SEED Platform Documentation

Release 1.3.2

Mar 02, 2017

Contents

The Standard Energy Efficiency Data (SEED) Platform™ is a web-based application that helps organizations easily manage data on the energy performance of large groups of buildings. Users can combine data from multiple sources, clean and validate it, and share the information with others. The software application provides an easy, flexible, and cost-effective method to improve the quality and availability of data to help demonstrate the economic and environmental benefits of energy efficiency, to implement programs, and to target investment activity.

The SEED application is written in Python/Django, with AngularJS, Bootstrap, and other JavaScript libraries used for the front-end. The back-end database is required to be PostgreSQL.

The SEED web application provides both a browser-based interface for users to upload and manage their building data, as well as a full set of APIs that app developers can use to access these same data management functions.

Work on SEED Platform is managed by the National Renewable Energy Laboratory, with funding from the U.S. Department of Energy.

CHAPTER 1

Getting Started

Development Setup

Installation on OSX

These instructions are for installing and running SEED on Mac OSX in development mode.

Quick Installation Instructions

This section is intended for developers who may already have their machine ready for general development. If this is not the case, skip to Prerequisites.

- install Postgres 9.4 and redis for cache and message broker
- use a virtualenv (if desired)
- *git clone git@github.com:seed-platform/seed.git*
- create a *local_untracked.py* in the *config/settings* folder and add CACHE and DB config (example *local_untracked.py.dist*)
- *export DJANGO_SETTINGS_MODULE=config.settings.dev*
- *pip install -r requirements/local.txt*
- *./manage.py migrate*
- *./manage.py create_default_user*
- *./manage.py runserver*
- *celery -A seed worker -l info -c 4 –maxtasksperchild 1000 –events*
- navigate to *http://127.0.0.1:8000/app/#/profile/admin* in your browser to add users to organizations
- main app runs at *127.0.0.1:8000/app*

The *python manage.py create_default_user* will setup a default *superuser* which must be used to access the system the first time. The management command can also create other superusers.

```
./manage.py create_default_user --username=demo@seed.lbl.gov --organization=lbl --
↪password=demo123
```

Prerequisites

These instructions assume you have MacPorts or Homebrew. Your system should have the following dependencies already installed:

- git (*port install git* or *brew install git*)
- Mercurial (*port install hg* or *brew install mercurial*)
- graphviz (*brew install graphviz*)
- virtualenv and virtualenvwrapper (Recommended)

> **Note:** Although you *could* install Python packages globally, this is the easiest way to install Python packages. Setting these up first will help avoid polluting your base Python installation and make it much easier to switch between different versions of the code.

```
pip install virtualenv
pip install virtualenvwrapper
```

- Follow instructions on virtualenvwrapper to setup your environment.
- Once you have these installed, creating and entering a new virtualenv called "`seed`" for SEED development is by calling:

```
mkvirtualenv --python=python2.7 seed
```

PostgreSQL 9.4

MacPorts:

```
sudo su - root
port install postgresql94-server postgresql94 postgresql94-doc
# init db
mkdir -p /opt/local/var/db/postgresql94/defaultdb
chown postgres:postgres /opt/local/var/db/postgresql94/defaultdb
su postgres -c '/opt/local/lib/postgresql94/bin/initdb -D /opt/local/var/db/
↪postgresql94/defaultdb'

# At this point, you may want to add start/stop scripts or aliases to
# ~/.bashrc or your virtualenv ``postactivate`` script
# (in ``~/.virtualenvs/{env-name}/bin/postactivate``).

alias pg_start='sudo su postgres -c "/opt/local/lib/postgresql94/bin/pg_ctl \
    -D /opt/local/var/db/postgresql94/defaultdb \
    -l /opt/local/var/db/postgresql94/defaultdb/postgresql.log start"'
alias pg_stop='sudo su postgres -c "/opt/local/lib/postgresql94/bin/pg_ctl \
    -D /opt/local/var/db/postgresql94/defaultdb stop"'

pg_start

sudo su - postgres
PATH=$PATH:/opt/local/lib/postgresql94/bin/
```

Homebrew:

```
brew install postgres
# follow the post install instructions to add to launchagents or call
# manually with `postgres -D /usr/local/var/postgres`
# Skip the remaining Postgres instructions!
```

Configure PostgreSQL. Replace 'seeddb', 'seeduser' with desired db/user. By default use password *seedpass* when prompted

```
createuser -P seeduser
createdb `whoami`
psql -c 'CREATE DATABASE "seeddb" WITH OWNER = "seeduser";'
psql -c 'GRANT ALL PRIVILEGES ON DATABASE "seeddb" TO seeduser;'
psql -c 'ALTER USER seeduser CREATEDB;'
psql -c 'ALTER USER seeduser CREATEROLE;'
```

Now exit any root environments, becoming just yourself (even though it's not that easy being green), for the remainder of these instructions.

Python Packages

Run these commands as your normal user id.

Change to a virtualenv (using virtualenvwrapper) or do the following as a superuser. A virtualenv is usually better for development. Set the virtualenv to seed.

```
workon seed
```

Make sure PostgreSQL command line scripts are in your PATH (if using port)

```
export PATH=$PATH:/opt/local/lib/postgresql94/bin
```

Some packages (uWSGI) may need to find your C compiler. Make sure you have 'gcc' on your system, and then also export this to the *CC* environment variable:

```
export CC=gcc
```

Install requirements with *pip*

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

Install library with *setup.py*

```
python setup.py install
```

NodeJS/npm

Install npm. You can do this by installing from nodejs.org, MacPorts, or Homebrew:

MacPorts:

```
sudo port install npm
```

Homebrew:

```
brew install npm
```

Configure Django and Databases

In the *config/settings* directory, there must be a file called *local_untracked.py* that sets up databases and a number of other things. To create and edit this file, start by copying over the template

```
cd config/settings
cp local_untracked.py.dist local_untracked.py
```

Edit *local_untracked.py*. Open the file you created in your favorite editor. The PostgreSQL config section will look something like this:

```
# postgres DB config
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'seeddb',
        'USER': 'seeduser',
        'PASSWORD': 'seedpass',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

You may want to comment out the AWS settings.

For Redis, edit the *CACHES* and *BROKER_URL* values to look like this:

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.cache.RedisCache',
        'LOCATION': "127.0.0.1:6379",
        'OPTIONS': {'DB': 1},
        'TIMEOUT': 300
    }
}
BROKER_URL = 'redis://127.0.0.1:6379/1'
```

Run Django Migrations

Change back to the root of the repository. Now run the migration script to set up the database tables

```
export DJANGO_SETTINGS_MODULE=config.settings.dev
./manage.py migrate
```

Django Admin User

You need a Django admin (super) user.

```
./manage.py create_default_user --username=admin@my.org --organization=lbnl --
↪password=badpass
```

Of course, you need to save this user/password somewhere, since this is what you will use to login to the SEED website.

If you want to do any API testing (and of course you do!), you will need to add an API KEY for this user. You can do this in postgresql directly:

```
psql seeddb seeduser
seeddb=> update landing_seeduser set api_key='DEADBEEF' where id=1;
```

The 'secret' key DEADBEEF is hard-coded into the test scripts.

Install Redis

You need to manually install Redis for Celery to work.

MacPorts:

```
sudo port install redis
```

Homebrew:

```
brew install redis
# follow the post install instructions to add to launchagents or
# call manually with `redis-server`
```

Install JavaScript Dependencies

The JS dependencies are installed using node.js package management (npm), with a helper package called *bower*.

```
./bin/install_javascript_dependencies.sh
```

Start the Server

You should put the following statement in ~/.bashrc or add it to the virtualenv post-activation script (e.g., in *~/.virtualenvs/seed/bin/postactivate*).

```
export DJANGO_SETTINGS_MODULE=config.settings.dev
```

The combination of Redis, Celery, and Django have been encapsulated in a single shell script, which examines existing processes and doesn't start duplicate instances:

```
./bin/start-seed.sh
```

When this script is done, the Django stand-alone server will be running in the foreground.

Login

Open your browser and navigate to http://127.0.0.1:8000

Login with the user/password you created before, e.g., *admin@my.org* and *badpass*.

Note: these steps have been combined into a script called *start-seed.sh*. The script will also try to not start Celery or Redis if they already seem to be running.

Todo

These instructions are out of date and will be updated soon.

Installation on Docker

Installing on Windows and OS X

Install Docker Toolbox (Windows/OS X)

Install Docker-Toolbox, which installs several applications including Docker, Docker Machine, and Docker Compose. It is possible to install these individually as well by installing Docker, Docker-Machine, and Docker-Compose.

Create Docker-Machine Image

The command below will create a 100GB volume for development. This is a very large volume and can be adjusted. Make sure to create a volume greater than 30GB.

```
docker-machine create --virtualbox-disk-size 100000 -d virtualbox dev
```

Start Docker-Machine Image

```
docker-machine start dev  # if not already running
eval $(docker-machine env dev)
```

Run Docker Compose

```
docker-compose build
```

Be Patient ... If the containers build successfully, then start the containers

```
docker-compose up
```

Note that you may need to build the containers a couple times for everything to converge

Create User

```
docker-compose run web ./manage.py create_default_user
```

Login

Get the Docker IP address (*docker-machine ip dev*) and point your browser at http://*ip-address*:8000 and log in with the account:

```
username: demo@seed.lbl.gov
password: demo
```

Installing Docker on Ubuntu

Docker will run directly on Ubuntu, so, in contrast to Windows/OS X there is no need to run in in Virtual Box, unless desired. Running without Virtual Box means you can set it up so that local changes are reflected in the containers and you can edit files etc with your normal setup.

First follow the instructions here: https://docs.docker.com/engine/installation/linux/ubuntulinux/

If you set up a docker group and add yourself to it you can run docker commands directly. Otherwise you will need to precede docker commands with sudo. You will need to log out entirely for the changes to take place. You can test this by running

```
docker run hello-world
```

if you still have issues, try rebooting.

If you ran the hello-word docker you can use the following to clean up. First check to see what existing containers there are, and what there status is:

```
docker ps -a
```

You should see something similar to this if there are no containers.

:: CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

If you ran the hello world container you should see it listed (the image is hello-word).Check its status to make sure it exited. Then you can go ahead and remove it. A quick way to remove all old containers is this.

```
docker rm $(docker ps -a -q)
```

Otherwise specify the numeric id to remove individual containers.

Next you can list images in a similar way.

```
docker images
```

Images not connected to a container are known as dangling images. You can get rid of them using this command:

```
docker rmi -f $(docker images -q -a -f dangling=true)
```

Otherwise they can be removed using *docker rmi image* using the image name or id shown by docker images.

Install Docker Compose

```
sudo apt-get install python-pip
sudo pip install docker-compose
```

Optionally install Virtual Box and Docker-Machine

This is only necessary if you want to run inside Virtual Box.

```
sudo apt-get install virtual-box
wget https://github.com/docker/machine/releases/download/v0.7.0/docker-machine-`uname␣
↪-s`-`uname -m`
sudo mv docker-$(uname -s)-$(uname -m) /usr/local/bin/docker-machine
sudo chmod +x /usr/local/bin/docker-machine
```

If you do this proceed by following the instructions for Windows/OS X starting from Create Docker-Machine Image.

Setting up without a Virtual Machine

If we don't use Virtual Box we can run Docker directly. This assumes you are in the same directory as the Git repo. You should also set up a virtualenv for it. Setting it up this way means it will use your local ip, so you will be able to access the SEED website via localhost. As we are using containers we don't have to worry about setting up the database and Redis directly, Docker will do this for us.

In this part we are going to set up the project so that the seed directory in web container's root file system points to the copy on your local file system (i.e. the directory with the repo in it). This is an advantage of running docker directly: changes on your local file system show up in the container so you can edit with your local tools etc. without having to have them running in the container.

Before you start ensure you have set up a virtualenv for the project. Then at a minimum you will need to install the tos module manually.

```
pip install -e  'git+https://github.com/revsys/django-tos.
↪git@aca823ccd12fdb897b2827832458b3c34e91dee6#egg=django_tos-master'
```

Note the quotes.

If you notice complaints about this not being present try: *pip install ip install -r requirements/base.txt*, you might also need to install test and local

Edit *docker-compose.yml* in the repo base.

Look for the section web:, then underneath it the volumes: section. Add two lines like this:

- $HOME/projects/seed:/seed - $HOME/.virtualenvs/seed/src/django-tos-master:/seed/src/django-tos-master

You will to change the part before the colon to match your local setup. On my system the repo is a directory called seed under the projects folder in my home directory for the first line. In the second line my virtualenvs live under .virtualenvs in my home directory as I use virtualenv wrapper. You will need to adjust this to match your local setup.

Then you will need to open the ports for Redis and PostgreSQL. In the section *db-postgres:* add

::

ports:

- "5432:5432"

in db-redis add

::

ports:

- "6379:6379"

You should be careful not to add the changes to this file to your git commits as it is local only. You can do this with the following command.

```
git update-index  --skip-worktree docker-compose.yml
```

Doing this ensures git preserves your local changes and will warn you of any conflicts caused by upstream changes. Occasionally it might be necessary to temporarily unset the flag using *–no-skip-worktree* (you can reset it afterwards). You can find more on how acts, and how to fix conflicts here: http://fallengamer.livejournal.com/93321.html

Next do the following to create a local settings file

```
cp config/settings/local_untracked.py.dist config/settings/local_untracked.py
```

Then you will need to edit the databases section. Here is a sample

```
# postgres DB config
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'seed',
        'USER': 'seed',
        'PASSWORD': 'seed',
        'HOST': '172.17.0.1',
        'PORT': '5432',
    }
}
```

The tricky part is the HOST line. The web server can't connect to the database on localhost with this setup. Use *ifconfig* to find out the ip addresses on your local machine. In this example 172.17.0.1 was listed for docker0 and that worked.

Run Docker Compose

```
docker-compose build
```

Note this process will spit out a warning that some containers are being ignored. Don't worry they will be set up later.

Be Patient ... If the containers build successfully, then start the containers

```
docker-compose up
```

Note that you may need to build the containers a couple times for everything to converge. You will likely need to do this. Run *docker-compose up* hit Ctrl-C, then run both the commands again to get everything working correctly.

Note for whatever reason things like collectstatic aren't run automatically if you aren't using Virtual Box. You can fix it with the following. Use this to connect to a shell in the container.

```
docker exec -it "seed_web_1" bash
```

Then run the following when you are there.

```
bin/postcompile
```

You might see some errors, don't worry, these mostly occur because its trying to use Amazon S3., which is not relevant here.

This should only need to be done once (unless things change, e.g. adding more static files) as long as the docker image is around.

Create User

```
docker-compose run web ./manage.py create_default_user
```

Login

Point your browser at [http://127.0.0.1:8000{]}(http://127.0.0.1:8000) and log in with the account:

```
username: demo@seed.lbl.gov
password: demo
```

CHAPTER 2

Deployment Guide

SEED is intended to be installed on Linux instances in the cloud (e.g. AWS), and on local hardware. SEED Platform does not officially support Windows for production deployment. If this is desired, see the Django notes.

AWS Setup

Amazon Web Services (AWS) provides the preferred hosting for the SEED Platform.

seed is a Django Project and Django's documentation is an excellent place for general understanding of this project's layout.

Prerequisites

Ubuntu server 14.04 or newer.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install -y libpq-dev python-dev python-pip libatlas-base-dev \
gfortran build-essential g++ npm libxml2-dev libxslt1-dev git mercurial \
libssl-dev curl uwsgi-core uwsgi-plugin-python
```

PostgreSQL and Redis are not included in the above commands. For a quick installation on AWS it is okay to install PostgreSQL and Redis locally on the AWS instance. If a more permanent and scalable solution, it is recommended to use AWS's hosted Redis (ElastiCache) and PostgreSQL service.

Note: postgresql >=9.4 is required to support **'JSON Type'_**

```
# To install PostgreSQL and Redis locally
sudo apt-get install redis-server
sudo apt-get install postgresql postgresql-contrib
```

Amazon Web Services (AWS) Dependencies

The following AWS services are used for **SEED**:

- RDS (PostgreSQL >=9.4)
- ElastiCache (redis)

- SES

Python Dependencies

Clone the **SEED** repository from **github**

```
$ git clone git@github.com:SEED-platform/seed.git
```

enter the repo and install the python dependencies from requirements

```
$ cd seed
$ sudo pip install -r requirements/local.txt
```

JavaScript Dependencies

`npm` is required to install the JS dependencies. The `bin/install_javascript_dependencies.sh` script will download all JavaScript dependencies and build them. `bower` and `gulp` should be installed globally for convenience.

```
$ sudo apt-get install build-essential
$ sudo apt-get install curl
```

```
$ sudo npm install -g bower gulp
$ bin/install_javascript_dependencies.sh
```

Database Configuration

Copy the `local_untracked.py.dist` file in the `config/settings` directory to `config/settings/local_untracked.py`, and add a `DATABASES` configuration with your database username, password, host, and port. Your database configuration can point to an AWS RDS instance or a PostgreSQL 9.4 database instance you have manually installed within your infrastructure.

```
# Database
DATABASES = {
    'default': {
        'ENGINE':'django.db.backends.postgresql_psycopg2',
        'NAME': 'seed',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

In the above database configuration, `seed` is the database name, this is arbitrary and any valid name can be used as long as the database exists.

create the database within the postgres `psql` shell:

```
CREATE DATABASE seed;
```

or from the command line:

```
createdb seed
```

create the database tables and migrations:

```
python manage.py syncdb
python manage.py migrate
```

create a superuser to access the system

```
$ python manage.py create_default_user --username=demo@example.com --
↪organization=example --password=demo123
```

Note: Every user must be tied to an organization, visit `/app/#/profile/admin` as the superuser to create parent organizations and add users to them.

Cache and Message Broker

The SEED project relies on redis for both cache and message brokering, and is available as an AWS ElastiCache service. `local_untracked.py` should be updated with the `CACHES` and `BROKER_URL` settings.

