will redirect the User to a printable version of their submitted entry, so they may print the form and attach receipts.

Brainstorm clean ways to deal with Store Accounts and Returns.

Credit Card Entry Form

Create an Entry Form for Communards to record Personal Credit Cards. Entry fields should include:

- Person's Name
- Credit Card Name
- Statement Date
- Statement Total

Transactions should have a `Detail`, `Amount` and `Account` field.

A valid form will require that `Statement Balance = Sum of Amounts`.

A successful submission will redirect the User to a printable version of their submitted entry, so they may print the form and attach receipts.

Bank Statement Form

Use OFX to pull bank statements from banks.

Present the user with a form prefilled with all entered transactions that match. If no transaction matches, fill the amount and date leaving account and memo blank. User will enter these fields. For non-matched lines, the application will create either a bank spending or receiving entry for the bank and user-specified account.

Budgeted Accounts

Add optional Budgets to Accounts

Do we do:

- Budgets per year, month, quarter?
- Rolling budgets or fixed date?

Application should display:

- Accounts balance as % of budget
- Accounts that are overbudget or close to their budget.
- Alert for accountants/administrators when Accounts hit their budget or get within $X.XX of their budget.

Tax Preparation Report

A report to help the tax preparer collect information.

Yearly Budget Report

This report should show the things we go over in our annual budget meeting. YTD Expense spending and Income, compared with the YTD for previous years.

By default it should show from the beginning of the current fiscal year to the current date. Maybe it could extrapolate for the rest of the year and show breakdown by quarter?

People Tracking

Allow us to add members, interns, prov. members, visitors etc. Can be used to automatically pay stipends and bonuses for interns and members. Can also help the tax preparer learn the # of intern days in the year.

Can track the "Render unto Caesar" form details.

Store email address & email monthly balance reports to members and interns.

Alternate Home Page

The Home Page is currently the *Chart of Accounts Page*.

An alternative is making the Home Page a "Recent Activity/Alerts" page. This new page could show:

- Recently created or updated Journal Entries
- Balances of "watched" Accounts
- Over-budget or close-to-budget Accounts (see *Budgeted Accounts*)

Design Specifications describe the interface between the User and the Application, how the User interacts with the Application and how the Application responds.

AcornAccounting is expected to be used by at least 3 different types of people:

1. Communards - People interested in checking balances and project budgets
2. Accountants - People responsible for entering in new data
3. Budgeters - People responsible for analyzing the data for planning and external uses

Use Cases lists every activity that each of these people are expected to perform. *Screen Descriptions* describes how the application allows the actor to perform the activities. Included in the Descriptions are the *Screen's Conditions*, which describe the expected usage of each Screen.

The *Future Features* section contains the planning and design information for features not implemented in the current release.

See the *Specification Explanation* for additional information on Use Cases and Screen Descriptions/Conditions.

CHAPTER 7

Technical Specifications

core Package

forms Module

class core.forms.**BootstrapAuthenticationForm**(**args*, ***kwargs*)
A login form using bootstrap attributes.

class core.forms.**DateRangeForm**(**args*, ***kwargs*)
A form for acquiring a DateRange via a start_date and enddate.

class core.forms.**RequiredBaseFormSet**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*)
A BaseFormSet that requires at least one filled form.

class core.forms.**RequiredBaseInlineFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)
A BaseInlineFormSet that requires at least one filled form.

class core.forms.**RequiredFormSetMixin**
This class ensures at least one form in the formset is filled.

clean()
Ensure that at least one filled form exists.

context_processors Module

core.context_processors.**template_accessible_settings**(*request*)
Inject template-accessible settings into every context.

core Module

core.core.**first_day_of_month**()
Return the first day of the current month in MM/DD/YYYY format.

core.core.**first_day_of_year**()
Return the first day of the current year in MM/DD/YYYY format.

core.core.**process_month_start_date_range_form**(*request*)
Returns a DateRangeForm, start_date and stop_date based on the request GET data. Defaults to using beginning of this month to today.

core.core.**process_quick_search_form**(*get_dictionary*, *get_variable*, *form*)
Return a form and the id of the related model.

QuickSearchForms are Forms with a single Select input filled with model instances. Each Search submits with a different GET variable.

This function determines if the get_variable is in the request's GET data. If so, and the Form is valid, it will bind the form and return a tuple containing the bound form and the id of the selected object. Otherwise, the function will return tuple containing an unbound form and None.

For usage, see core.templatetags.core_tags().

Parameters

- **get_dictionary** – The request's GET dictionary.
- **get_variable** (*str*) – The GET variable to search the request for, it's presence indicates form submission.
- **form** (Form) – The form class to use.

Returns A tuple containing a bound or unbound Form and the objects id

Return type tuple

core.core.**process_year_start_date_range_form**(*request*)
Returns a DateRangeForm, start_date and stop_date based on the request GET data. Defaults to using beginning of this year to today.

core.core.**remove_trailing_zeroes**(*number*)
Reduce any trailing zeroes down to two decimal places.

Parameters **number** (*A number or string representing a number.*) – The number to reduce.

Returns The number with zeroes past the 2nd decimal place removed.

Return type String

core.core.**today_in_american_format**()
Return the Today's Date in MM/DD/YYYY format.

middleware Module

class core.middleware.**LoginRequiredMiddleware**
Add Ability to Hide All Pages From Unauthorized Users.

Checks the REQUIRE_LOGIN setting to see if the Middleware is enabled, and adds exceptions for any URL regexes defined in the REQUIRE_LOGIN_EXEMPT_URLS setting.

process_request(*request*)
Redirect to Login Page if Enabled & User Unauthenticated.

views Module

Abstract views used throughout the application.

class core.views.**AddApprovableEntryView**
A View for adding, editing, approving and deleting Approvable Entries

To use this view, subclass it and set the following class attributes:

- entry_class - The class of the Approvable Entry
- entry_form_class - The class of the Entry's Form
- transaction_formset_class - The class of the Entry's Transaction's FormSet
- verbose_name - A user-friendly name for the Entry type
- receipt_class - The class of the Entry's Receipts
- receipt_entry_field - The field to access an Entry from a Receipt
- list_entries_view - The view used to list all Entries(a string of the module-dotted path for use with reverse, e.g. `trips.views.list_trip_entries`)
- add_entry_view - The view used to add a new Entry
- show_entry_view - The view used to show a single Entry.

The Entry must have the following methods defined:

- entry.get_number() - Produces a journal number like XX#000123
- entry.approve_entry() - Creates a JournalEntry, Transactions and Receipts from the Entry. Should not delete the Entry.
- entry.get_next_entry() - Produces a Queryset of the Entries to fetch next, if any exist.

The following instance variables are set during request processing:

- entry - The model instance being created/modified.
- entry_form - The bound form
- transaction_formset - The bound formset

render(*request*, *entry_id*, *template_name*)
Use the specified class attributes to render the view.

core.views.**list_entries**(*request*, *template_name*, *entry_class*)
Return a response for listing all specified Entries.

core.views.**show_single_entry**(*request*, *entry_id*, *template_name*, *entry_class*)
View single Entry, specified by id and class.

Subpackages

commands Package

generatedata Module

class core.management.commands.generatedata.**Command**
Bases: django.core.management.base.BaseCommand

args = ''

handle(**args*, ***options*)

help = 'Generate mass random data'

core.management.commands.generatedata.**get_random_account**()

core.management.commands.generatedata.**get_random_bank**()

import_myob Module

Django Accounting Command to import MYOB Accounts and Transactions

Requires all child accounts for a header to be directly underneath the header, coming before any child headers.

Creates Accounts based on MYOB data, then all Entries/Transactions.

Export Account and Journal entries in MYOB and place in root folder of project.

class core.management.commands.import_myob.**Command**
Bases: django.core.management.base.BaseCommand

args = ''

handle(**args*, ***options*)

help = ' Will created Accounts, Entries and Transactions based on MYOB data.\n ACCOUNTS.TXT and JOURNAL.TX

templatetags Package

accounting_filters Module

core.templatetags.core_filters.**capitalize_words**(*value*)
Capitalize the first letter of each lowercase word.

core.templatetags.core_filters.**currency**(*value*)

core.templatetags.core_filters.**tab_number**(*value*)

accounting_tags Module

core.templatetags.core_tags.**date_range_form**(*form*)

core.templatetags.core_tags.**quick_search**(*GET*)

core.templatetags.core_tags.**receipt_list**(*journal_entry*, *show_heading=True*)
Render a Journal Entries Receipts in a <ul>.

accounts Package

models Module

class accounts.models.**Account**(**args*, ***kwargs*)
Holds information on Accounts.

get_balance()
Returns the value balance for the Account.

The Account model stores the credit/debit balance in the balance field. This method will convert the credit/debit balance to a value balance for Account.type where a debit increases the Account's value, instead of decreasing it(the normal action).

The Current Year Earnings Account does not source it's balance field but instead uses the Sum of all Accounts with type of 4 to 8.

See also:

flip_balance() method for more information on value balances.

Returns The Account's current value balance.

Return type decimal.Decimal

get_balance_by_date

Calculate the Account's balance at the end of a specific date.

For the Current Year Earnings Account, Transactions from all Accounts with type of 4 to 8 will be used.

Parameters **date** (*datetime.date*) – The day whose balance should be returned.

Returns The Account's balance at the end of the specified date.