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.cache.RedisCache',
        'LOCATION': "seed-core-cache.ntmprk.0001.usw2.cache.amazonaws.com:6379",
        'OPTIONS': { 'DB': 1 },
        'TIMEOUT': 300
    }
}
BROKER_URL = 'redis://seed-core-cache.ntmprk.0001.usw2.cache.amazonaws.com:6379/1'
```

Running Celery the Background Task Worker

Celery is used for background tasks (saving data, matching, creating projects, etc) and must be connected to the message broker queue. From the project directory, `celery` can be started:

```
celery -A seed worker -l INFO -c 2 -B --events --maxtasksperchild 1000
```

Running the Development Web Server

The Django dev server (not for production use) can be a quick and easy way to get an instance up and running. The dev server runs by default on port 8000 and can be run on any port. See Django's runserver documentation for more options.

```
$ ./manage.py runserver
```

Running a Production Web Server

Our recommended web server is uwsgi sitting behind nginx. The `bin/start_uwsgi.sh` script can been created to start `uwsgi` assuming your Ubuntu user is named `ubuntu`.

Also, static assets will need to be moved to S3 for production use. The `bin/post_compile` script contains a list of commands to move assets to S3.

```
$ bin/post_compile
```

```
$ bin/start_uwsgi
```

The following environment variables can be set within the `~/.bashrc` file to override default Django settings.

```
export SENTRY_DSN=https://xyz@app.getsentry.com/123
export DEBUG=False
export ONLY_HTTPS=True
```

General Linux Setup

While Amazon Web Services (AWS) provides the preferred hosting for SEED, running on a bare-bones Linux server follows a similar setup, replacing the AWS services with their Linux package counterparts, namely: PostgreSQL and Redis.

SEED is a Django project and Django's documentation is an excellent place to general understanding of this project's layout.

Prerequisites

Ubuntu server 14.04 or newer

Install the following base packages to run SEED:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install libpq-dev python-dev python-pip libatlas-base-dev \
gfortran build-essential g++ npm libxml2-dev libxslt1-dev git mercurial \
libssl-dev curl uwsgi-core uwsgi-plugin-python
sudo apt-get install redis-server
sudo apt-get install postgresql postgresql-contrib
```

Note: postgresql >=`9.3` is required to support JSON Type

Configure PostgreSQL

```
$ sudo su - postgres
$ createdb "seed-deploy"
$ createuser -P DBUsername
$ psql
postgres=# GRANT ALL PRIVILEGES ON DATABASE "seed-deploy" TO DBUsername;
postgres=# \q
$ exit
```

Note: Any database name and username can be used here in place of "seed-deploy" and DBUsername

Python Dependencies

clone the **seed** repository from **github**

```
$ git clone git@github.com:SEED-platform/seed.git
```

enter the repo and install the python dependencies from requirements

```
$ cd seed
$ sudo pip install -r requirements/local.txt
```

JavaScript Dependencies

`npm` is required to install the JS dependencies. The `bin/install_javascript_dependencies.sh` script will download all JavaScript dependencies and build them. `bower` and `gulp` should be installed globally for convenience.

```
$ curl -sL https://deb.nodesource.com/setup_5.x | sudo -E bash -
$ sudo apt-get install -y nodejs
$ sudo npm install -g bower gulp
```

```
$ bin/install_javascript_dependencies.sh
```

Django Database Configuration

Copy the `local_untracked.py.dist` file in the `config/settings` directory to `config/settings/local_untracked.py`, and add a `DATABASES` configuration with your database username, password, host, and port. Your database configuration can point to an AWS RDS instance or a PostgreSQL 9.4 database instance you have manually installed within your infrastructure.

```
# Database
DATABASES = {
    'default': {
        'ENGINE':'django.db.backends.postgresql_psycopg2',
        'NAME': 'seed-deploy',
        'USER': 'DBUsername',
        'PASSWORD': '',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Note: Other databases could be used such as MySQL, but are not supported due to the postgres-specific JSON Type

In in the above database configuration, `seed` is the database name, this is arbitrary and any valid name can be used as long as the database exists. Enter the database name, user, password you set above.

The database settings can be tested using the Django management command, `./manage.py dbshell` to connect to the configured database.

create the database tables and migrations:

```
$ python manage.py migrate
```

Cache and Message Broker

The SEED project relies on redis for both cache and message brokering, and is available as an AWS ElastiCache service or with the `redis-server` Linux package. (`sudo apt-get install redis-server`)

`local_untracked.py` should be updated with the `CACHES` and `BROKER_URL` settings.

```
CACHES = {
    'default': {
        'BACKEND': 'redis_cache.cache.RedisCache',
        'LOCATION': "127.0.0.1:6379",
        'OPTIONS': {'DB': 1},
        'TIMEOUT': 300
    }
}
BROKER_URL = 'redis://127.0.0.1:6379/1'
```

Creating the initial user

create a superuser to access the system

```
$ python manage.py create_default_user --username=demo@example.com --
↪organization=example --password=demo123
```

Note: Every user must be tied to an organization, visit `/app/#/profile/admin` as the superuser to create parent organizations and add users to them.

Running celery the background task worker

Celery is used for background tasks (saving data, matching, creating projects, etc) and must be connected to the message broker queue. From the project directory, `celery` can be started:

```
celery -A seed worker -l INFO -c 2 -B --events --maxtasksperchild 1000
```

Running the development web server

The Django dev server (not for production use) can be a quick and easy way to get an instance up and running. The dev server runs by default on port 8000 and can be run on any port. See Django's runserver documentation for more options.

```
$ python manage.py runserver --settings=config.settings.dev
```

Running a production web server

Our recommended web server is uwsgi sitting behind nginx. The python package `uwsgi` is needed for this, and should install to `/usr/local/bin/uwsgi` Since AWS S3, is not being used here, we recommend using `dj-static` to load static files.

Note: The use of the `dev` settings file is production ready, and should be used for non-AWS installs with `DEBUG` set to `False` for production use.

```
$ sudo pip install uwsgi dj-static
```

Generate static files:

```
$ sudo ./manage.py collectstatic --settings=config.settings.dev
```

Update `config/settings/local_untracked.py`:

```
DEBUG = False
# static files
STATIC_ROOT = 'collected_static'
STATIC_URL = '/static/'
```

Start the web server:

```
$ sudo /usr/local/bin/uwsgi --http :80 --module standalone_uwsgi --max-requests 5000 -
↪-pidfile /tmp/uwsgi.pid --single-interpreter --enable-threads --cheaper-initial 1 -
↪p 4
```

Warning: Note that uwsgi has port set to `80`. In a production setting, a dedicated web server such as NGINX would be receiving requests on port 80 and passing requests to uwsgi running on a different port, e.g 8000.

environmental variables

The following environment variables can be set within the `~/.bashrc` file to override default Django settings.

```
export SENTRY_DSN=https://xyz@app.getsentry.com/123
export DEBUG=False
export ONLY_HTTPS=True
```

SMTP service

In the AWS setup, we use SES to provide an email service Django can use as an email backend and configured it in our config/settings/main.py:

```
EMAIL_BACKEND = 'django_ses.SESBackend'
```

Many options for setting up your own SMTP service/server or using other SMTP third party services are available and compatible including gmail.

Django can likewise send emails via python's smtplib with sendmail or postfix installed. See their docs for more info.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

local_untracked.py

```
# PostgreSQL DB config
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'seed',
        'USER': 'your-username',
        'PASSWORD': 'your-password',
        'HOST': 'your-host',
        'PORT': 'your-port',
    }
}

# config for local storage backend
DEFAULT_FILE_STORAGE = 'django.core.files.storage.FileSystemStorage'
STATICFILES_STORAGE = DEFAULT_FILE_STORAGE
DOMAIN_URLCONFS = {}
DOMAIN_URLCONFS['default'] = 'urls.main'

CACHES = {
    'default': {
        'BACKEND': 'redis_cache.cache.RedisCache',
        'LOCATION': "127.0.0.1:6379",
        'OPTIONS': {'DB': 1},
        'TIMEOUT': 300
    }
}

# redis celery config
BROKER_URL = 'redis://127.0.0.1:6379/1'
CELERY_DEFAULT_QUEUE = 'seed-dev'
CELERY_QUEUES = (
    Queue(
        CELERY_DEFAULT_QUEUE,
        Exchange(CELERY_DEFAULT_QUEUE),
        routing_key=CELERY_DEFAULT_QUEUE
    ),
)

# SMTP config
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

# static files
STATIC_ROOT = 'collected_static'
STATIC_URL = '/static/'
```

Monitoring

Flower

Flower is used to monitor background tasks. Assuming your redis broker is running on *localhost* and on port *6379*, DB *1*. Then go to localhost:5555 to check celery. If running on AWS, the *bin/start_flower.sh* will start flower on port *8080* and be available for Google credentialed accounts.

```
flower --port=5555 --broker=redis://localhost:6379/1`
```

Sentry

Todo

Fill this out

CHAPTER 3

API

Authentication

Authentication is handled via an authorization token set in an http header. To request an API token, go to `/app/#/profile/developer` and click 'Get a New API Key'.

Every request must include an 'Authorization' http header made up of your username (email) and your API key, separated with a ':'. For example, with curl:

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e␣
↪https://seed-platform.org/api/v2/schema/
```

Or using the Python Requests library:

```
headers = {'authorization': 'user@email_address.com:
↪5edfd7f1f0696d4139118f8b95ab1f05d0dd418e'}
result = requests.get('https://seed-platform.org/api/v2/schema/', headers=headers)
print result.json()
```

If authentication fails, the response's status code will be 302, redirecting the user to `/app/login`.

Payloads

Many requests require a JSON-encoded payload and parameters in the query string of the url. A frequent requirement is including the organization_id of the org you belong to. For example:

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e␣
↪\
  https://seed-platform.org/api/v2/organizations/12/
```

Or in a JSON payload:

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e␣
↪\
  -d '{"organization_id":6, "role": "viewer"}' \
  https://seed-platform.org/api/v2/users/12/update_role/
```

Using Python:

```
headers = {'authorization': 'user@email_address.com:
↪5edfd7f1f0696d4139118f8b95ab1f05d0dd418e'}
params = json.dumps({'organization_id': 6, 'role': 'viewer'})
result = requests.post('https://seed-platform.org/api/v2/users/12/update_role/',
                       data=params,
                       headers=headers)
print result.json()
```

Responses

Responses from all requests will be JSON-encoded objects, as specified in each endpoint's documentation. In the case of an error, most endpoints will return this instead of the expected payload (or an HTTP status code):

```
.. code-block::
```

{ "status": "error", "message": "explanation of the error here"

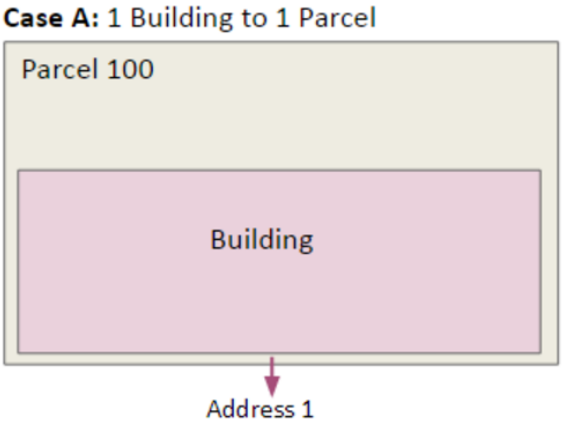
}

API-Endpoints

A list of interactive endpoints are available by accessing the API menu item on the left navigation pane within you account on your SEED instance.

To view a list of non-interactive endpoints without an account, view swagger on the development server.

Data Model

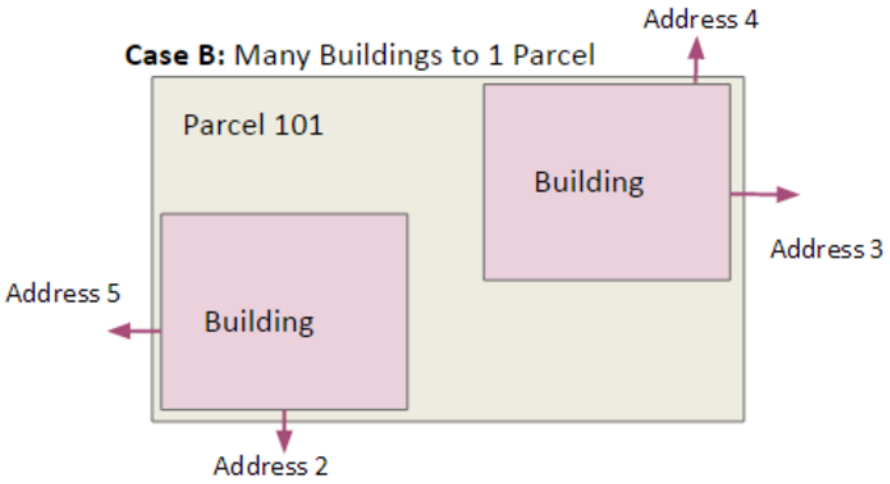


Source Data

Tax Assessor Data One Tax Lot ID per record		
Tax Lot ID	**Address**	**District**
100	44 West 1st	Willow

Building Data	
Building ID	**Tax Lot ID**
30	100

Portfolio Manager Data One PM record associated with one Tax Lot ID or Building ID					
PM ID	**Building ID**	**Tax Lot ID**	**Energy Score**	**EUI**	**Year Ending**
1	30	100	76	15,000	12/31/2015

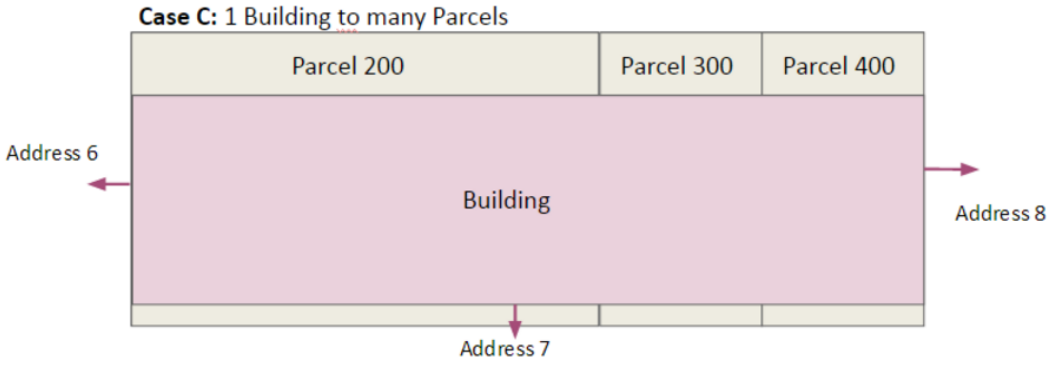


Source Data

Tax Assessor Data One Tax Lot ID per record		
Tax Lot ID	**Address**	**District**
101	15 Broadway	Willow

Building Data	
Building ID	**Tax Lot ID**
101-A	101
101-B	101

Portfolio Manager Data Multiple PM records associated with one Tax Lot ID or Building ID					
PM ID	**Building ID**	**Tax Lot ID**	**Energy Score**	**EUI**	**Year Ending**
2	101-A	101	66	12,000	12/31/2015
3	101-B	101	98	2,500	12/31/2015

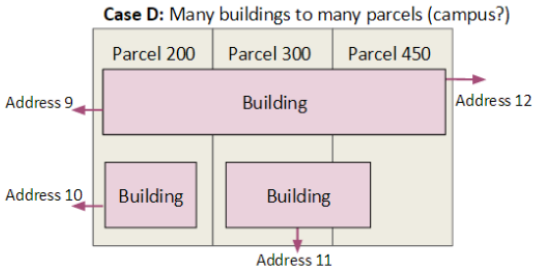


Source Data

Tax Assessor Data One Tax Lot ID per record		
Tax Lot ID	**Address**	**District**
200	1 Adams	Willow
300	2 West	Willow
400	3 Exeter	Willow

Building Data	
Building ID	**Tax Lot ID**
44	200;300;400

Portfolio Manager Data One PM record or Building ID associated with Multiple Tax Lot IDs					
PM ID	**Building ID**	**Tax Lot ID**	**Energy Score**	**EUI**	**Year Ending**
4	44	200;300;400	82	161,000	12/31/2015

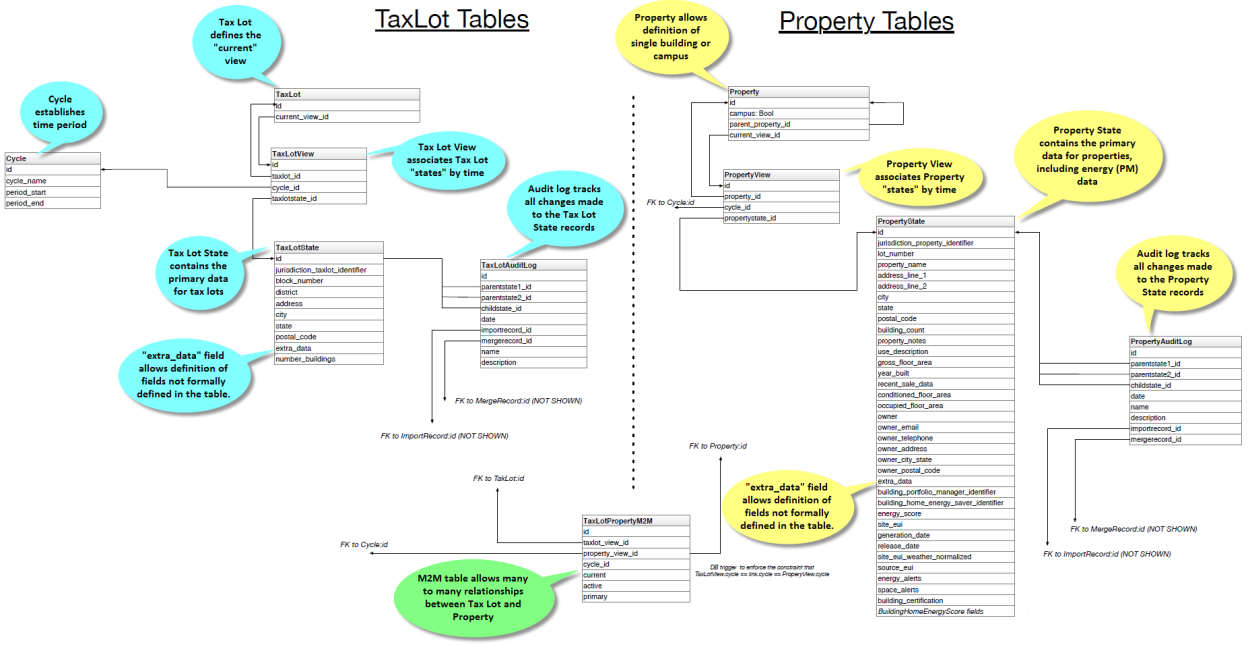


Source Data

Tax Assessor Data One Tax Lot ID per record		
Tax Lot ID	**Address**	**District**
200	1 Adams	Willow
300	2 West	Willow
400	3 Exeter	Willow

Building Data	
Building ID	**Tax Lot ID**
L1	200;300;400
L2	200
L3	300;450

Portfolio Manager Data Hierarchical campus to building: One PM record for campus and multiple PM records for campus buildings related many to many to Tax Lots							
PM ID	**Property Name**	**Parent Name**	**Parent PM ID**	**Tax Lot ID**	**Energy Score**	**EUI**	**Year Ending**
5	Lucky Campus	Lucky Campus	5	200;300;450	--	--	12/31/2013
6	Building 1	Lucky Campus	5	200;300;450	59	107	12/31/2013
7	Building 2	Lucky Campus	5	200	62	268	12/31/2013
8	Building 3	Lucky Campus	5	300;450	74	961	12/31/2013



Todo

Documentation below is out of state and needs updated.

Our primary data model is based on a tree structure with BuildingSnapshot instances as nodes of the tree and the tip of the tree referenced by a CanonicalBuilding.

Take the following example: a user has loaded a CSV file containing information about one building and created the first BuildingSnapshot (BS0). At this point in time, BS0 is linked to the first CanonicalBuilding (CB0), and CB0 is also linked to BS0.

```
BS0 <-- CB0
BS0 --> CB0
```

These relations are represented in the database as foreign keys from the BuildingSnapshot table to the CanonicalBuilding table, and from the CanonicalBuilding table to the BuildingSnapshot table.

The tree structure comes to fruition when a building, BS0 in our case, is matched with a new building, say BS1, enters the system and is auto-matched.

Here BS1 entered the system and was matched with BS0. When a match occurs, a new BuildingSnapshot is created, BS2, with the fields from the existing BuildingSnapshot, BS0, and the new BuildingSnapshot, BS1, merged together. If both the existing and new BuildingSnapshot have data for a given field, the new record's fields are preferred and merged into the child, B3.

The fields from new snapshot are preferred because that is the newer of the two records from the perspective of the system. By preferring the most recent fields this allows for evolving building snapshots over time. For example, if an existing canonical record has a Site EUI value of 75 and some changes happen to a building that cause this to change to 80 the user can submit a new record with that change.

All BuildingSnapshot instances point to a CanonicalBuilding.

```
BS0  BS1
  \ /
  BS2 <-- CB0

BS0 --> CB0
BS1 --> CB0
BS2 --> CB0
```

parents and children

BuildingSnapshots also have linkage to other BuildingSnapshots in order to keep track of their *parents* and *children*. This is represented in the Django model as a many-to-many relation from BuildingSnapshot to BuildingSnapshot. It is represented in the PostgreSQL database as an additional seed_buildingsnapshot_children table.

In our case here, BS0 and BS1 would both have *children* BS2, and BS2 would have *parents* BS0 and BS1.

Note: throughout most of the application, the `search_buildings` endpoint is used to search or list active building. This is to say, buildings that are pointed to by an active CanonicalBuilding. The `search_mapping_results` endpoint allows the search of buildings regardless of whether the BuildingSnapshot is pointed to by an active CanonicalBuilding or not and this search is needed during the mapping preview and matching sections of the application.

For illustration purposes let's suppose BS2 and a new building BS3 match to form a child BS4.

parent	child
BS0	BS2
BS1	BS2
BS2	BS4
BS3	BS4

And the corresponding tree would look like:

```
BS0  BS1
  \ /
  BS2  BS3
    \  /
     BS4 <-- CB0

BS0 --> CB0
BS1 --> CB0
BS2 --> CB0
BS3 --> CB0
BS4 --> CB0
```

matching

During the auto-matching process, if a *raw* BuildingSnapshot matches an existing BuildingSnapshot instance, then it will point to the existing BuildingSnapshot instance's CanonicalBuilding. In the case where there is no existing BuildingSnapshot to match, a new CanonicalBuilding will be created, as happened to B0 and C0 above.

field	BS0	BS1	BS2 (child)
id1	**11**	11	11
id2		**12**	12
id3	**13**		13
id4	14	**15**	15

manual-matching vs auto-matching

Since BuildingSnapshots can be manually matched, there is the possibility for two BuildingSnapshots each with an active CanonicalBuilding to match and the system has to choose to move only one CanonicalBuilding to the tip of the tree for the primary BuildingSnapshot and *deactivate* the secondary BuildingSnapshot's CanonicalBuilding.

Take for example:

```
BS0  BS1
  \ /
  BS2  BS3
    \  /
     BS4 <-- CB0 (active: True)          BS5 <-- CB1 (active: True)
```

If a user decides to manually match BS4 and BS5, the system will take the primary BuildingSnapshot's Canonical-Building and have it point to their child and deactivate CB1. The deactivation is handled by setting a field on the CanonicalBuilding instance, *active*, from `True` to `False`.

Here is what the tree would look like after the manual match of **BS4** and **BS5**:

```
BS0  BS1
  \ /
  BS2  BS3
```

```
    \  /
     BS4  BS5 <-- CB1 (active: False)
       \  /
        BS6 <-- CB0 (active: True)
```

Even though BS5 is pointed to by a CanonicalBuilding, CB1, BS5 will not be returned by the normal `search_buildings` endpoint because the CanonicalBuilding pointing to it has its field `active` set to `False`.

Note: anytime a match is **unmatched** the system will create a new CanonicalBuilding or set an existing CanonicalBuilding's active field to `True` for any leaf BuildingSnapshot trees.

---

what really happens to the BuildingSnapshot table on import (and when)

The above is conceptually what happens but sometimes the devil is in the details. Here is what happens to the BuildingSnapshot table in the database when records are imported.

Every time a record is added at least two BuildingSnapshot records are created.

Consider the following simple record:

Property Id	Year Ending	Property Floor Area	Address 1	Release Date
499045	2000	1234	1 fake st	12/12/2000

The first thing the user is upload the file. When the user sees the "Successful Upload!" dialog one record has been added to the BuildingSnapshot table.

This new record has an id (73700 in this case) and a created and modified timestamp. Then there are a lot of empty fields and a source_type of 0. Then there is the extra_data column which contains the contents of the record in key-value form:

Address 1 "1 fake st"

Property Id "499045"

Year Ending "2000"

Release Date "12/12/2000"

Property Floor Area "1234"

And a corresponding extra_data_sources that looks like

Address 1 73700

Property Id 73700

Year Ending 73700

Release Date 73700

Property Floor Area 73700

All of the fields that look like _source_id are also populated with 73700 E.G. owner_postal_code_source_id.

The other fields of interest are the organization field which is populated with the user's default organization and the import_file_id field which is populated with a reference to a data_importer_importfile record.

At this point the record has been created before the user hits the "Continue to data mapping" button.

The second record (id = 73701) is created by the time the user gets to the screen with the "Save Mappings" button. This second record has the following fields populated:

- id
- created
- modified
- pm_property_id
- year_ending
- gross_floor_area
- address_line_1
- release_date
- source_type (this is 2 instead of 0 as with the other record)
- import_file_id
- organization_id.

That is all. All other fields are empty. In this case that is all that happens.

Now consider the same user uploading a new file from the next year that looks like

Property Id	Year Ending	Property Floor Area	Address 1	Release Date
499045	2000	1234	1 fake st	12/12/2001

As before one new record is created on upload. This has id 73702 and follows the same pattern as 73700. And similarly 73703 is created like 73701 before the "Save Mappings" button appears.

However this time the system was able to make a match with an existing record. After the user clicks the "Confirm mappings & start matching" button a new record is created with ID 73704.

73704 is identical to 73703 (in terms of contents of the BuildingSnapshot table only) with the following exceptions:

- created and modified timestamps are different
- match type is populated and has a value of 1
- confidence is populated and has a value of .9
- source_type is 4 instead of 2
- canonical_building_id is populated with a value
- import_file_id is NULL
- last_modified_by_id is populated with value 2 (This is a key into the landing_seeduser table)
- address_line_1_source_id is 73701
- gross_floor_area_source_id is populated with value 73701
- pm_property_id_source_id is populated with 73701
- release_date_source_id is populated with 73701
- year_ending_source_id is populated with 73701

what really happens to the CanonicalBuilding table on import (and when)

In addition to the BuildingSnapshot table the CanonicalBuilding table is also updated during the import process. To summarize the above 5 records were created in the BuildingSnapshot table:

1. 73700 is created from the raw 2000 data
2. 73701 is the mapped 2000 data,
3. 73702 is created from the raw 2001 data
4. 73703 is the mapped 2001 data
5. 73704 is the result of merging the 2000 and 2001 data.

In this process CanonicalBuilding is updated twice. First when the 2000 record is imported the CanonicalBuilding gets populated with one new row at the end of the matching step. I.E. when the user sees the "Load More Data" screen. At this point there is a new row that looks like

id	active	canonical_building_id
20505	TRUE	73701

At this point there is one new canonical building and that is the BuildingSnapshot with id 73701. Next the user uploads the 2001 data. When the "Matching Results" screen appears the CanonicalBuilding table has been updated. Now it looks like

id	active	canonical_building_id
20505	TRUE	73704

There is still only one canonical building but now it is the BuildingSnapshot record that is the result of merging the 2000 and 2001 data: id = 73704.

organization

BuildingSnapshots belong to an Organization field that is a foreign key into the organization model (orgs_organization in Postgres).

Many endpoints filter the buildings based on the organizations the requesting user belongs to. E.G. get_buildings changes which fields are returned based on the requesting user's membership in the BuildingSnapshot's organization.

*_source_id fields

Any field in the BuildingSnapshot table that is populated with data from a submitted record will have a corresponding _source_id field. E.G pm_property_id has pm_property_id_source_id, address_line_1 has address_line_1_source_id, etc...

These are foreign keys into the BuildingSnapshot that is the source of that value. To extend the above table

field	BS0	BS1	BS2 (child)	BS2 (child) _source_id
id1	**11**		11	BS0
id2		**12**	12	BS1

NOTE: The BuildingSnapshot records made from the raw input file have all the _source_id fields populated with that record's ID. The non-canonical BuildingSnapshot records created from the mapped data have none set. The canonical BuildingSnapshot records that are the result of merging two records have only the _source_id fields set where the record itself has data. E.G. in the above address_line_1 is set to "1 fake st." so there is a value in the canonical

BuildingSnapshot's address_line_1_source_id field. However there is no block number so block_number_source_id is empty. This is unlike the two raw BuildingSnapshot records who also have no block_number but nevertheless have a block_number_source_id populated.

extra_data

The BuildingSnapshot model has many "named" fields. Fields like "address_line_1", "year_built", and "pm_property_id". However the users are allowed to submit files with arbitrary fields. Some of those arbitrary fields can be mapped to "named" fields. E.G. "Street Address" can usually be mapped to "Address Line 1". For all the fields that cannot be mapped like that there is the extra_data field.

extra_data is Django json field that serves as key-value storage for other user-submitted fields. As with the other "named" fields there is a corresponding extra_data_sources field that serves the same role as the other _source_id fields. E.G. If a BuildingSnapshot has an extra_data field that looks like

an_unknown_field 1

something_else 2

It should have an extra_data_sources field that looks like

an_unknown_field some_BuildingSnapshot_id

something_else another_BuildingSnapshot_id

saving and possible data loss

When saving a BuildingSnapshot file some fields that are truncated if too long. The following are truncated to 128 characters

- tax_lot_id
- pm_property_id
- custom_id_1
- lot_number
- block_number
- district
- owner
- owner_email
- owner_telephone
- owner_address
- owner_city_state
- owner_postal_code

And the following are truncated to 255:

- property_name
- address_line_1
- address_line_2

- city
- postal_code
- state_province
- building_certification

No truncation happens to any of the fields stored in extra_data.

CHAPTER 5

Mapping

This document describes the set of calls that occur from the web client or API down to the back-end for the process of mapping.

An overview of the process is:

1. Import - A file is uploaded and saved in the database
2. Mapping - Mapping occurs on that file
3. Matching / Merging
4. Pairing

Import

From the web UI, the import process invokes *seed.views.main.save_raw_data* to save the data. When the data is done uploading, we need to know whether it is a Portfolio Manager file, so we can add metadata to the record in the database. The end of the upload happens in *seed.data_importer.views.DataImportBackend.upload_complete* or *seed.data_importer.views.handle_s3_upload_complete*, depending on whether it is using a local file system or Amazon S3-based backend. At this point, the request object has additional attributes for Portfolio Manager files. These are saved in the model *seed.data_importer.models.ImportFile*.

Mapping

After the data is saved, the UI invokes *DataFileViewSet.mapping_suggestions* to get the columns to display on the mapping screen. This loads back the model that was mentioned above as an *ImportFile* instance, and then the *from_portfolio_manager* property can be used to choose the branch of the code:

If it is a Portfolio Manager file the *seed.common.mapper.get_pm_mapping* method provides a high-level interface to the Portfolio Manager mapping (see comments in the containing file, *mapper.py*), and the result is used to populate the return value for this method, which goes back to the UI to display the mapping screen.

Otherwise the code does some auto-magical logic to try and infer the "correct" mapping.

Matching

Todo

document

Pairing

Todo

document

CHAPTER 6

Modules

Audit Logs Package

Submodules

Models

class seed.audit_logs.models.**AuditLog**(**args*, ***kwargs*)

Bases: django_extensions.db.models.TimeStampedModel

An audit log of events and notes. Inherits created and modified from TimeStampedModel

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception AuditLog.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

AuditLog.**action_response**

A placeholder class that provides a way to set the attribute on the model.

AuditLog.**content_object**

Provide a generic many-to-one relation through the content_type and object_id fields.

This class also doubles as an accessor to the related object (similar to ForwardManyToOneDescriptor) by adding itself as a model attribute.

AuditLog.**content_type**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

AuditLog.**get_audit_type_display**(**moreargs*, ***morekwargs*)

AuditLog.**get_next_by_created**(**moreargs*, ***morekwargs*)

AuditLog.**get_next_by_modified**(**moreargs*, ***morekwargs*)

AuditLog.**get_previous_by_created**(**moreargs*, ***morekwargs*)

AuditLog.**get_previous_by_modified**(**moreargs*, ***morekwargs*)

AuditLog.**objects = <seed.audit_logs.models.AuditLogManager object>**

AuditLog.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

AuditLog.**save**(**args*, ***kwargs*)
Ensure that only notes are saved

AuditLog.**to_dict**()
serializes an audit_log

AuditLog.**user**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

class seed.audit_logs.models.**AuditLogManager**
Bases: django.db.models.manager.Manager

ExpressionManager with update preventing the update of non-notes

get_queryset()

log_action(*request*, *conent_object*, *organization_id*, *action_response=None*, *action_note=None*, *audit_type=0*)

use_for_related_fields = True

class seed.audit_logs.models.**AuditLogQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)
Bases: django.db.models.query.QuerySet

update(**args*, ***kwargs*)
only notes should be updated, so filter out non-notes

Tests

class seed.audit_logs.tests.**AuditLogModelTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_audit()
tests audit save

test_audit_save()
audit_log LOG should not be able to save/update

test_audit_update()
audit_log LOG should not be able to save/update

test_generic_relation()
test CanonicalBuilding.audit_logs

test_get_all_audit_logs_for_an_org()
gets all audit logs for an org

test_model___unicode__()
tests the AuditLog inst. str or unicode

test_note()
tests note save

test_note_save()
notes should be able to save/update

class seed.audit_logs.tests.**AuditLogViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_create_note()
tests create_note

test_get_building_logs()
test the django view get_building_logs

test_update_note()
tests update_note

URLs

Views

seed.audit_logs.views.**create_note**(*request*, **args*, ***kwargs*)
Retrieves logs for a building.

POST Expects the CanonicalBuildings's id in the JSON payload as building_id. Expects an organization_id (to which project belongs) in the query string. Expects the action_note to be in the JSON payload as action_note

Returns:

```
'audit_log' : {
    'user': {
        'first_name': user's firstname,
        'last_name': user's last_name,
        'id': user's id,
        'email': user's email address
    },
    'id': audit log's id,
    'audit_type': 'Note',
    'created': DateTime,
    'modified': DateTime,
    'action': method triggering log entry,
    'action_response': response of action,
    'action_note': the note body
    'organization': {
        'name': name of org,
        'id': id of org
```

```
},
'status': 'success'
```

seed.audit_logs.views.**get_building_logs**(*request*, **args*, ***kwargs*)

Retrieves logs for a building.

GET Expects the CanonicalBuildings's id in the query string as building_id. Expects an organization_id (to which project belongs) in the query string.