Return type decimal.Decimal

get_balance_change_by_month(*date*)

Calculates the Accounts net change in balance for the month of the specified date.

For the Current Year Earnings Account, Transactions from all Accounts with type of 4 to 8 will be used.

Parameters **date** (*datetime.date*) – The month to calculate the net change for.

Returns The Account's net balance change for the specified month.

Return type decimal.Decimal

class accounts.models.**BaseAccountModel**(**args*, ***kwargs*)

Abstract class storing common attributes of Headers and Accounts.

Subclasses must implement the _calculate_full_number and _get_change_tree methods.

clean()

Set the type and calculate the full_number.

The type attribute will be inherited from the parent and the full_number will be calculated if the object has an id.

delete(**args*, ***kwargs*)

Renumber Headers or Accounts when deleted.

get_full_number()

Retrieve the Full Number from the model field.

save(**args*, ***kwargs*)

Resave Headers or Accounts if the parent has changed.

This method first checks to see if the parent attribute has changed. If so, it will cause the object and all related objects(the change_tree) to be saved once the pending changes have been saved.

class accounts.models.**Header**(**args*, ***kwargs*)

Groups Accounts Together.

get_account_balance()

Traverse child Headers and Accounts to generate the current balance.

Returns The Value Balance of all Accounts and Headers under this Header.

Return type decimal.Decimal

class accounts.models.**HistoricalAccount**(**args*, ***kwargs*)

A model for Archiving Historical Account Data. It stores an Account's balance (for Assets, Liabilities and Equities) or net_change (for Incomes and Expenses) for a certain month in a previous Fiscal Years.

Hard data is stored in additon to a link back to the originating Account.

Note: This model is automatically generated by the add_fiscal_year() view.

Note: This model does not inherit from the BaseAccountModel because it has no parent attribute and cannot inherit from MPTTModel.

---

account

The Account this HistoricalAccount was generated for. The HistoricalAccount will remain even if the Account is deleted.

number

The Account number, formatted as *type-num* must be unique with respect to date.

name

The Account's name, must be unique with respect to date.

type

The Account's type, chosen from TYPE_CHOICES.

amount

The end-of-month balance for HistoricalAccounts with type 1-3, and the net change for HistoricalAccounts with type 4-8. This field represents the credit/debit amount not the value amount. To retrieve the value amount for a HistoricalAccount use the get_amount() method.

date

A datetime.date object representing the 1st day of the Month and Year the archive was created.

flip_balance

Determines whether the HistoricalAccount.amount should be flipped based on the HistoricalAccount.type.

For example, debits(negative HistoricalAccount.amount) increase the value of Assets, Expenses, Cost of Sales and Other Expenses, while decreasing the value of all other Account Types.

In essence, this method will return True if the credit/debit amount needs to be negated(multiplied by -1) to display the value amount, and False if the credit/debit amount is the displayable value amount.

get_absolute_url(**moreargs*, ***morekwargs*)

The default URL for a HistoricalAccount points to the listing for the date's month and year.

get_amount

Calculates the flipped/value balance or net_change for Asset, Cost of Sales, Expense and Other Expense HistoricalAccounts.

The amount field for HistoricalAccounts represents the credit/debit amounts but debits for Asset, Cost of Sales, Expense and Other Expenses represent a positive value instead of a negative value. This function returns the value amount of these accounts instead of the debit/credit amount. E.g., a negative(debit) amount will be returned as a positive value amount.

If the HistoricalAccount is not one of these types, the HistoricalAccounts normal amount will be returned.

Returns The value amount for the HistoricalAccount

Return type decimal.Decimal

managers Module

class accounts.managers.**AccountManager**
A Custom Manager for the Account Model.

This class inherits from the CachingTreeManager.

active()
This method will return a Querset containing all Active Accounts.

get_banks()
This method will return a Queryset containing any Bank Accounts.

forms Module

views Module

signals Module

accounts.signals.**transaction_delete**(*sender*, *instance*, ***kwargs*)
Refund Transaction before deleting from database.

accounts.signals.**transaction_postsave**(*sender*, *instance*, ***kwargs*)
Change Account Balance on Save.

accounts.signals.**transaction_presave**(*sender*, *instance*, ***kwargs*)
Refund Account balances if updating a Transaction.

entries Package

models Module

class entries.models.**Transaction**(**args*, ***kwargs*)
Transactions itemize Account balance changes.

Transactions are grouped by Entries and Events. Entries group Transactions by date while Events group by specific events.

Transactons can be related to Entries through the journal_entry, bankspend_entry or bankreceive_entry attributes, or through the main_transaction attribute of the BankReceivingEntry or BankSpendingEntry models. Transactions may only be related to Entries through one of these ways, never multiple ones.

journal_entry
The JournalEntry this Transaction belongs to, if any.

bankspend_entry
The BankSpendingEntry this Transaction belongs to, if any.

bankreceive_entry
The BankReceivingEntry this Transaction belongs to, if any.

account
The Account this Transaction is charged to.

detail
Information about the specific charge.

balance_delta
The change in balance this Transaction represents. A positive value indicates a Credit while a negative value is a Debit.

event
The Event this Transaction is related to, if any.

reconciled
Whether or not this Transaction has been marked as Reconciled.

date
The datetime.date of the Transaction. By default, this is automatically pulled from the related Entry when the Transaction is saved.

clean()
Prevent relations to a void BankSpendingEntry.

get_absolute_url(**moreargs*, ***morekwargs*)
Return a URL to the related Entry's detail page.

get_entry_number()
Return the related Entry's number.

get_final_account_balance()
Return the Account's value balance after the Transaction has occured.

This is accomplished by subtracting the balance_delta of all newer Transactions balance.

Note: The value balance is not the same as credit/debit balance. For Assets, Liabilities and Equity Accounts, a debit balance means a positive value balance instead of the normal negative value balance.

Returns The Account's post-transaction value balance.

Return type Decimal

get_initial_account_balance()
Return the value balance of the Account from before the Transaction occured.

Returns The Account's pre-transaction value balance.

Return type Decimal

get_journal_entry()
Return the related Entry.

get_memo()
Return the related Entry's memo.

save(*pull_date=True*, **args*, ***kwargs*)
Pull the date from the related Entry before saving.

class entries.models.**BaseJournalEntry**(**args*, ***kwargs*)
Journal Entries group Transactions by discrete points in time.

For example, a set of transfers, a check being deposited, stipends being paid.

Journal Entries ensure that `Transactions` are balanced, that there is an equal amount of credits for every debit.

Note: This class is an abstract class to prevent multi-table inheritance.

See also:

Module `entries.views` Views related to showing, creating and editing Entries.

`date`
The date the entry occured.

`memo`
A short description of the Entry.

`comments`
Any additional comments about the Entry.

`created_at`
The date and time the Entry was created.

`updated_at`
The date and time the Entry was last updated. Defaults to `created_at`.

`get_absolute_url`()
Return a link to the Entry's detail page.

`get_edit_url`()
Return an edit link to the Entry's edit page.

`get_number`()
Return the number of the Entry.

`in_fiscal_year`()
Determines whether the `BaseJournalEntry.date` is in the current `FiscalYear`.

Returns True if there is no current `FiscalYear`.

Returns Whether or not the `date` is in the current `FiscalYear`.

Return type bool

class `entries.models.`**`JournalEntry`**(**args*, ***kwargs*)
A concrete class of the `BaseJournalEntry` model.

`get_absolute_url`(**moreargs*, ***morekwargs*)
Return a link to the Entry's detail page.

`save`(**args*, ***kwargs*)
Save all related `Transactions` after saving.

class `entries.models.`**`BankSpendingEntry`**(**args*, ***kwargs*)
Holds information about a Check or ACH payment for a Bank `Account`. The `main_transaction` is linked to the Bank `Account`.

Bank Spending Entries credit the Bank `Account` (via the `main_transaction`) while debiting all related `Transactions`.

`check_number`
The number of the Check, if applicable. An ACH Payment should have no `check_number` and `ach_payment` value of `True` will cause the `check_number` to be set to `None`. This value must be unique with respect to the `main_transaction's account` attribute.

ach_payment
A boolean representing if this BankSpendingEntry is an ACH Payment or not. If this is True the check_number will be set to None.

payee
An optional Payee for the BankSpendingEntry.

void
A boolean representing whether this BankSpendingEntry is void. If this value switches from False to True, all of this BankSpendingEntry's Transactions will be deleted and it's main_transaction will have it's balance_delta set to 0. Switching void back to False will simply allow transactions to be saved again, it will not recreate any previouse Transactions.

main_transaction
The Transaction that links this BankSpendingEntry with it's Bank Account.

clean()
Only a check_number or an ach_payment must be entered, not both.

The check_number must be unique per BankSpendingEntry.main_transaction Account.

get_absolute_url(**moreargs*, ***morekwargs*)
Return a link to the Entry's detail page.

get_edit_url()
Return the Entry's edit URL.

get_number
Return the formatted check_number or ##ACH##.

save(**args*, ***kwargs*)
Delete related Transactions if void, update Transaction dates.

class entries.models.**BankReceivingEntry**(**args*, ***kwargs*)
Holds information about receiving money for a Bank Account. The main_transaction is linked to the Bank Account.