Returns:

```
'audit_logs' : [
    {
        'user': {
            'first_name': user's firstname,
            'last_name': user's last_name,
            'id': user's id,
            'email': user's email address
        },
        'id': audit log's id,
        'audit_type': 'Log' or 'Note',
        'created': DateTime,
        'modified': DateTime,
        'action': method triggering log entry,
        'action_response': response of action,
        'action_note': the note body if Note or further description
        'organization': {
            'name': name of org,
            'id': id of org
        }
    }, ...
],
'status': 'success'
```

seed.audit_logs.views.**update_note**(*request*, **args*, ***kwargs*)

Retrieves logs for a building.

PUT Expects the CanonicalBuildings's id in the JSON payload as building_id. Expects an organization_id (to which project belongs) in the query string. Expects the action_note to be in the JSON payload as action_note Expects the audit_log_id to be in the JSON payload as audit_log_id

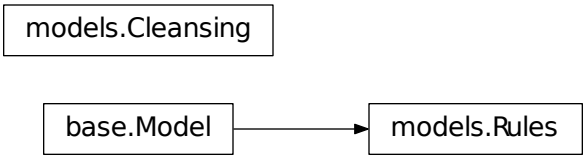
Returns:

```
'audit_log' : {
    'user': {
        'first_name': user's firstname,
        'last_name': user's last_name,
        'id': user's id,
        'email': user's email address
    },
    'id': audit log's id,
    'audit_type': 'Note',
    'created': DateTime,
    'modified': DateTime,
    'action': method triggering log entry,
    'action_response': response of action,
    'action_note': the note body
    'organization': {
        'name': name of org,
```

```
        'id': id of org
},
'status': 'success'
```

Cleansing Package

Inheritance



Submodules

Models

class seed.cleansing.models.**Cleansing**(*organization*, **args*, ***kwargs*)

Bases: object

static cache_key(*file_pk*)

Static method to return the location of the cleansing results from redis.

Parameters **file_pk** – Import file primary key

Returns

cleanse(*record_type*, *data*)

Send in data as a queryset from the BuildingSnapshot ids. :param record_type: one of property/taxlot :param data: rows of data to be cleansed :return:

data_type_check(*datum*)

Check the data types of the fields. These should never be wrong as these are the data in the database.

This chunk of code is currently ignored.

Parameters **datum** – Database record containing the BS version of the fields populated

Returns None

get_fieldnames(*record_type*)

Get fieldnames to apply to results.

in_range_checking(*datum*)

Check for errors in the min/max of the values.

Parameters **datum** – Database record containing the BS version of the fields populated

Returns None

static initialize_cache(*file_pk*)
Initialize the cache for storing the results. This is called before the celery tasks are chunked up.

Parameters **file_pk** – Import file primary key

Returns

missing_matching_field(*datum*)
Look for fields in the database that are not matched. Missing is defined as a None in the database

Parameters **datum** – Database record containing the BS version of the fields populated

Returns None

TODO: NL: Should we check the extra_data field for the data?

missing_values(*datum*)
Look for fields in the database that are empty. Need to know the list of fields that are part of the cleansing section.

The original intent of this method would be very intensive to run (looking at all fields except the ignored). This method was changed to check for required values.

Parameters **datum** – Database record containing the BS version of the fields populated

Returns None

TODO: Check the extra_data field for the data?

prune_data()
Prune the results will remove any entries that have zero cleansing_results

Returns None

reset_results()

save_to_cache(*file_pk*)
Save the results to the cache database. The data in the cache are stored as a list of dictionaries. The data in this class are stored as a dict of dict. This is important to remember because the data from the cache cannot be simply loaded into the above structure.

Parameters **file_pk** – Import file primary key

Returns None

class seed.cleansing.models.**Rules**(*id*, *org*, *field*, *enabled*, *category*, *type*, *min*, *max*, *severity*, *units*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Rules.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

static Rules.**delete_rules**(*organization*)

Rules.**get_category_display**(**moreargs*, ***morekwargs*)

Rules.**get_severity_display**(**moreargs*, ***morekwargs*)

Rules.**get_type_display**(**moreargs*, ***morekwargs*)

static Rules.**initialize_rules**(*organization*)

Rules.**objects = <django.db.models.manager.Manager object>**

Rules.**org**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

static Rules.**restore_defaults**(*organization*)

Tasks

Tests

class seed.cleansing.tests.**CleansingDataSample**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_cleanse()

class seed.cleansing.tests.**CleansingDataTestCoveredBuilding**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_cleanse(**args*, ***kwargs*)

test_simple_login()

class seed.cleansing.tests.**CleansingDataTestPM**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_cleanse()

class seed.cleansing.tests.**CleansingViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_get_cleansing_results()

test_get_csv()

test_get_progress()

URLs

Views

Common Package

Submodules

Mapper

Util

seed.common.util.**apply_map**(*map_path*, *data_path*, *out_file*)
Apply a JSON mapping to data, and write the output.

Args: map_path (str): Path to mapping file data_path (str): Path to data file out_file (file): output stream

Return: None

seed.common.util.**create_map**(*path_in*, *path_out*)
Create a JSON mapping file, suitable for *map.Mapping()*, from a CSV input file in our own custom style.

Input columns: CurrentSEED,NewSEED,PM1,PM2,Type (ignore rest)

Parameters

- **path_in** –
- **path_out** –

Returns None

seed.common.util.**find_duplicates**(*map_path*, *data_path*, *out_file*)
Find duplicates created by a given mapping on a given input file.

Args: map_path (str): Path to mapping file data_path (str): Path to data file out_file (file): output stream

Return: None

Views

seed.common.views.**api_error**(*reason*)
JSON response for API error.

seed.common.views.**api_success**(***kwargs*)
JSON response for API success.

seed.common.views.**missing_request_keys**(*keys*, *body*)
Check for *keys* in *body*.

Args: keys (list): List of keys to check body (dict): body of request

Returns: None if all present, JSON error response (using *api_error()*) if one or more is missing.

seed.common.views.**typeof_request_values**(*types*, *body*)
Check for type of request values.

All keys in types are present in body.

Parameters

- **types** (*dict*) – key to type map, where *type* is a function used to test the type conversion. It should take one argument, the value, and raise a ValueError if it is invalid.
- **body** (*dict*) – body of request

Returns JSON

Example:

```
None if all OK, JSON error response (using `api_error()`)
if one or more is of the wrong type.
```

Configuration

Submodules

Storage

class config.storage.**CachedS3BotoStorage**(**args*, ***kwargs*)

Bases: storages.backends.s3boto.S3BotoStorage

S3 storage backend that saves the files locally, too.

save(*name*, *content*)

Template Context

config.template_context.**sentry_js**(*request*)

config.template_context.**session_key**(*request*)

Tests

Utils

config.utils.**de_camel_case**(*name*)

Views

config.views.**robots_txt**(*request*, *allow=False*)

WSGI

WSGI config for config project.

This module contains the WSGI application used by Django's development server and any production WSGI deployments. It should expose a module-level variable named application. Django's runserver and runfcgi commands discover this application via the WSGI_APPLICATION setting.

Usually you will have the standard Django WSGI application here, but it also might make sense to replace the whole Django WSGI application with a custom one that later delegates to the Django one. For example, you could introduce WSGI middleware here, or combine a Django application with an application of another framework.

Data Package

Submodules

BEDES

Module contents

Data Importer Package

Submodules

Managers

class seed.data_importer.managers.**NotDeletedManager**
Bases: django.db.models.manager.Manager

get_queryset(**args*, ***kwargs*)

use_for_related_fields = True

Models

URLs

Utils

class seed.data_importer.utils.**CoercionRobot**
Bases: object

lookup_hash(*uncoerced_value*, *destination_model*, *destination_field*)

make_key(*value*, *model*, *field*)

seed.data_importer.utils.**acquire_lock**(*name*, *expiration=None*)
Tries to acquire a lock from the cache. Also sets the lock's value to the current time, allowing us to see how long it has been held.

Returns False if lock already belongs by another process.

seed.data_importer.utils.**chunk_iterable**(*iter*, *chunk_size*)
Breaks an iterable (e.g. list) into smaller chunks, returning a generator of the chunk.

seed.data_importer.utils.**get_core_pk_column**(*table_column_mappings*, *primary_field*)

seed.data_importer.utils.**get_lock_time**(*name*)
Examines a lock to see when it was acquired.

seed.data_importer.utils.**release_lock**(*name*)
Frees a lock.

Views

class seed.data_importer.views.**ImportFileViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ViewSet

authentication_classes = (<class 'rest_framework.authentication.SessionAuthentication'>, <class 'seed.authentica

cleansing_progress(*request*, **args*, ***kwargs*)
Return the progress of the cleansing. — type:

status: required: true type: string description: either success or error

progress: type: integer description: status of background cleansing task

parameter_strategy: replace parameters:

•name: pk description: Import file ID required: true paramType: path

filtered_mapping_results(*request*, **args*, ***kwargs*)
Retrieves a paginated list of Properties and Tax Lots for an import file after mapping. — parameter_strategy: replace parameters:

•name: pk description: Import File ID (Primary key) type: integer required: true paramType: path

response_serializer: MappingResultsResponseSerializer

first_five_rows(*request*, **args*, ***kwargs*)
Retrieves the first five rows of an ImportFile. — type:

status: required: true type: string description: either success or error

first_five_rows: type: array of strings description: list of strings for each of the first five rows for this import file

parameter_strategy: replace parameters:

•name: pk description: "Primary Key" required: true paramType: path

get_cleansing_results(*request*, **args*, ***kwargs*)
Retrieve the details of the cleansing script. — type:

status: required: true type: string description: either success or error

message: type: string description: additional information, if any

progress: type: integer description: integer percent of completion

data: type: JSON description: object describing the results of the cleansing

parameter_strategy: replace parameters:

•name: pk description: Import file ID required: true paramType: path

get_csv(*request*, **args*, ***kwargs*)
Download a csv of the results. — type:

status: required: true type: string description: either success or error

progress_key: type: integer description: ID of background job, for retrieving job progress

parameter_strategy: replace parameters:

•name: pk description: Import file ID required: true paramType: path

mapping_done(*request*, **args*, ***kwargs*)
Tell the backend that the mapping is complete. — type:

status: required: true type: string description: either success or error

message: required: false type: string description: error message, if any

parameter_strategy: replace parameters:

•name: pk description: Import file ID required: true paramType: path

matching_results (*request*, **args*, ***kwargs*)

Retrieves the number of matched and unmatched BuildingSnapshots for a given ImportFile record.

GET Expects import_file_id corresponding to the ImportFile in question.

Returns:

```
{
    'status': 'success',
    'matched': Number of BuildingSnapshot objects that have matches,
    'unmatched': Number of BuildingSnapshot objects with no matches.
}
```

perform_mapping (*request*, **args*, ***kwargs*)

Starts a background task to convert imported raw data into PropertyState and TaxLotState, using user's column mappings. — type:

status: required: true type: string description: either success or error

progress_key: type: integer description: ID of background job, for retrieving job progress

parameter_strategy: replace parameters:

•name: pk description: Import file ID required: true paramType: path

raise_exception = True

raw_column_names (*request*, **args*, ***kwargs*)

Retrieves a list of all column names from an ImportFile. — type:

status: required: true type: string description: either success or error

raw_columns: type: array of strings description: list of strings of the header row of the ImportFile

parameter_strategy: replace parameters:

•name: pk description: "Primary Key" required: true paramType: path

retrieve (*request*, **args*, ***kwargs*)

Retrieves details about an ImportFile. — type:

status: required: true type: string description: either success or error

import_file: type: ImportFile structure description: full detail of import file

parameter_strategy: replace parameters:

•name: pk description: "Primary Key" required: true paramType: path

save_column_mappings (*request*, **args*, ***kwargs*)

Saves the mappings between the raw headers of an ImportFile and the destination fields in the *to_table_name* model which should be either PropertyState or TaxLotState

Valid source_type values are found in `seed.models.SEED_DATA_SOURCES`

Payload:

```
{
    "import_file_id": ID of the ImportFile record,
    "mappings": [
        {
            'from_field': 'eui',  # raw field in import file
            'to_field': 'energy_use_intensity',
            'to_table_name': 'PropertyState',
        },
        {
            'from_field': 'gfa',
            'to_field': 'gross_floor_area',
            'to_table_name': 'PropertyState',
        }
    ]
}
```

Returns:

```
{'status': 'success'}
```

save_raw_data(*request*, **args*, ***kwargs*)

Starts a background task to import raw data from an ImportFile into PropertyState objects as extra_data. If the cycle_id is set to year_ending then the cycle ID will be set to the year_ending column for each record in the uploaded file. Note that the year_ending flag is not yet enabled. — type:

status: required: true type: string description: either success or error

message: required: false type: string description: error message, if any

progress_key: type: integer description: ID of background job, for retrieving job progress

parameter_strategy: replace parameters:

- name: pk description: Import file ID required: true paramType: path
- name: cycle_id description: The ID of the cycle or the string "year_ending" paramType: string required: true

start_system_matching(*request*, **args*, ***kwargs*)

Starts a background task to attempt automatic matching between buildings in an ImportFile with other existing buildings within the same org. — type:

status: required: true type: string description: either success or error

progress_key: type: integer description: ID of background job, for retrieving job progress

parameter_strategy: replace parameters:

- name: pk description: Import file ID required: true paramType: path

suffix = None

class seed.data_importer.views.**LocalUploaderViewSet**(***kwargs*)

Bases: rest_framework.viewsets.GenericViewSet

Endpoint to upload data files to, if uploading to local file storage. Valid source_type values are found in seed.models.SEED_DATA_SOURCES

Returns:

```
{
    'success': True,
```

```
    'import_file_id': The ID of the newly-uploaded ImportFile
}
```

create(*request*, **args*, ***kwargs*)

Upload a new file to an import_record. This is a multipart/form upload. — parameters:

- name: import_record description: the ID of the ImportRecord to associate this file with. required: true paramType: body
- name: source_type description: the type of file (e.g. 'Portfolio Raw' or 'Assessed Raw') required: false paramType: body
- name: source_program_version description: the version of the file as related to the source_type required: false paramType: body
- name: file or qqfile description: In-memory file object required: true paramType: Multipart

suffix = None

class seed.data_importer.views.**MappingResultsPayloadSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

filter_params = <django.contrib.postgres.fields.jsonb.JSONField>

class seed.data_importer.views.**MappingResultsPropertySerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

class seed.data_importer.views.**MappingResultsResponseSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

class seed.data_importer.views.**MappingResultsTaxLotSerializer**(*instance=None*, *data=<class rest_framework.fields.empty>*, ***kwargs*)

Bases: rest_framework.serializers.Serializer

Module contents

Features Package

Submodules

Module contents

Green Button Package

Subpackages

Green Button Tests Package

Submodules

XML Importer Tests

class seed.green_button.tests.test_xml_importer.**GreenButtonXMLImportTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests of various ways of authenticating to the API.

Uses the get_building endpoint in all cases.

assert_models_created()
Tests that appropriate models for the sample xml file have been created.

setUp()

tearDown()

test_create_models()
Test of xml_importer.create_models.

test_import_xml()
Test of xml_importer.import_xml.

class seed.green_button.tests.test_xml_importer.**GreenButtonXMLParsingTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests helper functions for pulling green button building data out of xml snippets.

assert_fn_mapping(*fn*, *mapping*)
Takes a function fn and a mapping from input values to expected output values. Asserts that fn returns the expected output for each of the input values.

setUp()

tearDown()

test_as_collection()
Test of xml_importer.as_collection.

test_building_data()
Test of xml_importer.building_data.

test_energy_type()
Test of xml_importer.energy_type.

test_energy_units()
Test of function that converts a green button 'uom' (unit of measurement?) integer to one of seed.models.ENERGY_UNITS.

test_interval_block_data()
Test of xml_importer.interval_block_data.

test_interval_data()
Test of xml_importer.interval_data.

test_meter_data()
Test of xml_importer.meter_data.

Module contents

Submodules

seed.green_button.xml_importer module

seed.green_button.xml_importer.**as_collection**(*val*)
Takes a value, returns that value if it is not a string and is an Iterable, and returns a list containing that value if it is not an Iterable or if it is a string. Returns None when val is None.

Parameters **val** – any value

Returns list containing val or val if it is Iterable and not a string.

seed.green_button.xml_importer.**building_data**(*xml_data*)
Extracts information about a building from a Green Button XML file.

Parameters **xml_data** – dictionary returned by xmltodict.parse when called on the contents of a Green Button XML file

Returns dictionary

•building information for a Green Button XML file

•information describing the meter used for collection

•list of time series meter reading data

seed.green_button.xml_importer.**create_models**(*data*, *import_file*)
Create a BuildingSnapshot, a CanonicalBuilding, and a Meter. Then, create TimeSeries models for each meter reading in data.

Parameters

- **data** – dictionary of building data from a Green Button XML file in the form returned by xml_importer.building_data
- **import_file** – ImportFile referencing the original xml file; needed for linking to BuildingSnapshot and for determining super_organization

Returns the created CanonicalBuilding

seed.green_button.xml_importer.**energy_type**(*service_category*)
Returns the seed model energy type corresponding to the green button service category.

Parameters **service_category** – int that is a green button service_category (string args will be converted to integers)

Returns int in seed.models.ENERGY_TYPES

seed.green_button.xml_importer.**energy_units**(*uom*)
Returns the seed model energy unit corresponding to the green button uom.

Parameters **uom** – int that is the green button uom number corresponding to the energy units supported by the green button schema (string args will be converted to integers)

Returns int in seed.models.ENERGY_UNITS

seed.green_button.xml_importer.**import_xml**(*import_file*)
Given an import_file referencing a raw Green Button XML file, extracts building and time series information from the file and constructs required database models.

Parameters **import_file** – a seed.models.ImportFile instance representing a Green Button XML file that has been previously uploaded

Returns the created CanonicalBuilding Inst.

seed.green_button.xml_importer.**interval_block_data**(*ib_xml_data*)
Takes a dictionary containing the contents of an IntervalBlock node from a Green Button XML file and returns a dictionary containing the start_time of the time series collection, the duration of the collection, and a list of readings containing the time series data from a meter.

Parameters **ib_xml_data** – dictionary of the contents of an IntervalBlock from a Green Button XML file

Returns dictionary containing meta data about an entire collection period and a list of the specific meter readings

seed.green_button.xml_importer.**interval_data**(*reading_xml_data*)
Takes a dictionary representing the contents of an IntervalReading XML node and pulls out data for a single time series reading. The dictionary will be a sub-dictionary of the dictionary returned by xmltodict.parse when called on a Green Button XML file. Returns a flat dictionary containing the interval data.

Parameters **reading_xml_data** – dictionary of IntervalReading XML node content in format specified by the xmltodict library.

Returns dictionary representing a time series reading with keys 'cost', 'value', 'start_time', and 'duration'.

seed.green_button.xml_importer.**meter_data**(*raw_meter_meta*)
Takes a dictionary representing the contents of the entry node in a Green Button XML file that specifies the meta data about the meter that was used to record time series data for that file. Returns a flat dictionary containing the meter meta data.

Parameters **raw_meter_meta** – dictionary of the contents of the meter specification entry node in a Green Button XML file

Returns dictionary containing information about a meter with keys 'currency', 'power_of_ten_multiplier', and 'uom'

Module contents

Landing Package

Subpackages

seed.landing.management package

Subpackages

Landing Management Package

Submodules

Update EULA

class seed.landing.management.commands.update_eula.**Command**(*stdout=None*, *stderr=None*, *no_color=False*)

Bases: django.core.management.base.BaseCommand

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

help = 'Update the Terms of Service with a new HTML file'

Module contents

Module contents

Submodules

Forms

class seed.landing.forms.**LoginForm**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *field_order=None*)

Bases: django.forms.forms.Form

base_fields = OrderedDict([('email', <django.forms.fields.EmailField object>), ('password', <django.forms.fields.Cha

declared_fields = OrderedDict([('email', <django.forms.fields.EmailField object>), ('password', <django.forms.field

media

Models

class seed.landing.models.**SEEDUser**(**args*, ***kwargs*)

Bases: django.contrib.auth.base_user.AbstractBaseUser, django.contrib.auth.models.PermissionsMixin

An abstract base class implementing a fully featured User model with admin-compliant permissions.

Username, password and email are required. Other fields are optional.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception SEEDUser.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

SEEDUser.**REQUIRED_FIELDS = ['email']**

SEEDUser.**USERNAME_FIELD = 'username'**

SEEDUser.**auditlog_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

SEEDUser.**buildingsnapshot_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

SEEDUser.**columnmapping_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

SEEDUser.**cycle_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

SEEDUser.**default_building_detail_custom_columns**
A placeholder class that provides a way to set the attribute on the model.

SEEDUser.**default_custom_columns**
A placeholder class that provides a way to set the attribute on the model.

SEEDUser.**default_organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

SEEDUser.**email_user**(*subject*, *message*, *from_email=None*)
Sends an email to this User.

SEEDUser.**generate_key**()
Creates and sets an API key for this user. Adapted from tastypie:

https://github.com/toastdriven/django-tastypie/blob/master/tastypie/models.py#L47 # noqa

SEEDUser.**get_absolute_url**()

SEEDUser.**get_full_name**()
Returns the first_name plus the last_name, with a space in between.

SEEDUser.**get_next_by_date_joined**(**moreargs*, ***morekwargs*)

SEEDUser.**get_previous_by_date_joined**(**moreargs*, ***morekwargs*)

SEEDUser.**get_short_name**()
Returns the short name for the user.

SEEDUser.**groups**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**importrecord_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**last_modified_user**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**logentry_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**modified_import_records**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**objects = <django.contrib.auth.models.UserManager object>**

SEEDUser.**organizationuser_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

SEEDUser.**orgs**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`project_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`projectbuilding_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`projectpropertyview_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`projecttaxlotview_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`save`**(**args*, ***kwargs*)
Ensure that email and username are synced.

`SEEDUser.`**`user_agreement`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`SEEDUser.`**`user_permissions`**

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Tests

class `seed.landing.tests.`**`UserLoginTest`**(*methodName='runTest'*)

Bases: `django.test.testcases.TestCase`

`setUp`()

`test_simple_login`()

Happy path login with no ToS.

`test_tos_login`()

Happy path login when there is a ToS.

URLs

Views

`seed.landing.views.`**`landing_page`**(*request*)

`seed.landing.views.`**`login_view`**(*request*)

Standard Django login, with additions: Lowercase the login email (username) Check user has accepted ToS, if any.

`seed.landing.views.`**`password_reset`**(*request*)

`seed.landing.views.`**`password_reset_complete`**(*request*)

`seed.landing.views.`**`password_reset_confirm`**(*request*, *uidb64=None*, *token=None*)

`seed.landing.views.`**`password_reset_done`**(*request*)

`seed.landing.views.`**`password_set`**(*request*, *uidb64=None*, *token=None*)

`seed.landing.views.`**`signup`**(*request*, *uidb64=None*, *token=None*)

Module contents

Library Packages

Submodules

Exporter Module

class seed.lib.exporter.**Exporter**(*export_id*, *export_name*, *export_type*)
Class to handle the exporting of buildings

export(*buildings*, *fields*, *row_cb*)
The main method of export. Uses the export type defined by the initializer

Parameters

- **buildings** – Array of building ids to export
- **fields** – Array of fields to export
- **row_cb** – ID for row cache

Returns

export_csv(*qs*, *fields=[]*, *cb=None*)

export_xls(*qs*, *fields=[]*, *cb=None*)

static fields_from_queryset(*qs*)
Creates a list of all accessible fields on a model based off of a queryset.

This method should not be here. It seems that is should be on the building snapshot model. Not moved yet because I am unsure if the qs argument is more than one data type (i.e. BuildingSnapshot and/or ?)

filename()
The expected file name based on the export_id, export_name, and export_type

Returns String of the expected filename

subdirectory()
Create and return the subdirectory

Returns String of the subdirectory

static subdirectory_from_export_id(*export_id*)
Return the subdirectory as constructed by the instance method.

Parameters **export_id** – The export ID

Returns String of the path to the exported file

tempfile = None

valid_export_type()

seed.lib.exporter.**batch_qs**(*qs*, *batch_size=1000*)
From: https://djangosnippets.org/snippets/1170/

Returns a (start, end, total, queryset) tuple for each batch in the given queryset.

Usage:

seed.lib.exporter.**construct_obj_row**(*obj*, *fields*)
Creates an exportable row of data from an object and a list of fields. Ignores nones and instances of the Django Manager object, replacing them with blank unicode strings.

seed.lib.exporter.**get_field_name_from_model**(*field*, *model*)
Takes a field name like "building_snapshot__state" and returns the verbose field name as set in django, to be used as the header in exported files.

Parameters

- **field** –
- **qs** –

Returns

seed.lib.exporter.**get_field_value_from_instance**(*field*, *obj*)
Does some deep diving to find the right value given a string like "building_snapshot__state"

seed.lib.exporter.**qs_to_rows**(*qs*, *fields*)

Module contents

Mapping Package

Submodules

seed.mappings.mapper module

seed.mappings.seed_mappings module

Module contents

Managers Package

Subpackages

Manager Tests Package

Submodules

Test JSON Manager

class seed.managers.tests.test_json_manager.**TestJsonManager**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_order_by_returns_all_buildings()
Test that we're able to order by values of a json field.

Module contents

Submodules

JSON

class seed.managers.json.**JsonManager**
Bases: django.db.models.manager.Manager

get_queryset()

class seed.managers.json.**JsonQuerySet**(*model=None*, *query=None*, *using=None*, *hints=None*)
Bases: django.db.models.query.QuerySet

PRIMARY = 'extra_data'

TABLE = 'seed_buildingsnapshot'

json_order_by(*key*, *order_by*, *order_by_rev=False*, *unit=None*)

Module contents

Models

Submodules

AuditLog

Columns

class seed.models.columns.**Column**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

The name of a column for a given organization.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Column.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Column.**SOURCE_CHOICES = (('P', 'Property'), ('T', 'Taxlot'))**

Column.**SOURCE_CHOICES_MAP = {'P': 'property', 'T': 'taxlot'}**

Column.**SOURCE_PROPERTY = 'P'**

Column.**SOURCE_TAXLOT = 'T'**

static Column.**create_mappings**(*mappings*, *organization*, *user*)
Create the mappings for an organization and a user based on a simple array of array object.

Args: mappings: dictionary containing mapping information

mappings: [

{ 'from_field': 'eui', 'to_field': 'energy_use_intensity', 'to_table_name': 'property',

}, {

'from_field': 'eui', 'to_field': 'energy_use_intensity', 'to_table_name': 'property',

}

]

organization: Organization object user: User object

Returns: True (data are saved in the ColumnMapping table in the database)

Column.**enum**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Column.**get_extra_data_source_display**(**moreargs*, ***morekwargs*)

Column.**mapped_mappings**

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Column.**objects = <django.db.models.manager.Manager object>**

Column.**organization**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Column.**raw_mappings**

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

static Column.**save_column_names**(*model_obj*)

Save unique column names for extra_data in this organization.

This is a record of all the extra_data keys we've ever seen for a particular organization.

Parameters **model_obj** – model_obj instance (either PropertyState or TaxLotState).

Column.**to_dict**()
Convert the column object to a dictionary

Returns dict

Column.**unit**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

class seed.models.columns.**ColumnMapping**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Stores previous user-defined column mapping.

We'll pull from this when pulling from varied, dynamic source data to present the user with previous choices for that same field in subsequent data loads.

exception **DoesNotExist**
Bases: django.core.exceptions.ObjectDoesNotExist

exception ColumnMapping.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

ColumnMapping.**column_mapped**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

ColumnMapping.**column_raw**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

static ColumnMapping.**delete_mappings**(*organization*)
Delete all the mappings for an organization. Note that this will erase all the mappings so if a user views an existing Data Mapping the mappings will not show up as the actual mapping, rather, it will show up as new suggested mappings

Parameters **organization** – instance, Organization

Returns int, Number of records that were deleted

static ColumnMapping.**get_column_mappings**(*organization*)
Returns dict of all the column mappings for an Organization's given source type

Parameters **organization** – instance, Organization.

Returns dict, list of dict.

Use this when actually performing mapping between data sources, but only call it after all of the mappings have been saved to the ColumnMapping table.

static ColumnMapping.**get_column_mappings_by_table_name**(*organization*)
Breaks up the get_column_mappings into another layer to provide access by the table name as a key.

Parameters **organization** – instance, Organization

Returns dict

ColumnMapping.**get_source_type_display**(**moreargs*, ***morekwargs*)

ColumnMapping.**is_concatenated**()
Returns True if the ColumnMapping represents the concatenation of imported column names; else returns False.

ColumnMapping.**is_direct**()
Returns True if the ColumnMapping is a direct mapping from imported column name to either a BEDES column or a previously imported column. Returns False if the ColumnMapping represents a concatenation.

ColumnMapping.**objects = <django.db.models.manager.Manager object>**

ColumnMapping.**remove_duplicates**(*qs*, *m2m_type='column_raw'*)
Remove any other Column Mappings that use these columns.

Parameters

- **qs** – queryset of Column. These are the Columns in a M2M with this instance.
- **m2m_type** – str, the name of the field we're comparing against. Defaults to 'column_raw'.

ColumnMapping.**save**(**args*, ***kwargs*)
Overrides default model save to eliminate duplicate mappings.

> **Warning:** Other column mappings which have the same raw_columns in them will be removed!

ColumnMapping.**super_organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

ColumnMapping.**to_dict**()
Convert the ColumnMapping object to a dictionary

Returns dict

ColumnMapping.**user**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

seed.models.columns.**get_column_mapping**(*raw_column*, *organization*, *attr_name='column_mapped'*)
Find the ColumnMapping objects that exist in the database from a raw_column

Parameters

- **raw_column** – str, the column name of the raw data.
- **organization** – Organization inst.
- **attr_name** – str, name of attribute on ColumnMapping to pull out. whether we're looking at a mapping from the perspective of a raw_column (like we do when creating a mapping), or mapped_column, (like when we're applying that mapping). # TODO: Remove the use of this attr_name

Returns list of mapped items, float representation of confidence.

Cycles

class seed.models.cycles.**Cycle**(*id*, *organization*, *user*, *name*, *start*, *end*, *created*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Cycle.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Cycle.**get_next_by_created**(**moreargs*, ***morekwargs*)

Cycle.**get_next_by_end**(**moreargs*, ***morekwargs*)

Cycle.**get_next_by_start**(**moreargs*, ***morekwargs*)

Cycle.**get_previous_by_created**(**moreargs*, ***morekwargs*)

Cycle.**get_previous_by_end**(**moreargs*, ***morekwargs*)

Cycle.**get_previous_by_start**(**moreargs*, ***morekwargs*)

Cycle.**importfile_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Cycle.**objects = <django.db.models.manager.Manager object>**

Cycle.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Cycle.**propertyview_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Cycle.**taxlotproperty_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Cycle.**taxlotview_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Cycle.**user**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Joins

class seed.models.joins.**TaxLotProperty**(*id*, *property_view*, *taxlot_view*, *cycle*, *primary*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception TaxLotProperty.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

TaxLotProperty.**cycle**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

TaxLotProperty.**objects = <django.db.models.manager.Manager object>**

TaxLotProperty.**property_view**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

TaxLotProperty.**taxlot_view**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

Generic Models

class seed.models.models.**AttributeOption**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Holds a single conflicting value for a BuildingSnapshot attribute.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception AttributeOption.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

AttributeOption.**building_variant**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

AttributeOption.**get_value_source_display**(**moreargs*, ***morekwargs*)

AttributeOption.**objects = <django.db.models.manager.Manager object>**

class seed.models.models.**BuildingAttributeVariant**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

Place to keep the options of BuildingSnapshot attribute variants.

When we want to select which source's values should sit in the Canonical Building's position, we need to draw from a set of options determined during the matching phase. We should only have one 'Variant' container per field_name, per snapshot.

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception BuildingAttributeVariant.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

BuildingAttributeVariant.**building_snapshot**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

BuildingAttributeVariant.**objects = <django.db.models.manager.Manager object>**

BuildingAttributeVariant.**options**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

class seed.models.models.**Compliance**(*id*, *created*, *modified*, *compliance_type*, *start_date*, *end_date*, *deadline_date*, *project*)

Bases: django_extensions.db.models.TimeStampedModel

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception Compliance.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

Compliance.**get_compliance_type_display**(**moreargs*, ***morekwargs*)

Compliance.**get_next_by_created**(**moreargs*, ***morekwargs*)

Compliance.**get_next_by_modified**(**moreargs*, ***morekwargs*)

Compliance.**get_previous_by_created**(*moreargs, **morekwargs)

Compliance.**get_previous_by_modified**(*moreargs, **morekwargs)

Compliance.**objects = <django.db.models.manager.Manager object>**

Compliance.**project**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Compliance.**to_dict**()

class seed.models.models.**CustomBuildingHeaders**(*args, **kwargs)
Bases: django.db.models.base.Model

Specify custom building header mapping for display.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception CustomBuildingHeaders.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

CustomBuildingHeaders.**building_headers**
A placeholder class that provides a way to set the attribute on the model.

CustomBuildingHeaders.**objects = <seed.managers.json.JsonManager object>**

CustomBuildingHeaders.**super_organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

class seed.models.models.**Enum**(*args, **kwargs)
Bases: django.db.models.base.Model

Defines a set of enumerated types for a column.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Enum.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Enum.**column_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Enum.**enum_values**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Enum.**objects = <django.db.models.manager.Manager object>**

class seed.models.models.**EnumValue**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Individual Enumerated Type values.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception EnumValue.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

EnumValue.**objects = <django.db.models.manager.Manager object>**

EnumValue.**values**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

class seed.models.models.**Meter**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Meter specific attributes.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Meter.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Meter.**building_snapshot**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Meter.**get_energy_type_display**(**moreargs*, ***morekwargs*)

Meter.**get_energy_units_display**(**moreargs*, ***morekwargs*)

Meter.**objects = <django.db.models.manager.Manager object>**

Meter.**timeseries_data**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

class seed.models.models.**NonCanonicalProjectBuildings**(**args*, ***kwargs*)

Bases: django.db.models.base.Model

Holds a reference to all project buildings that do not point at a canonical building snapshot.

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception NonCanonicalProjectBuildings.**MultipleObjectsReturned**

Bases: django.core.exceptions.MultipleObjectsReturned

NonCanonicalProjectBuildings.**objects = <django.db.models.manager.Manager object>**

NonCanonicalProjectBuildings.**projectbuilding**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

class seed.models.models.**StatusLabel**(*id*, *created*, *modified*, *name*, *color*, *super_organization*)

Bases: django_extensions.db.models.TimeStampedModel

BLUE_CHOICE = 'blue'

COLOR_CHOICES = (('red', <django.utils.functional.__proxy__ object>), ('blue', <django.utils.functional.__proxy__ obje

DEFAULT_LABELS = ['Residential', 'Non-Residential', 'Violation', 'Compliant', 'Missing Data', 'Questionable Report', '

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

StatusLabel.**GRAY_CHOICE = 'gray'**

StatusLabel.**GREEN_CHOICE = 'green'**

StatusLabel.**LIGHT_BLUE_CHOICE = 'light blue'**

exception StatusLabel.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

StatusLabel.**ORANGE_CHOICE = 'orange'**

StatusLabel.**RED_CHOICE = 'red'**

StatusLabel.**WHITE_CHOICE = 'white'**

StatusLabel.**canonicalbuilding_set**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

StatusLabel.**get_color_display**(**moreargs*, ***morekwargs*)

StatusLabel.**get_next_by_created**(**moreargs*, ***morekwargs*)

StatusLabel.**get_next_by_modified**(**moreargs*, ***morekwargs*)

StatusLabel.**get_previous_by_created**(**moreargs*, ***morekwargs*)

StatusLabel.**get_previous_by_modified**(**moreargs*, ***morekwargs*)

StatusLabel.**objects = <django.db.models.manager.Manager object>**

StatusLabel.**property_set**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

StatusLabel.**super_organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

StatusLabel.**taxlot_set**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

StatusLabel.**to_dict**()

class seed.models.models.**TimeSeries**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

For storing energy use over time.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception TimeSeries.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

TimeSeries.**meter**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

TimeSeries.**objects = <django.db.models.manager.Manager object>**

class seed.models.models.**Unit**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Unit of measure for a Column Value.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Unit.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Unit.**column_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Unit.**get_unit_type_display**(**moreargs*, ***morekwargs*)

Unit.**objects = <django.db.models.manager.Manager object>**

seed.models.models.**get_ancestors**(*building*)
gets all the non-raw, non-composite ancestors of a building

Recursive function to traverse the tree upward.