Bank Receiving Entries debit the Bank Account (via the main_transaction) while crediting all related Transactions.

payor
The Person or Company making the payment.

main_transaction
The Transaction that links this BankSpendingEntry with it's Bank Account.

get_absolute_url(**moreargs*, ***morekwargs*)
Return the Entry's detail page.

get_edit_url()
Return the Entry's edit page.

get_number
Return the Entry's formatted number.

save(**args*, ***kwargs*)
Set the date's of all related Transactions.

managers Module

class entries.managers.**TransactionManager**
A Custom Manager for the Transaction Model.

Note: Using this Manager as a Model's default Manager will cause this Manager to be used when the Model is accessed through Related Fields.

get_query_set()
Return a caching.base.CachingQuerySet.

get_totals(*net_change=False*)
Calculate debit and credit totals for the respective Queryset.

Groups and Sums the default Queryset by positive/negative balance_delta. Totals default to 0 if no corresponding Transaction is found.

Optionally:

- Returns the Net Change(credits + debits) with the totals.

Parameters

- **query** (Q) – Optional Q query used to filter Manager's Queryset.
- **net_change** (*bool*) – Calculate the difference between debits and credits.

Returns Debit and credit sums and optionally the net_change.

Return type tuple

class entries.managers.**TransactionQuerySet**(**args*, ***kw*)
A wrapper for the caching.base.CachingQuerySet.

The methods of this class mimic the TransactionManager class. This allows the chaining of our custom methods. For example:

Transaction.objects.filter(id__gt=500).get_totals()

get_totals(*net_change=False*)
See TransactionManager.get_totals().

forms Module

class entries.forms.**BankReceivingForm**(**args*, ***kwargs*)
A form for the BankReceivingEntry model.

clean_amount()
Negate the amount field.

BankReceivingEntries debit their bank Account> so a positive amount should become a debit and vice versa.

entries.forms.**BankReceivingTransactionFormSet**
A FormSet of Transactions for use with BankReceivingEntries, derived from the BaseBankTransactionFormSet.

alias of TransactionFormFormSet

class entries.forms.**BankSpendingForm**(**args*, ***kwargs*)
A form for the BankSpendingEntry model.

entries.forms.**BankSpendingTransactionFormSet**
A FormSet of Transactions for use with BankSpendingEntries, derived from the BaseBankTransactionFormSet.

alias of TransactionFormFormSet

class entries.forms.**BankTransactionForm**(**args*, ***kwargs*)
A form for entry of Bank Transactions.

Bank Transactions do not have a credit and debit field, instead they have only an amount. Whether this amount is a credit or debit depends on if the Transaction is related to a models.BankSpendingEntry or a models.BankSpendingEntry.

clean()
Set the balance_delta to the entered amount.

class entries.forms.**BaseBankForm**(**args*, ***kwargs*)
A Base form for common elements between the BankSpendingForm and the BankReceivingForm.

The account and amount fields be used to create the Entries main_transaction.

account
The Bank Account the Entry is for.

amount
The total amount this Entry represents.

clean()
Cleaning the BaseBankForm will modify or create the BankSpendingEntry.main_transaction or BankReceivingEntry.main_transaction.

The main_transaction will not be saved until this form's save method is called.

save(**args*, ***kwargs*)
Saving the BaseBankFormForm will save both the BaseBankForm and the BankSpendingEntry.main_transaction or BankReceivingEntry.main_transaction.

class entries.forms.**BaseBankTransactionFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)
An InlineFormSet used to validate it's form's amounts balance with the self.entry_form's amount.

Note: The self.entry_form attribute must be set by the view instantiating this form.

---

clean()
Checks that the Transactions' amounts balance the Entry's amount.

class entries.forms.**BaseEntryForm**(**args*, ***kwargs*)
An abstraction of General and Bank Entries.

clean_date()
The date must be in the Current FiscalYear.

class entries.forms.**BaseTransactionFormSet**(*data=None*, *files=None*, *instance=None*, *save_as_new=False*, *prefix=None*, *queryset=None*, ***kwargs*)
A FormSet that validates that a set of Transactions is balanced.

clean()
Checks that debits and credits balance out.

class entries.forms.**JournalEntryForm**(**args*, ***kwargs*)
A form for JournalEntries.

class entries.forms.**TransactionForm**(**args*, ***kwargs*)
A form for Transactions.

It splits the balance_delta field into a credit and debit field. A debit results in a negative balance_delta and a credit results in a positive balance_delta.

clean()
Make sure only a credit or debit is entered and set it to the balance_delta.

entries.forms.**TransactionFormSet**
A FormSet for Transactions, derived from the BaseTransactionFormSet.

alias of TransactionFormFormSet

class entries.forms.**TransferForm**(**args*, ***kwargs*)
A form for Transfer Entries, a specialized JournalEntry.

Transfer Entries move a discrete amount between two Accounts. The source is debited while the destination is credited.

source
The Account to remove the amount from.

destination
The Account to move the amount to.

amount
The amount of currency to transfer between Accounts.

detail
Any additional details about the charge.

clean()
Ensure that the source and destination Accounts are not the same.

entries.forms.**TransferFormSet**
A FormSet for Transfer Transactions, derived from the TransferForm.

alias of TransferFormFormSet

views Module

entries.views.**add_bank_entry**(*request*, **args*, ***kwargs*)
Add, Edit or Delete a BankSpendingEntry or BankReceivingEntry.

A journal_type of CD corresponds to BankSpendingEntries while a journal_type of CR corresponds to BankReceivingEntries

If there is no BankSpendingEntry or BankReceivingEntry with an id of the entry_id parameter, a new BankSpendingEntry or BankReceivingEntry will be created.

If the request contains POST data, either validate and save the data or delete the JournalEntry and all related Transactions, depending on if a submit or delete is sent.

Parameters

- **entry_id** (*int*) – The id of the Entry to edit. If None then a new entry will be created.
- **journal_type** (*str*) – The bank journal of the Entry(CD or CR).
- **template_name** (*str*) – The template to use.

Returns HTTP response containing a JournalEntryForm, a TransactionFormSet and a journal_type of GJ.

Return type HttpResponse or *~django.http.HttpResponseRedirect*

entries.views.**add_journal_entry**(*request*, **args*, ***kwargs*)
Add, Edit or Delete a JournalEntry.

If there is no JournalEntry with an id of the entry_id parameter, a new JournalEntry will be created.

If the request contains POST data, either validate and save the data or delete the JournalEntry and all related Transactions, depending on if a submit or delete is sent.

Parameters

- **entry_id** (*int*) – The id of the Entry to edit. If None then a new entry will be created.
- **template_name** (*str*) – The template to use.

Returns HTTP response containing a JournalEntryForm, a TransactionFormSet and a journal_type of GJ.

Return type HttpResponse or HttpResponseRedirect

entries.views.**add_transfer_entry**(*request*, **args*, ***kwargs*)
Add a Transfer Entry, a specialized JournalEntry.

Transfer Entries are JournalEntries where a discrete amount is being transfered from one account to another.

Normally, JournalEntries require that the Credit and Debit totals are equal. Transfer Entries require that every single Credit charge is balanced with an equal Debit charge.

Transfer Entries do not have their own class, they use the JournalEntry model.

Parameters **template_name** (*str*) – The template to use.

Returns HTTP response containing a JournalEntryForm, a TransferFormSet and a verbose_journal_type of Transfer Entry.

Return type HttpResponse

entries.views.**journal_ledger**(*request*, *template_name='entries/journal_ledger.html'*)
Display a list of Journal Entries.

Parameters **template_name** (*str*) – The template to use.

Returns HTTP response containing JournalEntry instances as context.

Return type HttpResponse

entries.views.**show_bank_entry**(*request*, *entry_id*, *journal_type*)
Display the details of a BankSpendingEntry or BankReceivingEntry.

Parameters

- **entry_id** (*int*) – The id of the Entry to display.
- **journal_type** (*str*) – The bank journal of the Entry(CD or CR).
- **template_name** (*str*) – The template to use.

Returns HTTP response containing the Entry instance and additional details as context.

Return type HttpResponse

entries.views.**show_journal_entry**(*request*, *entry_id*, *template_name='entries/entry_detail.html'*)
Display the details of a JournalEntry.

Parameters

- **entry_id** (*int*) – The id of the JournalEntry to display.

- **template_name** (*str*) – The template to use.

Returns HTTP response containing the `JournalEntry` instance and additional details as context.

Return type `HttpResponse`

fiscalyears Package

models Module

class `fiscalyears.models.`**`FiscalYear`**(**args*, ***kwargs*)

A model for storing data about the Company's Past and Present Fiscal Years.

The Current Fiscal Year is used for generating Account balance's and Archiving `Account` instances into `HistoricalAccounts`.

See also:

View `add_fiscal_year()` This view processes all actions required for starting a New Fiscal Year.

`year`
The ending Year of the Financial Year.

`end_month`
The ending Month of the Financial Year. Stored as integers, displayed with full names.

`period`
The length of the Fiscal Year in months. Available choices are 12 or 13.

`date`
The first day of the last month of the Fiscal Year. This is not editable, it is generated by the `FiscalYear` when saved, using the `end_month` and `year` values.

`save`(**args*, ***kwargs*)
The `FiscalYear` `save` method will generate the `date` object for the `date` attribute.

fiscalyears Module

`fiscalyears.fiscalyears.`**`get_start_of_current_fiscal_year`**()

Determine the Start Date of the Latest `FiscalYear`.

If there are no `FiscalYears` then this method will return `None`.