Parameters **building** – BuildingSnapshot inst.

Returns list of BuildingSnapshot inst., ancestors of building

```
source_type {
    2: ASSESSED_BS,
    3: PORTFOLIO_BS,
    4: COMPOSITE_BS,
    6: GREEN_BUTTON_BS
}
```

seed.models.models.**get_sourced_attributes**(*snapshot*)
Return all the attribute names that get sourced.

seed.models.models.**set_initial_sources**(*snapshot*)
Sets the PK for the original sources to self.

Projects

class seed.models.projects.**Project**(*id*, *created*, *modified*, *name*, *slug*, *owner*, *last_modified_by*, *super_organization*, *description*, *status*)
Bases: django_extensions.db.models.TimeStampedModel

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Project.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Project.**adding_buildings_status_percentage_cache_key**

Project.**building_snapshots**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Project.**compliance_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Project.**get_compliance**()

Project.**get_next_by_created**(**moreargs*, ***morekwargs*)

Project.**get_next_by_modified**(**moreargs*, ***morekwargs*)

Project.**get_previous_by_created**(**moreargs*, ***morekwargs*)

Project.**get_previous_by_modified**(**moreargs*, ***morekwargs*)

Project.**get_status_display**(**moreargs*, ***morekwargs*)

Project.**has_compliance**

Project.**last_modified_by**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Project.**objects = <django.db.models.manager.Manager object>**

Project.**organization**
For compliance with organization names in new data model.

Project.**owner**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

Project.**project_building_snapshots**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Project.**project_property_views**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

Project.**project_taxlot_views**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

```
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`Project.`**`property_count`**

`Project.`**`property_views`**

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`Project.`**`removing_buildings_status_percentage_cache_key`**

`Project.`**`super_organization`**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`Project.`**`taxlot_count`**

`Project.`**`taxlot_views`**

Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`Project.`**`to_dict`**`()`

class `seed.models.projects.`**`ProjectBuilding`**(*id*, *created*, *modified*, *building_snapshot*, *project*, *compliant*, *approved_date*, *approver*)

Bases: `django_extensions.db.models.TimeStampedModel`

exception **`DoesNotExist`**

Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `ProjectBuilding.`**`MultipleObjectsReturned`**

Bases: `django.core.exceptions.MultipleObjectsReturned`

`ProjectBuilding.`**`approver`**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectBuilding.`**`building_snapshot`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectBuilding.`**`get_next_by_created`**(**moreargs*, ***morekwargs*)

`ProjectBuilding.`**`get_next_by_modified`**(**moreargs*, ***morekwargs*)

`ProjectBuilding.`**`get_previous_by_created`**(**moreargs*, ***morekwargs*)

`ProjectBuilding.`**`get_previous_by_modified`**(**moreargs*, ***morekwargs*)

`ProjectBuilding.`**`noncanonicalprojectbuildings_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`ProjectBuilding.`**`objects = <django.db.models.manager.Manager object>`**

`ProjectBuilding.`**`project`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectBuilding.`**`to_dict`**()

class `seed.models.projects.`**`ProjectPropertyView`**(*id*, *created*, *modified*, *property_view*, *project*, *compliant*, *approved_date*, *approver*)
Bases: `django_extensions.db.models.TimeStampedModel`

exception **`DoesNotExist`**
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `ProjectPropertyView.`**`MultipleObjectsReturned`**
Bases: `django.core.exceptions.MultipleObjectsReturned`

`ProjectPropertyView.`**`approver`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectPropertyView.`**`get_next_by_created`**(**moreargs*, ***morekwargs*)

`ProjectPropertyView.`**`get_next_by_modified`**(**moreargs*, ***morekwargs*)

`ProjectPropertyView.`**`get_previous_by_created`**(**moreargs*, ***morekwargs*)

`ProjectPropertyView.`**`get_previous_by_modified`**(**moreargs*, ***morekwargs*)

`ProjectPropertyView.`**`objects = <django.db.models.manager.Manager object>`**

`ProjectPropertyView.`**`project`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectPropertyView.`**`property_view`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

class `seed.models.projects.`**`ProjectTaxLotView`**(*id*, *created*, *modified*, *taxlot_view*, *project*, *compliant*, *approved_date*, *approver*)

Bases: `django_extensions.db.models.TimeStampedModel`

exception **`DoesNotExist`**
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `ProjectTaxLotView.`**`MultipleObjectsReturned`**
Bases: `django.core.exceptions.MultipleObjectsReturned`

`ProjectTaxLotView.`**`approver`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`ProjectTaxLotView.`**`get_next_by_created`**(**moreargs*, ***morekwargs*)

`ProjectTaxLotView.`**`get_next_by_modified`**(**moreargs*, ***morekwargs*)

`ProjectTaxLotView.`**`get_previous_by_created`**(**moreargs*, ***morekwargs*)

`ProjectTaxLotView.`**`get_previous_by_modified`**(**moreargs*, ***morekwargs*)

ProjectTaxLotView.**objects = <django.db.models.manager.Manager object>**

ProjectTaxLotView.**project**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

ProjectTaxLotView.**taxlot_view**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

Properties

class seed.models.properties.**Property**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

The canonical property

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception Property.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

Property.**labels**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Property.**objects = <django.db.models.manager.Manager object>**

Property.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

Property.**parent_property**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

Property.**property_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

Property.**views**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

class seed.models.properties.**PropertyAuditLog**(*id*, *organization*, *parent1*, *parent2*, *state*, *view*, *name*, *description*, *import_filename*, *record_type*, *created*)

Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception PropertyAuditLog.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

PropertyAuditLog.**get_record_type_display**(**moreargs*, ***morekwargs*)

PropertyAuditLog.**objects = <django.db.models.manager.Manager object>**

PropertyAuditLog.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

PropertyAuditLog.**parent1**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyAuditLog.`**`parent2`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyAuditLog.`**`propertyauditlog__parent1`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyAuditLog.`**`propertyauditlog__parent2`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyAuditLog.`**`state`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyAuditLog.`**`view`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

class seed.models.properties.**PropertyState**(**args*, ***kwargs*)
Bases: django.db.models.base.Model

Store a single property

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception PropertyState.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

PropertyState.**clean**(**args*, ***kwargs*)

PropertyState.**extra_data**
A placeholder class that provides a way to set the attribute on the model.

PropertyState.**get_data_state_display**(**moreargs*, ***morekwargs*)

PropertyState.**get_merge_state_display**(**moreargs*, ***morekwargs*)

PropertyState.**import_file**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

PropertyState.**objects = <django.db.models.manager.Manager object>**

PropertyState.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

PropertyState.**promote**(*cycle*)
Promote the PropertyState to the view table for the given cycle

Args: cycle: Cycle to assign the view

Returns: The resulting PropertyView (note that it is not returning the PropertyState)

PropertyState.**propertyauditlog__state**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

PropertyState.**propertyview_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyState.`**`save`**(**args*, ***kwargs*)

`PropertyState.`**`to_dict`**(*fields=None*, *include_related_data=True*)
Returns a dict version of the PropertyState, either with all fields or masked to just those requested.

class `seed.models.properties.`**`PropertyView`**(**args*, ***kwargs*)
Bases: `django.db.models.base.Model`

Similar to the old world of canonical building.

exception `DoesNotExist`
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `PropertyView.`**`MultipleObjectsReturned`**
Bases: `django.core.exceptions.MultipleObjectsReturned`

`PropertyView.`**`cycle`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyView.`**`import_filename`**
Get the import file name form the audit logs

`PropertyView.`**`initialize_audit_logs`**(***kwargs*)

`PropertyView.`**`objects = <django.db.models.manager.Manager object>`**

`PropertyView.`**`project_property_views`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyView.`**`project_set`**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyView.`**`property`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyView.`**`propertyauditlog__view`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyView.`**`save`**(**args*, ***kwargs*)

`PropertyView.`**`state`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`PropertyView.`**`tax_lot_states`**()
Return a list of TaxLotStates associated with this PropertyView and Cycle

Returns list of TaxLotStates

`PropertyView.`**`tax_lot_views`**()
Return a list of TaxLotViews that are associated with this PropertyView and Cycle

Returns list of TaxLotViews

`PropertyView.`**`taxlotproperty_set`**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`PropertyView.`**`update_state`**(*new_state*, ***kwargs*)

TaxLots

class seed.models.tax_lots.**TaxLot**(*id*, *organization*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception TaxLot.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

TaxLot.**labels**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

pizza.toppings and topping.pizzas are ManyToManyDescriptor instances.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

TaxLot.**objects = <django.db.models.manager.Manager object>**

TaxLot.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

TaxLot.**views**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

class seed.models.tax_lots.**TaxLotAuditLog**(*id*, *organization*, *parent1*, *parent2*, *state*, *view*, *name*, *description*, *import_filename*, *record_type*, *created*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception TaxLotAuditLog.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

TaxLotAuditLog.**get_record_type_display**(**moreargs*, ***morekwargs*)

TaxLotAuditLog.**objects = <django.db.models.manager.Manager object>**

TaxLotAuditLog.**organization**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

TaxLotAuditLog.**parent1**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

TaxLotAuditLog.**parent2**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

TaxLotAuditLog.**state**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

TaxLotAuditLog.**taxlotauditlog__parent1**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

TaxLotAuditLog.**taxlotauditlog__parent2**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`TaxLotAuditLog.`**`view`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

class `seed.models.tax_lots.`**`TaxLotState`**(*id*, *confidence*, *import_file*, *organization*, *data_state*, *merge_state*, *custom_id_1*, *jurisdiction_tax_lot_id*, *block_number*, *district*, *address_line_1*, *normalized_address*, *address_line_2*, *city*, *state*, *postal_code*, *number_properties*, *extra_data*)

Bases: `django.db.models.base.Model`

exception **`DoesNotExist`**
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception `TaxLotState.`**`MultipleObjectsReturned`**
Bases: `django.core.exceptions.MultipleObjectsReturned`

`TaxLotState.`**`extra_data`**
A placeholder class that provides a way to set the attribute on the model.

`TaxLotState.`**`get_data_state_display`**(**moreargs*, ***morekwargs*)

`TaxLotState.`**`get_merge_state_display`**(**moreargs*, ***morekwargs*)

`TaxLotState.`**`import_file`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`TaxLotState.`**`objects = <django.db.models.manager.Manager object>`**

`TaxLotState.`**`organization`**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`TaxLotState.`**`promote`**(*cycle*)
Promote the TaxLotState to the view table for the given cycle

Args: cycle: Cycle to assign the view

Returns: The resulting TaxLotView (note that it is not returning the TaxLotState)

TaxLotState.**save**(**args*, ***kwargs*)

TaxLotState.**taxlotauditlog__state**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

TaxLotState.**taxlotview_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

TaxLotState.**to_dict**(*fields=None*, *include_related_data=True*)
Returns a dict version of the TaxLotState, either with all fields or masked to just those requested.

class seed.models.tax_lots.**TaxLotView**(*id*, *taxlot*, *state*, *cycle*)
Bases: `django.db.models.base.Model`

exception DoesNotExist
Bases: `django.core.exceptions.ObjectDoesNotExist`

exception TaxLotView.**MultipleObjectsReturned**
Bases: `django.core.exceptions.MultipleObjectsReturned`

TaxLotView.**cycle**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

TaxLotView.**import_filename**
Get the import file name form the audit logs

TaxLotView.**initialize_audit_logs**(***kwargs*)

TaxLotView.**objects = <django.db.models.manager.Manager object>**

TaxLotView.**project_set**
Accessor to the related objects manager on the forward and reverse sides of a many-to-many relation.

In the example:

```
class Pizza(Model):
    toppings = ManyToManyField(Topping, related_name='pizzas')
```

`pizza.toppings` and `topping.pizzas` are `ManyToManyDescriptor` instances.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`TaxLotView.`**`project_taxlot_views`**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`parent.children` is a `ReverseManyToOneDescriptor` instance.

Most of the implementation is delegated to a dynamically defined manager class built by `create_forward_many_to_many_manager()` defined below.

`TaxLotView.`**`property_states`**`()`

Return a list of PropertyStates associated with this TaxLotView and Cycle

Returns list of PropertyStates

`TaxLotView.`**`property_views`**`()`

Return a list of PropertyViews that are associated with this TaxLotView and Cycle

Returns list of PropertyViews

`TaxLotView.`**`save`**`(`*args*, ***kwargs*`)`

`TaxLotView.`**`state`**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`TaxLotView.`**`taxlot`**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

`child.parent` is a `ForwardManyToOneDescriptor` instance.

`TaxLotView.`**`taxlotauditlog__view`**

Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

TaxLotView.**taxlotproperty_set**
Accessor to the related objects manager on the reverse side of a many-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

parent.children is a ReverseManyToOneDescriptor instance.

Most of the implementation is delegated to a dynamically defined manager class built by create_forward_many_to_many_manager() defined below.

TaxLotView.**update_state**(*new_state*, ***kwargs*)

Module contents

Public Package

Submodules

Models

Because migrations are complicated, we're keeping our public fields here.

This deals with circular dependency issues between LANDINGUser and Organization

class seed.public.models.**SharedBuildingField**(**args*, ***kwargs*)
Bases: django_extensions.db.models.TimeStampedModel

BuildingSnapshot Exported Field, either public or internally shared.

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception SharedBuildingField.**MultipleObjectsReturned**
Bases: django.core.exceptions.MultipleObjectsReturned

SharedBuildingField.**field**
Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

SharedBuildingField.**get_field_type_display**(**moreargs*, ***morekwargs*)

SharedBuildingField.**get_next_by_created**(**moreargs*, ***morekwargs*)

SharedBuildingField.**get_next_by_modified**(**moreargs*, ***morekwargs*)

SharedBuildingField.**get_previous_by_created**(**moreargs*, ***morekwargs*)

SharedBuildingField.**get_previous_by_modified**(*moreargs, **morekwargs)

SharedBuildingField.**objects = <django.db.models.manager.Manager object>**

SharedBuildingField.**org**

Accessor to the related object on the forward side of a many-to-one or one-to-one relation.

In the example:

```
class Child(Model):
    parent = ForeignKey(Parent, related_name='children')
```

child.parent is a ForwardManyToOneDescriptor instance.

Module contents

SEED Package

Subpackages

Management Package

Subpackages

Management Packages

Submodules

S3

class seed.management.commands.set_s3_expires_headers_for_angularjs_partials.**Command**(*stdout=Nor*
stderr=Nor
no_color=F

Bases: django.core.management.base.BaseCommand

handle(*args, **options)

help = "Sets S3 Expires headers for AngularJS partials to prevent browser caching old html partials. ./manage.py set_s3

option_list = (<Option at 0x7f053b8ed440: –prefix>,)

Module contents

Module contents

Templatetags Package

Submodules

Breadcrumbs

class seed.templatetags.breadcrumbs.**BreadcrumbNode**(*vars*, *render_func=<function create_crumb>*)

Bases: django.template.base.Node

render(*context*)

class seed.templatetags.breadcrumbs.**UrlBreadcrumbNode**(*title*, *url_node*, *render_func=<function create_crumb>*)

Bases: django.template.base.Node

render(*context*)

seed.templatetags.breadcrumbs.**breadcrumb**(*parser*, *token*)

Section author: Andriy Drozdyuk

Renders the breadcrumb.

Example:

```
{% breadcrumb "Title of breadcrumb" url_var %}
{% breadcrumb context_var  url_var %}
{% breadcrumb "Just the title" %}
{% breadcrumb just_context_var %}
```

Parameters:

```
First parameter is the title of the crumb
Second (optional) parameter is the url variable to link to, produced by url tag,␣
↪i.e.:
    {% url "person_detail" object.id as person_url %}
    then:
    {% breadcrumb person.name person_url %}
```

seed.templatetags.breadcrumbs.**breadcrumb_root**(*parser*, *token*)

Section author: Andriy Drozdyuk

Renders the breadcrumb.

Examples:

```
{% breadcrumb "Title of breadcrumb" url_var %}
{% breadcrumb context_var  url_var %}
{% breadcrumb "Just the title" %}
{% breadcrumb just_context_var %}
```

Parameters:

```
First parameter is the title of the crumb,
Second (optional) parameter is the url variable to link to, produced by url tag,␣
↪i.e.:
    {% url "person_detail/" object.id as person_url %}
    then:
    {% breadcrumb person.name person_url %}
```

seed.templatetags.breadcrumbs.**breadcrumb_url**(*parser*, *token*)

Same as breadcrumb but instead of url context variable takes in all the arguments URL tag takes.

```
{% breadcrumb "Title of breadcrumb" person_detail person.id %}
{% breadcrumb person.name person_detail person.id %}
```

seed.templatetags.breadcrumbs.**breadcrumb_url_root**(*parser*, *token*)
Same as breadcrumb but instead of url context variable takes in all the arguments URL tag takes.