If there is one `FiscalYear` then the starting date will be the `period` amount of months before it's `date`.

If there are multiple `FiscalYears` then the first day and month after the Second Latest `FiscalYear` will be returned.

Returns The starting date of the current `FiscalYear`.

Return type `datetime.date` or `None`

forms Module

class `fiscalyears.forms.`**`FiscalYearAccountsForm`**(**args*, ***kwargs*)

This form is used to select whether to exclude an account from the `Transaction` purging caused by the `add_fiscal_year()` view. Selected `Accounts` will retain their unreconciled `Transactions`.

This form is used by the `modelformset_factory()` to create the `FiscalYearAccountsFormSet`.

See also:

View **`add_fiscal_year()`** This view processes all actions required for starting a New Fiscal Year.

`fiscalyears.forms.`**`FiscalYearAccountsFormSet`**
An FormSet of `Accounts` using the `FiscalYearAccountsForm` as the base form.

alias of `AccountFormFormSet`

class `fiscalyears.forms.`**`FiscalYearForm`**(**args*, ***kwargs*)
This form is used to create new `Fiscal Years`.

The form validates the period length against the new Fiscal Year's End Month and Year. To pass validation, the new Year must be greater than any previous years and the end Month must create a period less than or equal to the selected Period.

See also:

View **`add_fiscal_year()`** The `add_fiscal_year()` view processes all actions required for starting a New Fiscal Year.

Form **`FiscalYearAccountsForm`** This form allows selection of the accounts to exclude in the New Fiscal Year's Transaction purging.

`clean`()
Validates that certain `Accounts` exist and that the entered Year is in the allowed range.

Validates that no previous year's ending month is within the entered `Period` with respect to the entered `end_month`.

If the form causes a period change from 13 to 12 months, the method will ensure that there are no `Transactions` in the 13th month of the last `FiscalYear`.

If there are previous `FiscalYears` the method will make sure there are both a `Current Year Earnings` and `Retained Earnings` Equity `Accounts` with a `type` of `3`.

`clean_year`()
Validates that the entered `Year` value.

The value is required to be greater than or equal to any previously entered `Year`.

`views` Module

`fiscalyears.views.`**`add_fiscal_year`**(*request*, **args*, ***kwargs*)
Creates a new `FiscalYear` using a `FiscalYearForm` and `FiscalYearAccountsFormSet`.

Starting a new `FiscalYear` involves the following procedure:

1. Setting a `period` and Ending `month` and `year` for the New Fiscal Year.
2. Selecting Accounts to exclude from purging of unreconciled `Transactions`.
3. Create a `HistoricalAccount` for every `Account` and month in the previous `FiscalYear`, using ending balances for Asset, Liability and Equity `Accounts` and balance changes for the others.
4. Delete all `Journal Entries`, except those with unreconciled `Transactions` with `Accounts` in the exclude lists.
5. Move the `balance` of the `Current Year Earnings` `Account` into the `Retained Earnings` `Account`.

6.Zero the `balance` of all Income, Cost of Sales, Expense, Other Income and Other Expense `Accounts`.

Parameters **template_name** (*string*) – The template to use.

Returns HTTP response containing `FiscalYearForm` and `FiscalYearAccountsFormSet` as context. Redirects if successful POST is sent.

Return type HttpResponse or HttpResponseRedirect

events Package

models Module

class events.models.**BaseEvent**(**args*, ***kwargs*)

An abstract class for the commonalities of the `Event` and `HistoricalEvent` models.

name

The name of the archived `Event`.

number

The number designated to the `Event`. Generated from the `Event's date` and `abbreviation`.

date

The date of the `Event`.

city

The city the `Event` occurred in.

state

The state or region the `Event` occurred in.

class Meta

Order Events by Date.

class events.models.**Event**(**args*, ***kwargs*)

Hold information about Events.

get_absolute_url(**moreargs*, ***morekwargs*)

Return the URL of the Event's Details Page.

get_net_change()

Return the sum of all related credit and debit charges.

Return type `Decimal`

save(**args*, ***kwargs*)

Set the `number` before saving.

class events.models.**HistoricalEvent**(**args*, ***kwargs*)

Historical Events record `Events` that occurred in previous `Fiscal Years`.

When a `Fiscal Year` is closed, all `Transactions` from that `Fiscal Year` are purged. Historical Events preserve the state of `Events` before the purging. They do not save the `Transaction` data but rather the overall data such as the total number of debits.

Instances of this model are automatically created by the `add_fiscal_year()` view.

credit_total

The total number of Credits related to the `Event`.

debit_total
The total number of Debits related to the `Event`. This value should be a negative number.

net_change
The Net Change of all `Transactions` related to the `Event`.

views Module

events.views.**show_event_detail**(*request*, *event_id*, *template_name='events/event_detail.html'*)
Shows the details of an `Event` instance.

Parameters

- **event_id** (*int*) – The id of the `Event` to show.
- **template_name** (*string*) – The template to use.

Returns HTTP Response containing the `Event` instance, and the `Event's` Debit Total, Credit Total and Net Change.

Return type HttpResponse

reports Package

views Module

reports.views.**events_report**(*request*, *template_name='reports/events.html'*)
Display all `Events`.

Parameters **template_name** (*str*) – The template file to use to render the response.

Returns HTTP Response with an `events` context variable.

Return type HttpResponse

reports.views.**profit_loss_report**(*request*, *template_name='reports/profit_loss.html'*)
Display the Profit or Loss for a time period calculated using all Income and Expense `Accounts`.

The available `GET` parameters are `start_date` and `stop_date`. They control the date range used for the calculations.

This view is used to show the Total, Header and Account Net Changes over a specified date range. It uses the Net Changes to calculate various Profit amounts, passing them as context variables:

•`gross_profit`: Income - Cost of Goods Sold

•`operating_profit`: Gross Profit - Expenses

•`net_profit`: Operating Profit + Other Income - Other Expenses

Also included is the `headers` dictionary which contains the `income`, `cost_of_goods_sold`, `expenses`, `other_income`, and `other_expense` keys. These keys point to the root node for the respective :attr:~accounts.models.BaseAccountModel.type‘.

These nodes have additional attributes appended to them, `total`, `accounts` and `descendants`. `total` represents the total Net Change for the node. `accounts` and `descendants` are lists of child nodes(also with a `total` attribute).

Parameters **template_name** (*str*) – The template file to use to render the response.

Returns HTTP Response with the start/stop dates, `headers` dictionary and Profit Totals

Return type HttpResponse

reports.views.**trial_balance_report**(*request*, *template_name='reports/trial_balance.html'*)

Display the state and change of all `Accounts` over a time period.

The available `GET` parameters are `start_date` and `stop_date`.

The view also provides the `start_date`, `stop_date` and `accounts` context variables.

The `start_date` and `stop_date` variables default to the first day of the year and the current date.

The `accounts` variable is a list of dictionaries, each representing an `Account`. Each dictionary contains the `Account's` number, name, balance at the beginning and end of the date range, total debits and credits and the net change.

Parameters **template_name** (*str*) – The template file to use to render the response.

Returns HTTP Response with start and stop dates and an `accounts` list.

Return type HttpResponse

creditcards Package

models Module

class creditcards.models.**CreditCard**(**args*, ***kwargs*)

A Communard-Visible Wrapper for an `Account`.

account

The `Account` the CreditCard is linked to. This is the Account that will be credited when CreditCardEntries are approved.

name

A name for the Credit Card, used in forms accessible by Communards.

class creditcards.models.**CreditCardEntry**(**args*, ***kwargs*)

Communard entries used as a pre-approval state for JournalEntries.

These are used to group together `CreditCardTransactions` before the Entry is approved by an Accountant.

When approved, the CreditCardEntry massages it's data into a JournalEntry and Transactions so that the relevant Account balances are modified, then deletes itself.

date

The date the entry occured.

card

The CreditCard the entry belongs too.

name

The name of the communard who submitted the entry.

merchant

The name of the merchant the purchase was made at.

amount

The total amount of money spent. This is balanced against related CreditCardTransactions.

comments

Additional comments from the Communard.

created_at
The date & time the Entry was created.

approve_entry()
Creating a JournalEntry Transactions and Receipts from the Entry.

This does not delete the entry, as should be done when an Entry is approved. You **must manually delete** the CreditCardEntry.

Returns the created JournalEntry.

generate_memo()
Create a memo line from the Entry's attributes.

get_edit_url()
Return an edit link to the Entry's edit page.

get_next_entry()
Return a Queryset of the next possible Entries to display.

get_number()
Generate an Entry number using the id.

class creditcards.models.**CreditCardReceipt**(**args*, ***kwargs*)
Stores Receipts for an unapproved CreditCardEntry.

When the CreditCardEntry is approved, these are turned into receipts.models.Receipts.

receipt_file
The actual receipt stored as a file.

creditcard_entry
The CreditCardEntry this receipt bleongs to.

class creditcards.models.**CreditCardTransaction**(**args*, ***kwargs*)
Represents the individual charges for a CreditCardEntry.

Unlike a entries.models.Transaction, a CreditCardTransaction does not affect the balance of it's entries.models.Account.

creditcard_entry
The CreditCardEntry the CreditCardTransaction belongs to

account
The related accounts.models.Account.

detail
Information about the specific charge.

balance_delta
The change in balance this Transaction represents. A positive value indicates a Credit while a negative value is a Debit.

admin Module

forms Module

class creditcards.forms.**CreditCardEntryForm**(**args*, ***kwargs*)
A Form for CreditCardEntries along with multiple CreditCardReceipts.

views Module

class creditcards.views.**AddCreditCardEntry**
Customize the generic AddApprovableEntryView for CreditCardEntries.

entry_class
alias of CreditCardEntry

entry_form_class
alias of CreditCardEntryForm

receipt_class
alias of CreditCardReceipt

creditcards.views.**add_creditcard_entry**(*request*, *entry_id=None*, *template_name='creditcards/credit_card_form.html'*)
Add, edit, approve or delete a CreditCardEntry.

creditcards.views.**list_creditcard_entries**(*request*, **args*, ***kwargs*)
Retrieve every CreditCardEntry.

creditcards.views.**show_creditcard_entry**(*request*, *entry_id*, *template_name='creditcards/show_entry.html'*)
View a CreditCardEntry.

trips Package

models Module

class trips.models.**StoreAccount**(**args*, ***kwargs*)
A Communard-visible Wrapper for an Account.

This is used to allow Communards to select an Account for purchases made on store credit.

class trips.models.**TripEntry**(**args*, ***kwargs*)
Communard entries used as a pre-approval state for JournalEntries.

These are used to group together in-town purchases by Communards before the Entry is approved by an Accountant.

When approved, the TripEntry massages it's data into a JournalEntry and Transactions so that the relevant Account balances are actually modified, then the TripEntry deletes itself.

approve_entry()
Creating a JournalEntry Transactions and Receipts from the Entry.

This does not delete the entry, as should be done when an Entry is approved. You **must manually delete** the TripEntry.

Returns the created JournalEntry.

get_next_entry()
Return the next Entry for Editing/Approval.

get_number()
Generate an Entry Number using the number attribute.

class trips.models.**TripReceipt**(**args*, ***kwargs*)
Stores Receipts for an unapproved TripEntry.

class trips.models.**TripStoreTransaction**(**args*, ***kwargs*)
Represents a purchase at a StoreAccount for a TripEntry.

class trips.models.**TripTransaction**(**args*, ***kwargs*)
Represents the individual charges/returns for a TripEntry.

The creation of a TripTransaction does not affect Account balances, this only occurs when the related TripEntry is approved - by removing the TripEntry and TripTransactions and creating a JournalEntry and Transactions.

admin Module

forms Module

class trips.forms.**TripEntryForm**(**args*, ***kwargs*)
A From for TripEntries along with multiple TripReceipts.

views Module

class trips.views.**AddTripEntryView**
Extend the AddApprovableEntryView to apply to TripEntries.

This view adds an additional formset, the TripStoreTransactionFormSet.

entry_class
alias of TripEntry

entry_form_class
alias of TripEntryForm

receipt_class
alias of TripReceipt

trips.views.**add_trip_entry**(*request*, *entry_id=None*, *template_name='trips/form.html'*)
Add, edit, approve or delete a TripEntry.

trips.views.**list_trip_entries**(*request*, **args*, ***kwargs*)
Retrieve every TripEntry.

trips.views.**show_trip_entry**(*request*, *entry_id*, *template_name='trips/detail.html'*)
View a TripEntry.

bank_import Package

models Module

Models related to Bank Accounts & Imports.

class bank_import.models.**BankAccount**(**args*, ***kwargs*)
An Accountant-Visible Wrapper for an Account.

This lets us keep track of the importer class to use for each Account.

account
The Account the BankAccount is linked to. This is the Account that will have it's Entries imported.

name
A name for the Bank Account, used in forms.

bank
The module/function path to the statement importer to use for this account.

get_importer_class()
Import and return the Importer to use for the Bank Account.

class bank_import.models.**CheckRange**(**args*, ***kwargs*)
Store a Default Account/Memo/Payee for a BankAccount's Checks.

When importing a bank statement, any expense checks that fall within a CheckRange should be pre-filled with the `default_account`, `default_payee`, & `default_memo` of the CheckRange.

bank_account
The `BankAccount` that the CheckRange applies to.

start_number
The starting check number of the range.

end_number
The ending check number of the range.

default_account
The `Account` to use for checks that fall within the range.

default_payee
The Payee to use for checks that fall within the range.

default_memo
The Memo to use for checks that fall within the range.

`forms` Module

Forms & Formsets Used in the `import_bank` Views.

class bank_import.forms.**BankAccountForm**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)
A Form to Select a BankAccount and Upload an Import File.

class bank_import.forms.**ImportFormMixin**
A mixin to help customize Import specific Forms.

class bank_import.forms.**ReceivingImportForm**(**args*, ***kwargs*)
A form for importing unmatched BankReceivingEntries.

class Meta
Customize the field order & widgets.

model
alias of `BankReceivingEntry`

ReceivingImportForm.**save**(**args*, ***kwargs*)
Create the BankReceivingEntry and it's Transactions.

class bank_import.forms.**SpendingImportForm**(**args*, ***kwargs*)
A form for importing unmatched BankSpendingEntries.

class Meta
Remove the `comments` field from the base BankSpendingForm.

model
alias of BankSpendingEntry

SpendingImportForm.**save**(**args*, ***kwargs*)
Create the BankSpendingEntry and it's Transactions.

class bank_import.forms.**TransferImportForm**(**args*, ***kwargs*)
A form for importing unmatched Transfers.

save()
Save the Transfer by Creating a General Journal Entry.

views Module

Views for Importing Bank Statements.

bank_import.views.**import_bank_statement**(*request*, **args*, ***kwargs*)
Render the Import Upload Form, & the FormSets.

importers.base Module

Abstract Classes that Custom Importers Should Inherit From.

class bank_import.importers.base.**BaseImporter**(*file_object*, **args*, ***kwargs*)
An abstract class for the import_bank_statement view.

Descendants of this class handle parsing the file_object and returning the parsed transaction dictionaries.

get_data()
Return the parsed data.

process_file(*file_object*)
Process the File & Return a list of Dictionaries.

class bank_import.importers.base.**CSVImporter**(*file_object*, **args*, ***kwargs*)
An abstact Bank Statement Importer to process CSV Files.

The file_object passed to the constructor should point to a CSV file whose first row are headers for the columns.

Implementations must specify two class attributes:

1. CSV_TO_DATA_FIELDS - a dictionary mapping from CSV column names to data field names. Required field names are date, amount, check_number, type`, & ``memo.

1. CSV_TYPE_TO_DATA_TYPE - a dictionary mapping from CSV type column values to data type values. Valid data type values are transfer_deposit, transfer_withdrawal`, ``deposit, or withdrawal.

An optional CSV_DATE_FORMAT class attribute may modify the format string used to parse the date field. By default, MM/DD/YYYY is expected.

process_file(*file_object*)
Read the CSV file and Return the Data using CSV_TO_DATA_FIELDS.

class bank_import.importers.base.**QFXImporter**(*file_object*, **args*, ***kwargs*)
A Bank Statement importer to process QFX files.

Bank-specific implementations of this importer may want to cleanup the memo field, by overriding the *clean_memo* function.

clean_memo(*memo*)
Clean the memo field.

process_file(*file_object*)
Read the QFX file & Return the standardized data.

importers.vcb Module

Implements the Importers for Virginia Community Bank.

class bank_import.importers.vcb.**CSVImporter**(*file_object*, **args*, ***kwargs*)
Specify the Field Conversion & Type Conversion for VCB CSV Exports.

get_data()
Reverse the data, imported lines are in descending order by date.

importers.city_first_dc Module

Implements the Importers for City First - Bank of DC.

class bank_import.importers.city_first_dc.**QFXImporter**(*file_object*, **args*, ***kwargs*)
Clean up the memo field.

CHAPTER 8

Appendix: Installlation & Deployment Guides

Slackware Linux

This section will guide you through installation and deployment on Slackware Linux. Slackware 14.0 was used, but it should be fairly portable between versions if you use your versions SlackBuilds.

User

Start off by creating a new user to store and serve the application:

```
useradd -m -s /bin/bash accounting
passwd accounting
```

Dependencies

Make sure your system is updated and install some basic dependencies from the Slackware repositories. Additional dependencies may be discovered when trying to build PostgreSQL and Memcached.

```
slackpkg update
slackpkg upgrade-all
slackpkg install libmpc mpfr readline zlib libxml2 libxslt openldap-client tcl httpd git
```

Mark `/etc/rc.d/rc.httpd` as executable so it will start at boot:

```
chmod a+x /etc/rc.d/rc.httpd
```

We can then proceed to installing the PostgreSQL database, uWSGI and Memcached via SlackBuilds.