```
{% breadcrumb "Title of breadcrumb" person_detail person.id %}
{% breadcrumb person.name person_detail person.id %}
```

seed.templatetags.breadcrumbs.**create_crumb**(*title*, *url=None*)
Helper function

seed.templatetags.breadcrumbs.**create_crumb_first**(*title*, *url=None*)
Helper function

Test Helpers Package

Subpackages

Test Helper Factor Package

Subpackages

Test Helper Factory Lib Package

Submodules

Chomsky

seed.test_helpers.factory.lib.chomsky.**generate_chomsky**(*times=5*, *line_length=72*)

Submodules

Helpers

class seed.test_helpers.factory.helpers.**DjangoFunctionalFactory**

classmethod invalid_test_cc_number()

classmethod rand_bool()

classmethod rand_city()

classmethod rand_city_suffix()

classmethod rand_currency(*start=0*, *end=100*)

classmethod rand_date(*start_year=1900*, *end_year=2011*)

classmethod rand_domain()

classmethod rand_email()

classmethod rand_float(*start=0*, *end=100*)

classmethod **rand_int**(*start=0*, *end=100*)

classmethod **rand_name**()

classmethod **rand_phone**()

classmethod **rand_plant_name**()

classmethod **rand_str**(*length=None*)

classmethod **rand_street_address**()

classmethod **rand_street_suffix**()

classmethod **random_conversation**(*paragraphs=3*)

classmethod **test_cc_number**(*valid=True*)

classmethod **valid_test_cc_number**()

Module contents

Tests Package

Submodules

Tests (Functional) Package

Submodules

Test Browsers

Functional Testing with Selenium

The tests in seed.functional.tests.test_browser use Selenium to test things actually work in a browser. There are a number of things you need to know before adding tests.

..warning:

```
Ignore this at your peril!
```

In order to test against multiple browsers without having to duplicate Test Classes the Test Cases, the classes they subclass from, and all the tests themselves are all generated on the fly.

The tests are written in such a way that, hopefully, adding tests, will be familiar, and little different from normal. If all you want to do is add a test, providing you bear in mind the caveats below, you should not need to concern your self with how things actually work behind the scenes. Follow the instructions and you can write one test method and it will automatically run against multiple browsers.

Adding a test

test_browser.py contains two basic test classes:

LoggedOutTests this contains test methods that run when a user is logged out

LoggedInTests this contains test methods that run when a user is logged in

These should be sufficient for most cases and you can just add your test as normal.

> **Warning:** These are not defined in the top level scope

Rather they are defined **within** the equivalent generator. e.g. loggedout_tests_generator. This is the key to the behind the scenes magic, the generator invokes the class multiple times, each time sub-classing it from a different base class. These are identical but for the fact that they use a different browser to power the tests.

As long as you respect this, and indent your methods properly, you can write your tests as normal. You should also read the section, Running Tests, however, as the tests are not run in the normal way, so the error message generated by a failing test may surprise you.

Adding a browser specific test.

... WARNING:: Test detection, internally, is as smart as say Nose, tests must begin with `test_` to be recognized. Define your test as normal but put anything browser specific behind a guard condition. To do this you can test against the browser name:

```
if self.browser_type.name == 'Firefox':
```

The Browser Definition that defines this name is contained in browser_definition.py.

You can also add tests that will only be invoked by Travis using:

```
if os.getenv('TRAVIS') == 'true':
```

When tests are run locally(i.e. not on Travis) only cross platform browsers will be used. If you need to write a browser specific test against anything else, and you wish to run the test locally, you can override this by setting the environment 'SEED_TEST_BROWSER' to one of IE, SAFARI, EDGE. This will cause that browser to be used as well. You will need to ensure you have the relevant driver installed.

Running Tests.

Tests can be run as normal, but a failing test may produce different output from what you are expecting. In particular the FAIL: statement will list the name of the generator, not the test method. So you will see

However the place the test failed can still be derived from the trace back e.g.:

```
Traceback (most recent call last):
    [snip]...
    File "/home/paulmunday/projects/seed/seed/functional/tests/test_browser.py",
↪line 341, in test_pass
```

In addition the tests will print the name of a failed test and the browser used to standard error, though this prints in line with the tests, not the same place as the report from the test runner.

Adding Test Cases to test_browsers.py

The included classes should be sufficient but it is easy enough to add more. Just make sure to wrap you class definition in a generator. You can base this off an existing generator like so:

```
def my_test_generator():
    for browser in BROWSERS:
        class MyTestClass(LOGGED_IN_CLASSES[browser.name]):
            def my_test(self):
                pass
        Test = LoggedOutTests()
          for test in get_tsts(Test):
              yield test
```

Lets break that down a little:

` for browser in BROWSERS:` In combination with the next line this will create a copy of your test for each browser. BROWSERS is imported from base.py and is a list of Browser Definitions

`class MyTestClass(LOGGED_IN_CLASSES[browser.name]):` Rather than sub-classing directly this supplies a different base case each time we go around the loop. In this case the base classes are taken from LOGGED_IN_CLASSES. This is imported from base.py: its a list of Base Classes generated there that each use a different browser to power the same base Test Class. (LOGGED_OUT_CLASSES is also defined and it also possible to define other base classes there). This gives us the ability to test against multiple browsers easily. *You must* inherit from a class defined there.

` Test = LoggedOutTests()` Here we instantiate a copy of out derived test class. Since we invoke it directly, not via the test runner, the setUp methods won't be called, however get_tsts takes care of this.

`for test in get_tsts(Test):` get_tsts takes care of running the setUpClass method for us, then it examines the test class we passed looking for methods it thinks are tests (i.e. their name begins with `test_`). When it finds one it wraps it in a function that the test runner can call (making sure to invoke the setUp and tearDown methods) and will be safe for the generator to yield. (you can't return a Test Class from the generator, as unittest expects a function).

... Caution :: see setUp and tearDown below if you intend to override these.

setUp and tearDown

If you want to override these you need to be aware there are not invoked by the test runner. Instead the function yielded by the generator invokes them manually . You must take care to reverse anything you did in the setup (especially anything that calls a Model) or stuff will break in unpredictable ways. You should be doing this regardless anyway. Also be sure to call super as this takes care of opening and closing the browser.

How to Write Tests

Within the Selenium world there is a design pattern known as Page Objects. A Page object "represents an area in the web application user interface that your test is interacting", with the idea that this produces reusable code that only needs to be fixed in one place.

Essentially each page has its own associated Page Object class. These are used to load the page and query it. The SEED Selenium testing framework makes extensive use of these. Initiating a page object verifies the page object and has the ability to create any database records needed for the page to function.

Each page object has the same find_element(s)_by... Functionality as Web Elements (and the Browser/Webdriver) so you any page element you find using these also has the same methods. It also provides easy access to ActionChains methods for more complex browser interactions. Since these are all just wrappers around the Selenium Webdriver and Action Chain methods you can consult the existing Python-Selenium documentation for more information. http://selenium-python.readthedocs.io.

Page objects also have wait_for_element methods. These correspond to the equivalent find_element methods but contain Explicit Waits to allow time for the element to appear. These can be used just after a page is loaded, or for AJAX loaded elements to make sure they are present before querying them. These work in the same way as the corresponding find_element method. Note they can only locate a single element, the first found, in the same way as the find_element methods. There is no equivalent of the find_elements methods. To emulate these use a wait_for_element method to return the parent container then query this using find_elements.

In addition, since SEED has lots of tabular data, it extends the Page Object design pattern with the notion a Table object. Querying tables can be tricky with Selenium. Typically you need to identify a particular cell by XPath and this can be a laborious process. To compound this table structure is not always fixed in SEED so what columns and what order they will be displayed in will vary per user/organization.

Table objects aim to get around these limitations by providing an easily query-able representation of a table. This will be explained in more detail below, but essentially a Table Object stores each row in a Table Row object. A Table Row is a wrapper round an Ordered Dict that uses the table header as keys so you can access a particular cell by its table header without worrying about its position. In addition it can also be accessed by index (i.e. column number). So in this example:

Food	Cost $ | Quantity	
Bananas	1	20
Apples	0.5	10
Pears	0.8	20

table[1]['Food'] will return Bananas, as will table[1][0]

There are additional methods to locate a particular row or column for example table.get_row_by_field('Food', 'Bananas') returns {'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}.

Example Tests

Basic Example:

```
def test_buildings_list(self):
    buildings_list = BuildingsList(self, url=True)
```

..Note:

```
the self in BuildingsList(self, url=True) refers to the TestClass object.
We (have to) pass it in in order that the Page object can access its
functionality -- the web driver/browser specifically.
You **must** pass in self like this to use a Page object.
```

The above is a valid test to ensure that the Buildings list page loads. Instantiating a BuildingsList object with url=True causes the page to be populated with data and checks that it has been loaded. This happens in the __init__ method of BuildingsList so you get it for free when you use BuildingsList. All page objects have something similar).

It is a limited test however: though we have populated the page with data the test doesn't check the data is valid.

A better version of the test is this:

```
def test_buildings_list(self):
    buildings_list = BuildingsList(self, url=True)
    table = buildings_list.ensure_table_is_loaded()
    address = table.first_row['ADDRESS LINE 1']
    assert address.text == 'address'
```

Now the test checks some data is present as well (in this case 'address' is the default text set where a building is created if 'address_line_1' is not set).

Traversing Pages:

Often we need to test that a page, not only loads but can be reached by a user. Here is a complete example:

```
def test_building_list_tab_settings(self):
    """Make sure building list settings tab loads."""
    # load buildings list and create records
    buildings_list = BuildingsList(self, url=True)
    # locate setting link and click on it
    settings_link = buildings_list.find_element_by_id('list-settings')
    settings_link.click()

    # ensure settings page has loaded correctly.
    settings_page = BuildingListSettings(self)
    table = settings_page.ensure_table_is_loaded()
    assert table.first_row['COLUMN NAME'].text == 'Address Line 1'
```

In the first part of the test the Buildings list page is loaded and the BuildingsList page objects find_element_by_id method is used to locate and click on the appropriate record.

The second part creates an instance of BuildingListSettings page object, to check the page is loaded, then uses it to locate and check the table data.

..Note:

```
**Only** *self* is supplied as a parameter to the page object. When a page object is␣
↪used in this way it won't create any data and won't load the page directly. It will␣
↪however check the page has loaded.

Thus we can use it in this way to check that following a link loads the correct page␣
↪and that the data needed is already in place.
```

..Note:

```
If you navigate away form a page and then back you should call the reload()
method of the corresponding page object to ensure it has been loaded,
before interacting with it. Failing to do so will cause problems as
the browser might still be on the previous page.

This won't reload the page itself, you have to do that by following a link.
Because many pages in SEED are not actually separate pages, but rather
a new view of the same page constructed by an AJAX call there is no
way for the page object to do this (at least reliably).
```

Table Objects

..Note:

```
Tables and TableRow, TableColumn objects are all immutable by design.
```

Any time a page contains a (data) table the relevant page object *should* have a ensure_table_is_loaded method defined. (This is fairly trivial to do and will be covered later). This returns a Table object and *should* be used whenever you wish to check a table to ensure it contains the correct data (or that it has loaded correctly). Each table object can be though of containing two things, a set of headers and a collection of table rows.

Table headers consist of a list of the table headers stored as a list on the headers attribute: >>> table = MyPage.ensure_table_is_loaded() >>> print table.headers ['Food', 'Cost', Quantity']

Internally these are normally generated by a factory method that takes a Table web Element and returns a table object. This tries to work out what the table headers are by examining the table web element. If it can't do so (perhaps because the table header cell doesn't contain any text) it will substitute Col_0, Col_1 etc.

An individual table row can be accessed by index or by one of the convenience properties first_row and last_row: >>> assert table[0] == table.first_row

An individual table row consists of a TableRow object. The individual cells (`<td>...<.td>`) elements can be accessed by index or by key. The key is the relevant table header for that cell. Each cell is a <td> web element.

Example:

Food	Cost $ | Quantity	
Bananas	1	20
Apples	0.5	10
Pears	0.8	20

```
>>> table = MyPage.ensure_table_is_loaded()
>>> print table.headers
['Food', 'Cost', Quantity']
>>> print table[1]
{'Food': 'Apples', 'Cost': '0.5', 'Quantity: '10'}
>>> print table[1]['Food'].text
'Apples'
>>> print table[1][0].text
'Apples'
>>> print table.first_row
{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}
>>> print table.first_row['Food'].text
Bananas
>>> print table.last_row
{'Food': 'Pears', 'Cost': '0.8', 'Quantity: '20'}
>>> assert table.first_row['Food'].text == table[0][0].text
True
```

TableRow objects wrap OrderedDicts so have all the normal dictionary iter methods e.g.. values(), iteritems() etc. Comparisons and in methods work against the wrapped dict so work in the same way an OrderedDict would.

There are two other methods that can be used to retrieve table rows:

find_row_by_field and finds_row_by_field. The former is used to locate the first instance of a row where row[index] or row[header] matches a value. The latter returns a list of all rows that match:

```
>>> table.find_row_by_field('Food', 'Bananas')
{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}
>>> table.find_row_by_field('Food', 'Limes')
None
>>> table.find_rows_by_field(0, 'Bananas')
[{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}]
>>> table.find_rows_by_field('Food', 'Bananas')
[{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}]
```

There is a column method that returns a TableColumn object by its header value:

```
food_column = table.column('Food')
cost_column = table.column(1)
```

..Note:

```
an IndexError will be raised if the corresponding column is not found.
```

A table column is an immutable sequence with a header attribute.

```
>>> food_column.header
Food
>>> food_column
TableColumn('Food', ('Bananas', 'Apples', 'Pears')
>>> len(food_column)
3
>>> print food_column(1)
Apples
>>> 'Pears' in food_column
True
```

...Note: When comparing against a TableColumn the header is ignored and all comparators are coerced to tuples so you can compare against lists etc.

```
>>> food_column = ['Bananas', 'Apples', 'Pears']
True
>>> shopping_list = TableColumn('Shopping List', ['Bananas', 'Apples', 'Pears'])
>>> food_column == shopping_list
True
>>> food_column == shopping_list and food_column.header == shopping_list.header
False
```

All sequence methods work in a similar way (i.e.. !=, >, <, >=, <=)

Page Objects

Most of the methods on page objects are reflections of methods defined on Selenium web driver and the Selenium documentation should be consulted for information on these.

The exceptions are the wait_for_element methods which correspond to the equivalent find_element method, reload and ensure_table_is_loaded. The latter is only present on page objects that sub class Page and contain a table (see defining a Page object below for details of how this works).

Aside from page specific methods the base Page class provides some methods to set up the data needed to load a page. Normally these are called by the __init__ method of classes that sub-class Page so you don't have to call them directly. However there are occasionally times when you need to call them directly to set up the data for a page you will subsequently navigate to. Typically this occurs when you start on the main page before navigating elsewhere.

The methods are create_record and create_project.

create_record sets up an import record and file and optionally a canonical building and associated snapshot to go with them.

The import file and record are always created (in minimal form). To define any attributes on them pass in a dictionary to the import_record or import_file parameters.

To create a building set create_building to True and/or pass a dict to building.

create_project likewise creates a project and an associated canonical_building. You can supply a project name by passing name a string. Otherwise it will default to 'test'. A canonical building instance can be provided to building. Otherwise the page objects self.canonical_building will be used (assuming it has been set up by create_record)

In order to retrieve a canonical_building you can use the get_canonical_building method. This will return self.canonical_building (or None) if not the id parameter is supplied. If the optional id parameter is used the canoni-

cal_building with that id is returned. This is useful in cases where the canonical_building was created by another page object.

Defining Page Objects.

In most cases the appropriate page object should have already be defined in pages.py so you can import it there, so you should never need to call Page directly (though its perfectly possible for one offs).

If you don't find one its easy to define and add one to pages.py

Basic Example:

```
class Home(Page):
    def __init__(self, test_obj, use_url=None):
        if use_url:
            url = "/index.html"
        self.locator = Locator('NAME', 'my-button')
        super(Home, self).__init__(test_obj, locator, url=url)
        self.load_page()
```

This is all that is needed in simple cases. Most of it should be self explanatory. Be sure to call super before calling load_page. You need to always call the latter directly. Though you are free to add other, page specific methods, its typically to write an __init__ method and then rely on the methods defined by the Page base class.

self.locator is used to check that the page has been loaded correctly and **must** be defined. A Locator is a named tuple that consists of a strategy and a selector. Strategies are defined in page.py these correspond to Selenium waits.

Thus Locator('NAME', 'my-button') is equivalent to wait_for_element_by_name('my-button') which is like find_element_by_name('my-button') and looks for a element whose name attribute is 'my-button' e.g. click me

Where ever possible the Locator should be something that uniquely identifies the page. This is not always possible however. In these instances you can define a self.use_text attribute before calling super. This will provoke an additional check to ensure the page has loaded: The element identified by locator will also be checked to make sure it contains self.use_text (i.e. element .text == self.use_text).

Example with Table:

```
class Home(Page):
    def __init__(self, test_obj, use_url=None):
        if use_url:
            url = "/index.html"
        self.locator = Locator('NAME', 'my-button')
        self.table_locator = Locator('XPATH', '//table')
        super(Home, self).__init__(test_obj, locator, url=url)
        self.load_page()
```

If a page contains a table just add self.table_locator to give access to it. (Locator('XPATH', '//table') grabs the first table on the page and is often sufficient if there is not a better way of identifying it). Doing so causes the ensure_table_is_loaded method to be added to the Page object (Page actually defines __ensure_table_is_loaded, the double underscore causes name mangling so it can't be accessed directly. Behind the scenes Page sets ensure_table_is_loaded = None as a class attribute and in its __init__ method checks to to see if self.table_locator is defined. If it is it sets self.ensure_table_is_loaded to __ensure_table_is_loaded. This is done so an error will be raised if you try to access ensure_table_is_loaded on a page with out a table (locator).

The Page class uses the table locator to retrieve a table web element, this is fed to a factory method that returns a Table object when ensure_table_is_loaded is called. Normally this is sufficient. While the factory method does a good job in most cases there is a lot of variation in how data tables are constructed so its not practical to have it attempt to cover

all edge cases. In these cases you can override ensure_table_is_loaded. See BuildingListSettings in pages.py for an example.

Complex Example:

```
class DataMapping(Page):
    """
    Page object for the data mapping page

    dataset_id and create_import are mutually exclusive. dataset_id
    will take priority. The page will load directly (by url) if dataset_id
    or create_import are set. If import_record, import_record or building
    are supplied create_import will be set to True.