PostgreSQL

First we must create a postgres user and group. ID 209 is used to avoid conflicts with other SlackBuilds:

```
groupadd -g 209 postgres
usereadd -u 209 -g 209 -d /var/lib/pgsql postgres
```

We can then download and extract the SlackBuild:

```
wget http://slackbuilds.org/slackbuilds/14.0/system/postgresql.tar.gz
tar xvfz postgresql.tar.gz
```

We need to manually download the source code to the extracted directory, then we can compile it into a package:

```
cd postgresql
wget ftp://ftp.postgresql.org/pub/source/v9.3.0/postgresql-9.3.0.tar.bz2
./postgresql.SlackBuild
```

Now we can install the new package:

```
installpkg /tmp/postgresql-*.tgz
```

The database files need to be initialized:

```
su postgres -c "initdb -D /var/lib/pgsql/9.3/data"
```

We should make the rc.d script executable and fire up PostgreSQL:

```
chmod a+x /etc/rc.d/rc.postgresql
/etc/rc.d/rc.postgresql start
```

We will then edit `/etc/rc.d/rc.local` and `/etc/rc.d/rc.local_shutdown`, making sure it is started at boot and shutdown cleanly.

```
# rc.local
# Startup postgresql
if [ -x /etc/rc.d/rc.postgresql ]; then
    /etc/rc.d/rc.postgresql start
fi

# rc.local_shutdown
#!/bin/sh
# Stop postgres
if [ -x /etc/rc.d/rc.postgresql ]; then
    /etc/rc.d/rc.postgresql stop
fi
```

Memcached

Memcached requires you to build the `libevent` and `libmemcached` SlackBuilds first:

```
wget http://slackbuilds.org/slackbuilds/14.0/libraries/libevent.tar.gz
tar xvfz libevent.tar.gz
cd libevent
wget https://github.com/downloads/libevent/libevent/libevent-2.0.21-stable.tar.gz
./libevent.SlackBuild
installpkg /tmp/libevent-*.tgz

cd ..
wget http://slackbuilds.org/slackbuilds/14.0/libraries/libmemcached.tar.gz
tar xvfz libmemcached.tar.gz
cd libmemcached
wget https://launchpad.net/libmemcached/1.0/1.0.15/+download/libmemcached-1.0.15.tar.gz
./libmemcached.SlackBuild
installpkg /tmp/libmemcached-*.tgz
```

You can build and install Memcached the same way:

```
cd ..
wget http://slackbuilds.org/slackbuilds/14.0/network/memcached.tar.gz
tar xvfz memcached.tar.gz
```

```
cd memcached
wget http://memcached.googlecode.com/files/memcached-1.4.15.tar.gz
./memcached.SlackBuild
installpkg /tmp/memcached-*.tgz
```

Add the following line to `/etc/rc.d/rc.local` in order to get Memcached to start at boot:

```
# /etc/rc.d/rc.local
memcached -d 127.0.0.1 -u accounting
```

Keep in mind the default port is `11211`.

uWSGI

Again, download the SlackBuild and source, compile and install the package:

```
cd ..
wget http://slackbuilds.org/slackbuilds/14.1/network/uwsgi.tar.gz
tar xvfz uwsgi.tar.gz
cd uwsgi
wget http://projects.unbit.it/downloads/uwsgi-1.9.6.tar.gz
./uwsgi.SlackBuild
installpkg /tmp/uwsgi-1.9.6-x86_64-1_SBo.tgz
```

We will also need to build the Apache module:

```
cd /tmp/SBo/uwsgi-1.9.6/apache2
sudo apxs -i -c mod_uwsgi.c
```

Edit `/etc/httpd/httpd.conf` to use the uWSGI module:

```
echo "LoadModule uwsgi_module lib64/httpd/modules/mod_uwsgi.so" >> /etc/httpd/httpd.conf
```

Pip and VirtualEnv

We will use pip and virtualenv to manage the python dependencies. Start off by downloading and running the pip install script:

```
wget https://raw.github.com/pypa/pip/master/contrib/get-pip.py
python get-pip.py
```

Then install virtualenv:

```
pip install virtualenv
```

Install the Accounting Application

Download Source Code

We are now ready to grab the source code from the git repository. Do this as the `accounting` user. We chose to store the local repository at `~/AcornAccounting/`:

```
su - accounting
git clone ssh://git@aphrodite.acorn:22/srv/git/AcornAccounting.git ~/AcornAccounting
```

Create a Virtual Environment

Before proceeding we should make a VirtualEnv for the `accounting` user:

```
virtualenv ~/AccountingEnv
```

Activate the VirtualEnv by sourcing the `activate` script:

```
source ~/AccountingEnv/bin/activate
```

Install Python Dependencies

We can now install the python dependencies into our VirtualEnv:

```
cd AcornAccounting
pip install -r requirements/production.txt
```

Create a PostgreSQL User and Database

We need a database to store our data, and a user that is allowed to access it, we decided to name both `accounting`:

```
su - postgres
createuser -DERPS accounting
createdb accounting -O accounting
exit
```

You can now sync and migrate the database, creating the necessary schema:

```
# Fill in the following variables according to your setup
export DJANGO_SETTINGS_MODULE=accounting.settings.production
export DJANGO_SECRET_KEY=<your unique secret key>
export DB_HOST=localhost
export DB_USER=accounting
export DB_PASSWORD=<accounting user password>
export DB_NAME=accounting

cd ~/AcornAccounting/acornaccounting
python manage.py syncdb
python manage.py migrate
```

Note: If you already have a database dump in a `.sql` file, you may restore this into your new database by running the following:

```
psql -U accounting -d accounting -f database_dump.sql
```

You can test your installation by running the following, assuming you have set the environmental variables from above:

```
python manage.py runserver 0.0.0.0:8000
```

Deployment

Now that the application is installed and running, we will serve the files and pages using uWSGI and Apache. Apache will only be serving our static files.

The application will be served out of `/srv/accounting`, which should be a link to `/home/accounting/AcornAccounting/acornaccounting/`:

```
ln -s /home/accounting/AcornAccounting/acornaccounting/ /srv/accounting
```

Configure uWSGI

We will need two things to use uWSGI: a configuration file and an rc.d script for starting and stopping the uWSGI daemon.

We should start by creating a directory for our configuration file(you no longer need to be the `accounting` user):

```
mkdir /etc/uwsgi
```

Create the `/etc/uwsgi/accounting.ini` file containing the following configuration:

```
[uwsgi]
uid = accounting
gid = %(uid)
chdir = /srv/accounting/

plugin = python
pythonpath = %(chdir)
virtualenv = /home/accounting/AccountingEnv/
module = django.core.handlers.wsgi:WSGIHandler()

socket = 127.0.0.1:3031
master = true
workers = 10
max-requests = 5000
vacuum = True

daemonize = /var/log/accounting/uwsgi.log
pidfile = /var/run/accounting.pid
touch-reload = /tmp/accounting.touch

env = DJANGO_SETTINGS_MODULE=accounting.settings.production
env = DB_NAME=accounting
env = DB_PASSWORD=
env = DB_USER=accounting
env = DB_HOST=
env = DB_PORT=
env = DJANGO_SECRET_KEY=
env = CACHE_LOCATION=127.0.0.1:11211
```

Note: If you do not have a secure, unique secret key, you may generate one by running the following in the Python interpreter:

```
import random
print(''.join(
    [random.SystemRandom().choice(
        'abcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*(-_=+)')
     for i in range(50)])
)
```

We'll need to make some of the folders specified in the config:

```
mkdir /var/log/accounting
chown accounting /var/log/accounting
mkdir /var/run/uwsgi
```

Now we can make an rc.d script at `/etc/rc.d/rc.accounting` to let us start and stop the server:

```
#!/bin/bash
#
# Start/Stop/Restart the Accounting uWSGI server
#
# To make the server start at boot make this file executable:
#
#       chmod 755 /etc/rc.d/rc.accounting

INIFILE=/etc/uwsgi/accounting.ini
PIDFILE=/var/run/accounting.pid

case "$1" in
    'start')
        echo "Starting the Accounting uWSGI Process."
        uwsgi -i $INIFILE
        ;;
    'stop')
        echo "Stopping the Accounting uWSGI Process."
        uwsgi --stop $PIDFILE
        rm $PIDFILE
        ;;
    'restart')
        echo "Restarting the Accounting uWSGI Process."
        if [ -f $PIDFILE ]; then
            uwsgi --reload $PIDFILE
        else
            echo "Error: No Accounting uWSGI Process Found."
        fi
        ;;
    *)
        echo "Usage: /etc/rc.d/rc.accounting {start|stop|restart}"
        exit 1
        ;;
esac

exit 0
```

We need to give this the correct permissions to enable it:

```
chmod 755 /etc/rc.d/rc.accounting
/etc/rc.d/rc.accounting start
```

Make sure this has started the application and spawned uWSGI workers by checking the log:

```
less /var/log/accounting/uwsgi.log
```

We can automatically start the process from `rc.local` and stop it from `rc.local_shutdown`:

```
# /etc/rc.d/rc.local
if [ -x /etc/rc.d/rc.accounting ]; then
    /etc/rc.d/rc.accounting start
fi
```

```
# /etc/rc.d/rc.local_shutdown
if [ -x /etc/rc.d/rc.accounting ]; then
    /etc/rc.d/rc.accounting stop
fi
```

Configuring Apache

Apache will serve any files under the `/static/` directory, passing all other requests to uWSGI.

First we should collect all the static files into the appropriate directory:

```
su - accounting
source AccountingEnv/bin/activate
cd AcornAccounting/acornaccounting
python manage.py collectstatic
```

Now we can create a virtual host in `/etc/httpd/extra/httpd-accounting.conf` to hold the configuration:

```
<VirtualHost *:80>
    ServerName accounting.yourdomain.com
    ErrorLog "/var/log/httpd/accounting-error_log"
    CustomLog "/var/log/httpd/accounting-access_log" common
    DocumentRoot "/srv/accounting/"
    <Directory "/srv/accounting/">
        Options Indexes FollowSymLinks MultiViews
        AllowOverride None
        Require all granted
    </Directory>
    <Location />
        Options FollowSymLinks Indexes
        SetHandler uwsgi-handler
        uWSGISocket 127.0.0.1:3031
    </Location>

    Alias /static /srv/accounting/static/
    <Location /static>
        SetHandler none
    </Location>

    Alias /media/uploads /srv/accounting/uploads/
    <Location /media/uploads>
        SetHandler none
    </Location>
</VirtualHost>
```

Include this configuration file in `/etc/httpd/httpd.conf`:

```
echo "Include /etc/httpd/extra/httpd-accounting.conf" >> /etc/httpd/httpd.conf
```

Then restart apache:

```
/etc/rc.d/rc.httpd restart
```

The application should now be accessible at `http://accounting.yourdomain.com`

You can restart the uWSGI server by touching `/tmp/accounting.touch`:

```
touch /tmp/accounting.touch
```

CHAPTER 9

Glossary

ACH Automated Clearing House. An electronic network for credit and debit Transactions, such as on-line bill payments or income from credit cards. ACH Transactions do not have check numbers associated with them.

API Application Programming Interface. Specifies how software components interact with each other. An API documents and specifies program Routines, Data Structures, Object Classes and Variables.

Wireframe A visual guide that represents the skeletal framework of a website. A wireframe displays the overall page layout and the arrangement of its elements. Wireframes may not depict the final style, color or graphics of a website. Also known as Page Schematics or Screen Blueprints.

CHAPTER 10

Indices and tables

- *genindex*
- *modindex*
- *search*

a

accounts.managers, 53
accounts.models, 50
accounts.signals, 53

b

bank_import.forms, 69
bank_import.importers.base, 70
bank_import.importers.city_first_dc, 71
bank_import.importers.vcb, 71
bank_import.models, 68
bank_import.views, 70

c

core.context_processors, 47
core.core, 47
core.forms, 47
core.management.commands.generatedata, 49
core.management.commands.import_myob, 50
core.middleware, 48
core.templatetags.core_filters, 50
core.templatetags.core_tags, 50
core.views, 48
creditcards.admin, 66
creditcards.forms, 66
creditcards.models, 65
creditcards.views, 67

e

entries.forms, 57
entries.managers, 56
entries.views, 59
events.models, 63
events.views, 64

f

fiscalyears.fiscalyears, 61
fiscalyears.forms, 61
fiscalyears.models, 61
fiscalyears.views, 62

r

reports.views, 64

t

trips.admin, 68
trips.forms, 68
trips.models, 67
trips.views, 68

A

account (accounts.models.HistoricalAccount attribute), 52
account (bank_import.models.BankAccount attribute), 68
Account (class in accounts.models), 50
account (creditcards.models.CreditCard attribute), 65
account (creditcards.models.CreditCardTransaction attribute), 66
account (entries.forms.BaseBankForm attribute), 58
account (Transaction attribute), 53
AccountManager (class in accounts.managers), 53
accounts.managers (module), 53
accounts.models (module), 50
accounts.signals (module), 53
ACH, **81**
ach_payment (BankSpendingEntry attribute), 55
active() (accounts.managers.AccountManager method), 53
add_bank_entry() (in module entries.views), 59
add_creditcard_entry() (in module creditcards.views), 67
add_fiscal_year() (in module fiscalyears.views), 62
add_journal_entry() (in module entries.views), 59
add_transfer_entry() (in module entries.views), 60
add_trip_entry() (in module trips.views), 68
AddApprovableEntryView (class in core.views), 48
AddCreditCardEntry (class in creditcards.views), 67
AddTripEntryView (class in trips.views), 68
amount (accounts.models.HistoricalAccount attribute), 52
amount (creditcards.models.CreditCardEntry attribute), 65
amount (entries.forms.BaseBankForm attribute), 58
amount (entries.forms.TransferForm attribute), 59
API, **81**
approve_entry() (creditcards.models.CreditCardEntry method), 66
approve_entry() (trips.models.TripEntry method), 67
args (core.management.commands.generatedata.Command attribute), 49
args (core.management.commands.import_myob.Command attribute), 50

B

balance_delta (creditcards.models.CreditCardTransaction attribute), 66
balance_delta (Transaction attribute), 54
bank (bank_import.models.BankAccount attribute), 69
bank_account (bank_import.models.CheckRange attribute), 69
bank_import.forms (module), 69
bank_import.importers.base (module), 70
bank_import.importers.city_first_dc (module), 71
bank_import.importers.vcb (module), 71
bank_import.models (module), 68
bank_import.views (module), 70
BankAccount (class in bank_import.models), 68
BankAccountForm (class in bank_import.forms), 69
bankreceive_entry (Transaction attribute), 53
BankReceivingEntry (class in entries.models), 56
BankReceivingForm (class in entries.forms), 57
BankReceivingTransactionFormSet (in module entries.forms), 57
bankspend_entry (Transaction attribute), 53
BankSpendingEntry (class in entries.models), 55
BankSpendingForm (class in entries.forms), 57
BankSpendingTransactionFormSet (in module entries.forms), 57
BankTransactionForm (class in entries.forms), 58
BaseAccountModel (class in accounts.models), 51
BaseBankForm (class in entries.forms), 58
BaseBankTransactionFormSet (class in entries.forms), 58
BaseEntryForm (class in entries.forms), 58
BaseEvent (class in events.models), 63
BaseEvent.Meta (class in events.models), 63
BaseImporter (class in bank_import.importers.base), 70
BaseJournalEntry (class in entries.models), 54
BaseTransactionFormSet (class in entries.forms), 58
BootstrapAuthenticationForm (class in core.forms), 47

C

capitalize_words() (in module

core.templatetags.core_filters), 50
card (creditcards.models.CreditCardEntry attribute), 65
check_number (BankSpendingEntry attribute), 55
CheckRange (class in bank_import.models), 69
city (events.models.BaseEvent attribute), 63
clean() (accounts.models.BaseAccountModel method), 51
clean() (core.forms.RequiredFormSetMixin method), 47
clean() (entries.forms.BankTransactionForm method), 58
clean() (entries.forms.BaseBankForm method), 58
clean() (entries.forms.BaseBankTransactionFormSet method), 58
clean() (entries.forms.BaseTransactionFormSet method), 58
clean() (entries.forms.TransactionForm method), 59
clean() (entries.forms.TransferForm method), 59
clean() (entries.models.BankSpendingEntry method), 56
clean() (entries.models.Transaction method), 54
clean() (fiscalyears.forms.FiscalYearForm method), 62
clean_amount() (entries.forms.BankReceivingForm method), 57
clean_date() (entries.forms.BaseEntryForm method), 58
clean_memo() (bank_import.importers.base.QFXImporter method), 70
clean_year() (fiscalyears.forms.FiscalYearForm method), 62
Command (class in core.management.commands.generatedata), 49
Command (class in core.management.commands.import_myob), 50
comments (BaseJournalEntry attribute), 55
comments (creditcards.models.CreditCardEntry attribute), 65
core.context_processors (module), 47
core.core (module), 47
core.forms (module), 47
core.management.commands.generatedata (module), 49
core.management.commands.import_myob (module), 50
core.middleware (module), 48
core.templatetags.core_filters (module), 50
core.templatetags.core_tags (module), 50
core.views (module), 48
created_at (BaseJournalEntry attribute), 55
created_at (creditcards.models.CreditCardEntry attribute), 65
credit_total (events.models.HistoricalEvent attribute), 63
CreditCard (class in creditcards.models), 65
creditcard_entry (creditcards.models.CreditCardReceipt attribute), 66
creditcard_entry (creditcards.models.CreditCardTransaction attribute), 66
CreditCardEntry (class in creditcards.models), 65
CreditCardEntryForm (class in creditcards.forms), 66
CreditCardReceipt (class in creditcards.models), 66
creditcards.admin (module), 66
creditcards.forms (module), 66
creditcards.models (module), 65
creditcards.views (module), 67
CreditCardTransaction (class in creditcards.models), 66
CSVImporter (class in bank_import.importers.base), 70
CSVImporter (class in bank_import.importers.vcb), 71
currency() (in module core.templatetags.core_filters), 50

D

date (accounts.models.HistoricalAccount attribute), 52
date (BaseJournalEntry attribute), 55
date (creditcards.models.CreditCardEntry attribute), 65
date (events.models.BaseEvent attribute), 63
date (fiscalyears.models.FiscalYear attribute), 61
date (Transaction attribute), 54
date_range_form() (in module core.templatetags.core_tags), 50
DateRangeForm (class in core.forms), 47
debit_total (events.models.HistoricalEvent attribute), 63
default_account (bank_import.models.CheckRange attribute), 69
default_memo (bank_import.models.CheckRange attribute), 69
default_payee (bank_import.models.CheckRange attribute), 69
delete() (accounts.models.BaseAccountModel method), 51
destination (entries.forms.TransferForm attribute), 59
detail (creditcards.models.CreditCardTransaction attribute), 66
detail (entries.forms.TransferForm attribute), 59
detail (Transaction attribute), 54

E

end_month (fiscalyears.models.FiscalYear attribute), 61
end_number (bank_import.models.CheckRange attribute), 69
entries.forms (module), 57
entries.managers (module), 56
entries.views (module), 59
entry_class (creditcards.views.AddCreditCardEntry attribute), 67
entry_class (trips.views.AddTripEntryView attribute), 68
entry_form_class (creditcards.views.AddCreditCardEntry attribute), 67
entry_form_class (trips.views.AddTripEntryView attribute), 68
Event (class in events.models), 63
event (Transaction attribute), 54
events.models (module), 63
events.views (module), 64

events_report() (in module reports.views), 64

F

first_day_of_month() (in module core.core), 47
first_day_of_year() (in module core.core), 47
FiscalYear (class in fiscalyears.models), 61
FiscalYearAccountsForm (class in fiscalyears.forms), 61
FiscalYearAccountsFormSet (in module fiscalyears.forms), 62
FiscalYearForm (class in fiscalyears.forms), 62
fiscalyears.fiscalyears (module), 61
fiscalyears.forms (module), 61
fiscalyears.models (module), 61
fiscalyears.views (module), 62
flip_balance (accounts.models.HistoricalAccount attribute), 52

G

generate_memo() (creditcards.models.CreditCardEntry method), 66
get_absolute_url() (accounts.models.HistoricalAccount method), 52
get_absolute_url() (entries.models.BankReceivingEntry method), 56
get_absolute_url() (entries.models.BankSpendingEntry method), 56
get_absolute_url() (entries.models.BaseJournalEntry method), 55
get_absolute_url() (entries.models.JournalEntry method), 55
get_absolute_url() (entries.models.Transaction method), 54
get_absolute_url() (events.models.Event method), 63
get_account_balance() (accounts.models.Header method), 51
get_amount (accounts.models.HistoricalAccount attribute), 52
get_balance() (accounts.models.Account method), 50
get_balance_by_date (accounts.models.Account attribute), 51
get_balance_change_by_month() (accounts.models.Account method), 51
get_banks() (accounts.managers.AccountManager method), 53
get_data() (bank_import.importers.base.BaseImporter method), 70
get_data() (bank_import.importers.vcb.CSVImporter method), 71
get_edit_url() (creditcards.models.CreditCardEntry method), 66
get_edit_url() (entries.models.BankReceivingEntry method), 56
get_edit_url() (entries.models.BankSpendingEntry method), 56
get_edit_url() (entries.models.BaseJournalEntry method), 55
get_entry_number() (entries.models.Transaction method), 54
get_final_account_balance() (entries.models.Transaction method), 54
get_full_number() (accounts.models.BaseAccountModel method), 51
get_importer_class() (bank_import.models.BankAccount method), 69
get_initial_account_balance() (entries.models.Transaction method), 54
get_journal_entry() (entries.models.Transaction method), 54
get_memo() (entries.models.Transaction method), 54
get_net_change() (events.models.Event method), 63
get_next_entry() (creditcards.models.CreditCardEntry method), 66
get_next_entry() (trips.models.TripEntry method), 67
get_number (entries.models.BankReceivingEntry attribute), 56
get_number (entries.models.BankSpendingEntry attribute), 56
get_number() (creditcards.models.CreditCardEntry method), 66
get_number() (entries.models.BaseJournalEntry method), 55
get_number() (trips.models.TripEntry method), 67
get_query_set() (entries.managers.TransactionManager method), 57
get_random_account() (in module core.management.commands.generatedata), 49
get_random_bank() (in module core.management.commands.generatedata), 50
get_start_of_current_fiscal_year() (in module fiscalyears.fiscalyears), 61
get_totals() (entries.managers.TransactionManager method), 57
get_totals() (entries.managers.TransactionQuerySet method), 57

H

handle() (core.management.commands.generatedata.Command method), 49
handle() (core.management.commands.import_myob.Command method), 50
Header (class in accounts.models), 51
help (core.management.commands.generatedata.Command attribute), 49
help (core.management.commands.import_myob.Command attribute), 50
HistoricalAccount (class in accounts.models), 52
HistoricalEvent (class in events.models), 63

I

import_bank_statement() (in module bank_import.views), 70
ImportFormMixin (class in bank_import.forms), 69
in_fiscal_year() (entries.models.BaseJournalEntry method), 55

J

journal_entry (Transaction attribute), 53
journal_ledger() (in module entries.views), 60
JournalEntry (class in entries.models), 55
JournalEntryForm (class in entries.forms), 58

L

list_creditcard_entries() (in module creditcards.views), 67
list_entries() (in module core.views), 49
list_trip_entries() (in module trips.views), 68
LoginRequiredMiddleware (class in core.middleware), 48

M

main_transaction (BankReceivingEntry attribute), 56
main_transaction (BankSpendingEntry attribute), 56
memo (BaseJournalEntry attribute), 55
merchant (creditcards.models.CreditCardEntry attribute), 65
model (bank_import.forms.ReceivingImportForm.Meta attribute), 69
model (bank_import.forms.SpendingImportForm.Meta attribute), 69

N

name (accounts.models.HistoricalAccount attribute), 52
name (bank_import.models.BankAccount attribute), 68
name (creditcards.models.CreditCard attribute), 65
name (creditcards.models.CreditCardEntry attribute), 65
name (events.models.BaseEvent attribute), 63
net_change (events.models.HistoricalEvent attribute), 64
number (accounts.models.HistoricalAccount attribute), 52
number (events.models.BaseEvent attribute), 63

P

payee (BankSpendingEntry attribute), 56
payor (BankReceivingEntry attribute), 56
period (fiscalyears.models.FiscalYear attribute), 61
process_file() (bank_import.importers.base.BaseImporter method), 70
process_file() (bank_import.importers.base.CSVImporter method), 70
process_file() (bank_import.importers.base.QFXImporter method), 71
process_month_start_date_range_form() (in module core.core), 47
process_quick_search_form() (in module core.core), 48
process_request() (core.middleware.LoginRequiredMiddleware method), 48
process_year_start_date_range_form() (in module core.core), 48
profit_loss_report() (in module reports.views), 64
Python Enhancement Proposals
    PEP 20, 13
    PEP 257, 13, 17
    PEP 8, 12, 13

Q

QFXImporter (class in bank_import.importers.base), 70
QFXImporter (class in bank_import.importers.city_first_dc), 71
quick_search() (in module core.templatetags.core_tags), 50

R

receipt_class (creditcards.views.AddCreditCardEntry attribute), 67
receipt_class (trips.views.AddTripEntryView attribute), 68
receipt_file (creditcards.models.CreditCardReceipt attribute), 66
receipt_list() (in module core.templatetags.core_tags), 50
ReceivingImportForm (class in bank_import.forms), 69
ReceivingImportForm.Meta (class in bank_import.forms), 69
reconciled (Transaction attribute), 54
remove_trailing_zeroes() (in module core.core), 48
render() (core.views.AddApprovableEntryView method), 49
reports.views (module), 64
RequiredBaseFormSet (class in core.forms), 47
RequiredBaseInlineFormSet (class in core.forms), 47
RequiredFormSetMixin (class in core.forms), 47

S

save() (accounts.models.BaseAccountModel method), 51
save() (bank_import.forms.ReceivingImportForm method), 69
save() (bank_import.forms.SpendingImportForm method), 70
save() (bank_import.forms.TransferImportForm method), 70
save() (entries.forms.BaseBankForm method), 58
save() (entries.models.BankReceivingEntry method), 56
save() (entries.models.BankSpendingEntry method), 56
save() (entries.models.JournalEntry method), 55
save() (entries.models.Transaction method), 54
save() (events.models.Event method), 63
save() (fiscalyears.models.FiscalYear method), 61
show_bank_entry() (in module entries.views), 60

show_creditcard_entry() (in module creditcards.views), 67
show_event_detail() (in module events.views), 64
show_journal_entry() (in module entries.views), 60
show_single_entry() (in module core.views), 49
show_trip_entry() (in module trips.views), 68
source (entries.forms.TransferForm attribute), 59
SpendingImportForm (class in bank_import.forms), 69
SpendingImportForm.Meta (class in bank_import.forms), 69
start_number (bank_import.models.CheckRange attribute), 69
state (events.models.BaseEvent attribute), 63
StoreAccount (class in trips.models), 67

T

tab_number() (in module core.templatetags.core_filters), 50
template_accessible_settings() (in module core.context_processors), 47
today_in_american_format() (in module core.core), 48
Transaction (class in entries.models), 53
transaction_delete() (in module accounts.signals), 53
transaction_postsave() (in module accounts.signals), 53
transaction_presave() (in module accounts.signals), 53
TransactionForm (class in entries.forms), 58
TransactionFormSet (in module entries.forms), 59
TransactionManager (class in entries.managers), 56
TransactionQuerySet (class in entries.managers), 57
TransferForm (class in entries.forms), 59
TransferFormSet (in module entries.forms), 59
TransferImportForm (class in bank_import.forms), 70
trial_balance_report() (in module reports.views), 65
TripEntry (class in trips.models), 67
TripEntryForm (class in trips.forms), 68
TripReceipt (class in trips.models), 67
trips.admin (module), 68
trips.forms (module), 68
trips.models (module), 67
trips.views (module), 68
TripStoreTransaction (class in trips.models), 67
TripTransaction (class in trips.models), 68
type (accounts.models.HistoricalAccount attribute), 52

U

updated_at (BaseJournalEntry attribute), 55

V

void (BankSpendingEntry attribute), 56

W

Wireframe, **81**

Y

year (fiscalyears.models.FiscalYear attribute), 61