    :param: dataset_id: id of dataset (used in url)
    :param  create_import: create an import record before loading
    :param import_record: define additional attributes of the import record
    :param import_file: define additional attributes of the import file
    :param building: Add building if true, use dict for additional attributes

    :type: dataset_id: int
    :type: use_url: bool
    :type: create_import: bool
    :type: import_file: dict
    :type: import_record: dict
    :type: building: bool or dict
    """
    def __init__(self, test_obj, dataset_id=None, create_import=None,
                 import_record=None, import_file=None, building=None):
        locator = Locator('CLASS_NAME', 'mapping')
        # will cause ensure_table_is_loaded method to be added
        self.table_locator = Locator('CLASS_NAME', 'table')
        if import_record or import_file or building:
            create_import = True
        url = "app/#/data" if dataset_id or create_import else None

        super(DataMapping, self).__init__(
            test_obj, locator, url=url
        )

        # page set up
        if create_import and not dataset_id:
            create_building = True if building else False
            building = building if isinstance(building, dict) else None
            imports, canonical_building = self.create_record(
                create_building=create_building,
                import_record=import_record,
                import_file=import_file,
                building=building
            )
            if canonical_building:
                self.canonical_building = canonical_building
                self.building_id = self.canonical_building.id
            self.import_file = imports.import_file
            self.import_record = imports.import_record
            dataset_id = self.import_record.id
        if dataset_id:
            self.dataset_id = dataset_id
            self.url += "/{}".format(dataset_id)
```

```
        self.load_page()
```

This is a real example from pages.py and is about as complex as page objects need to get. Note that the create_record method is called **after** super but before load_page(). self.url is modified to add self.building_id here so that the page loads correctly.

base.py

This takes care of defining the base classes for use in tests.

Adding a new base class.

Add your class definition as normal, sub-classing `FunctionalLiveServerBaseTestCase` or one of the classes from derived from this.

Next add a factory function:

```
def myTestCaseFactory(browser):
    classname = get_classname('myTestCase', browser.name)
    return type(
        classname, (myTestCase, ),
        {'browser_type': browser}
    )
```

Then at the end of the file add a blank container dictionary and a call to your factory function in for loop:

```
MY_TEST_CLASSES = {}
for browser in BROWSERS:
    bname = browser.name
    MY_TEST_CLASSES[bname] = myTestCaseFactory(browser)
```

This will fill your container dictionary with Browser specific versions of your base class like this:

```
   {
       'Firefox': MyTestCaseFirefox,
       ...
   }

The container dictionary can then be imported in test_browser for use.
```

browser_definitions.py

This contains browser definitions and capabilities in order to set the right web driver on tests. They are used by the class factories in base.py and test generators in test_browser.py to generate browser specific versions of the test classes.

Browser definitions are listed in the BROWSERS list for easy import.

Adding a BrowserDefinition

A Browser definition is a named tuple that defines a browser for use in Test classes. e.g.:

```
myBrowser = BrowserDefinition(
    name = 'MyBrowserName',
    Capabilities = MyBrowserCapabilities,
    driver = MyBrowser.webdriver
)

or

myBrowser = BrowserDefinition(
    'MyBrowserName', MyBrowserCapabilities, MyBrowser.webdriver
)
```

Definitions:

- name is a string, e.g. browser name (and version).
- Capabilities is a dictionary that will be passed to the remote webdriver via Travis (which passes it to Sauce Labs) Describing it further is out of scope for this document. see: https://wiki.saucelabs.com/display/DOCS/Test+Configuration+and+Annotation
- webdriver will used to power the tests if they are run locally. Normally this can just be MyBrowser.webdriver, but you can define any function and pass it in here. Note for both it *must* be func not func() or webdriver not webdriver()

A browser capabilities factory is provided for your convenience in base.py. This should ensure a valid browser capacity definition.

Depending on whether the tests are running locally or on Travis Capabilities or driver will be used.

Make sure to add your browser definition to BROWSERS (or equivalent) (and your capacity to BROWSER_CAPACITIES).

page.py

This defines the Page object base class, TableRow, TableColumn and supporting functionality.

pages.py

Page specific page objects. Import from here.

Adding a Page object subclass

See Page objects above

test_browser.py

The tests themselves live here.

The Gory Details

There is a lot of indirection and dynamic definition going on underneath the hood (compounded by the fact that there's a lot of deep magic going on with unit tests in the first place). However I tried to write in such a way that it uses

common idioms for things that will be changed frequently so mostly this can be ignored. Explanations for what is going on can be found below if you want, or need, to know.

The rationale for all this is easy testing across multiple browsers. As long as we wrap them in the right way we need only to write our base and test classes once and we will get a set of tests for each browser definition with out having to worry about the definitions, if a new definition is added it will automatically get picked up by all tests.

base.py details

This contains the base class definition `FunctionalLiveServerBaseTestCase` as well as other classes derived from it. The thing to note about this is that the setUp method detects the environment the tests are running in in the setUp method and uses this to add the correct browser instance to the class instance.

At the end of the file this list is looped over and the browser definition passed to a factory function. This takes the base class and returns a browser specific version of it so TestCaseFactory(browser) returns TestCaseFactoryBrowser which is added to a dictionary that can be imported elsewhere.

test_browsers.py details

This contains the actual tests themselves. It imports BROWSERS and the browser class dictionaries e.g. LOGGED_OUT_CLASSES from base.by.

The Test Classes defined here live inside a generator.

Each generator loops over BROWSERS and subclasses the appropriate base class from the browser class dictionary. It would be nice if we could yield this test class instance directly and pass it to the test runner. Unfortunately unittest expects a function from a generator. Actually things are a little more complicated than this. If you yield an object that's not a function (i.e. everything else) it looks for the presence of a runTest method on it. If it find one it decides its a test and will call the object directly (not the runTest method) so yielding a class is like calling Class(), i.e. it instantiates the class but doesn't call the test_methods.

To get around this the generator instantiates a copy of that class and passes it to get_tsts(). This takes care of calling setUpClass, which would not otherwise be run, then inspects the object for test method. When it finds one it wraps a call to that method in a function that takes care of invoking setUp and tearDown before, and after its run. Since it is now a function this can be safely yielded by the generator to be invoked by the test runner.

Base

Page

This module defines the functionality needed to create Page objects.

A Page object is a way of representing the page being tested separately from the unit test, and contains methods that the unit test can use to interact with that page, and the elements it contains.

A Page object needs to be supplied a web driver (browser instance) on initialization: this is done by passing a Test Case instance (that inherits from a class defined in base.py). This is done so Page objects can also utilize some of its other functionality and attributes, as well as the browser instance.

Note a Page object should not *directly* contain any functionality necessary to set up the page, e.g. database interactions. These methods remain with the Test Case classes defined in base.py. However they may be accessed by classes that subclass Page by calling the test_obj method so that subclasses can handle setting up a page. Since creating a building snapshot is so common there is a create_building method provided by Page.

Example

Sub Classing Page

This is the preferred method and should be used any time a page will be used in multiple tests.

Defining the page object

class Home(Page):

def __init__(self, test_obj): url = "index.html" locator = Locator('NAME', 'my-button') super(Home, self).__init__(test_obj, locator, url=url) self.load_page()

> **Warning:** a locator must be defined and passed to super(Class, self).__init__

Calling the page object in a test

:: from seed.functional.tests.browser_definitions import import BROWSERS from seed.functional.tests.base import LOGGED_IN_CLASSES from seed.functional.tests.pages import Home

def my_tests_generator(): for browser in BROWSERS:

class Tests((LOGGED_OUT_CLASSES[browser.name]):

def my_test(self): home_page = Home(self) my_element = home_page.find_element_by_name('example') assert my_element.text = 'example text'

Example

Sub Classing the Page Object for a page with a table

class Home(Page):

def __init__(self, test_obj): url = "index.html" locator = Locator('NAME', 'my-button') # will cause ensure_table_is_loaded method to be added self.table_locator = Locator('XPATH', '//table') super(Home, self).__init__(test_obj, locator, url=url) self.load_page()

Calling the page object in a test

:: from seed.functional.tests.browser_definitions import import BROWSERS from seed.functional.tests.base import LOGGED_IN_CLASSES from seed.functional.tests.pages import Home

def my_tests_generator(): for browser in BROWSERS:

class Tests((LOGGED_OUT_CLASSES[browser.name]):

def my_test(self): home_page = Home(self) table = home.page.ensure_table_is_loaded() assert table[0][0].text = 'example text'

Example

Calling Page directly

:: from seed.functional.tests.browser_definitions import import BROWSERS from seed.functional.tests.base import LOGGED_IN_CLASSES from seed.functional.tests.page import Locator, Page

def my_tests_generator(): for browser in BROWSERS:

class Tests((LOGGED_OUT_CLASSES[browser.name]):

def my_test(self): url = "{}/home.html".format(self.live_server_url) my_locator = Locator('NAME', 'my-button') my_page = Page(self, my_locator, url=url) my_page.load_page() my_element = my_page.find_element_by_name('example') assert my_element.text = 'example text'

loads the next page, so we don't need a url my_element.click() my_other_locator = Locator('IF', 'my-id') my_other_page = Page(self, my_locator) my_other_element = my_other_page.find_element_by_id('example') assert my_other_element.text = 'example text'

:author Paul Munday<paul@paulmunday.net>

class seed.functional.tests.page.**Imports**(*import_file*, *import_record*)

import_file
Alias for field number 0

import_record
Alias for field number 1

class seed.functional.tests.page.**Locator**(*strategy*, *search_term*)

search_term
Alias for field number 1

strategy
Alias for field number 0

class seed.functional.tests.page.**Organization**(*org*, *sub_orgs*)

org
Alias for field number 0

sub_orgs
Alias for field number 1

class seed.functional.tests.page.**Page**(*test_obj*, *locator*, *url=None*, *timeout=None*, *use_text=None*)

A web page under test, :param: locator: Locator object used to identify page :param: url: if True load by url, if False assume page is loaded :param: timeout: time out in seconds :param: use_text: if a string is supplied its value will be checked on locator

Tpye locator: Locator object

Type url: Bool or string, if string append to self.url.

Type timeout: int or float

Type use_text: string

create_project(*name=None*, *building=None*)
Create a project (and project building). If no building is supplied self.canonical_building will be used if present.

Param name: project name

Param

building canonical building

Type name: string

Type building: CanonicalBuilding instance

create_record(*create_building=False*, *import_record=None*, *import_file=None*, *building=None*)
Set up an import/building snapshot in the db.

Pass dictionaries to import_file, import_record and building to set additional attributes. e.g. import_record= {'name': 'Name'} can be used to set the import record/data set name.

As ImportFile.file is a django.db.models.FileField field it can be tricky to manipulate/mock. Use mock_file_factory from base.py to generate a mock_file and add it to the import_file dict. This will be added after the record is created and can be used to set the file name etc.

Parameters

- **create_building** – create a building snapshot
- **import_file** – define additional attributes of the import file
- **building** – define additional attributes of the building snapshot

Param import_record: define additional attributes of the import record

Type create_building: Bool

Type import_record: dict

Type import_file: dict

Type building: dict

generate_buildings(*num*, *import_file=None*, *building_details=None*)
Create multiple buildings.

get_canonical_building(*id=None*)
Get canonical building, by id or associated with Page, (self.canonical_building)

Param id: building id/CanonicalBuilding primary key

Type id: int

Returns CanonicalBuilding instance or None

reload()
Reload the page if the browser has navigated away. i.e. another Page instance has been created.

> **Warning:** This method does not navigate back to the page. You will need to do so yourself in your test. e.g. by locating and clicking on a suitable link.
>
> It is not possible to reliably use self.browser.get(self.url) as get() watches for the page onload event and this does not happen on many pages as page loads are effectively overwritten as an ajax call once the page has been loaded the first time.

wait_for_element (*strategy*, *search*, *timeout=None*)
Get a page element, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_class_name (*selector*, *timeout=15*)
Get a page element by class name, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_css_selector (*selector*, *timeout=15*)
Get a page element by css, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_id (*selector*, *timeout=15*)
Get a page element by id, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_link_text (*selector*, *timeout=15*)
Get a page element by link_test, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_partial_link_text (*selector*, *timeout=15*)
Get a page element by partial_link_test, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_tag_name (*selector*, *timeout=15*)
Get a page element by tag name, allowing time for the page to load.

:returns WebElement.

wait_for_element_by_xpath (*selector*, *timeout=15*)
Get a page element by xpath, allowing time for the page to load.

:returns WebElement.

class seed.functional.tests.page.**Table** (*headers*, *rows*, *safe=True*)
Provides a convenient way to query/compare a table.

Param headers: Table headers

Param rows: Table rows

Param safe: if True raise an exception if number of columns and headers differs

Type headers: Sequence (list, tuple etc)

Type rows: Sequence of sequences/TableRows/OrderedDicts

Type safe: bool

Example

Food	Cost $ | Quantity	
Bananas	1	20
Apples	0.5	10
Pears	0.8	20

```
>>> table = MyPage.ensure_table_is_loaded()
>>> print table.headers
['Food', 'Cost', Quantity']
>>> print table[1]
```

```
{'Food': 'Apples', 'Cost': '0.5', 'Quantity: '10'}
>>> print table[1]['Food'].text
'Apples'
>>> print table[1][0].text
'Apples'
>>> print table.first_row
{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}
>>> print table.first_row['Food'].text
Bananas
>>> print table.last_row
{'Food': 'Pears', 'Cost': '0.8', 'Quantity: '20'}
>>> print table.column(0)
('Food', ['Bananas', 'Apples', 'Pears'])
>>> print table.column(0).header_text
Food
>>> print table.column(0)[0].text
Bananas
>>> print table.column('Food')
('Food', ['Bananas', 'Apples', 'Pears'])
>>> print len(table)
3
>>> print len(table.first_row)
3
>>> table.first_row.values()
['Bananas', '1', '20']
>>> table.first_row.keys()
['Food', 'Cost', Quantity']
>>> for key, val in table.first_row.items():
    >>>    print "{} = {}".format(key, val)
Food = Bananas etc.
>>> table.find_row_by_field('Food', 'Bananas')
{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}
>>> table.find_row_by_field('Food', 'Limes')
Noneed/functional/tests/page.py
>>> table.find_rows_by_field(0, 'Bananas')
[{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}]
>>> table.find_rows_by_field('Food', 'Bananas')
[{'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}]
>>> expected = Table(
    ['Food', 'Cost', Quantity'],
    [['Bananas', '1', '20'], ['Pears, 0.8, 20]]
)
>>> print expected.first_row
    {'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'}
>>> expected.first_row == table.first_row
True
>>> expected == table
False
>>> table.first_row in expected
True
>>> table[1] in expected
False
>>> {'Food': 'Apples', 'Cost': '0.5', 'Quantity': '10'} in table
True
>>> [row for row in table]
[TableRow({'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'})...]
```

Instantiating: Typically you use the page objects ensure_table_is_loaded() method to load the page:: >>> table

= MyPage.ensure_table_is_loaded()

This does the equivalent of this:: >>> table_web_element = year_ending = page.find_element_by_xpath("//table") >>> table = table_factory(table_web_element, webdriver)

You can also instantiate it directly, if, for instance you have a lot of values you want to test against:

```
from collections import OrderedDict
table = Table(
    ['Food', 'Cost', Quantity'],
    OrderedDict({'Food': 'Bananas', 'Cost': '1', 'Quantity: '20'})
)
```

Tuples or lists can be used. OK:

```
Table(
["col 1", "col 2", "col 3"],
[
    [1, 2, 3],
    [4, 5, 6]
]
Table(
    ["a", "b]",
    [
        TableRow(OrderedDict({'a': 0, 'b': 1})),
        OrderedDict({'a': 2, 'b': 3}),
        TableRow([('a', 4), ('b':5)]
    ]
)
```

Warning: This is probably not what you want:

```
Table(
    ["a", "b]",
    [[('a', 1), ('b':2)], [('a', 4), ('b':5)]]
)
```

Equivalent to:

```
Table(
    ["a", "b]",
    [
        TableRow(OrderedDict({
            'a': [('a', 1), ('b':2)]
            'b': [('a', 4), ('b':5)]
        })
    ]
)
```

column (*idx*)
Return a table column corresponding to idx header,

A table column is always returned if a column exists, but its values are set to None where row[idx] is not present.

An index error will be raised if column corresponding to idx is not present.

find_row_by_field (*idx*, *value*)
Returns first row where row[idx].text = value, or None

Param idx: index of column/cell or column name

Param value: return rows whose text matches this value

Type idx: int/string

Type value: string

find_rows_by_field(*idx*, *value*)

Returns all rows where row[idx].text = value, or an empty list

Param idx: index of column/cell or column name

Param value: return rows whose text matches this value

Type idx: int/string

Type value: string

class seed.functional.tests.page.**TableColumn**(*header*, *elements=None*)

An immutable sequence to represent a table column. The table header is stored in the table attribute.

Comparisons: Header is not compared with comparisons. All comparisons are coerced to tuples so you can compare against any seq.

Examples:: >>> col0 = TableColumn('cost', [1,2,3]) >>> col0.header cost >>> 1 in col0 True >>> col0 == (1, 2, 3) True >>> col0 == [1, 2, 3] True >>> col1 = TableColumn('price', [1,2,3]) >>> col0 == col1 True >>> col0 == col1 and coll0.header == col1.header False

class seed.functional.tests.page.**TableRow**(*constructor*, ***kwargs*)

TableRow is an immutable variant of an OrderedDict whose values can also be accessed by index (i.e. as if they were a list) when an int is supplied.

In order to achieve this all key values are coerced to strings. While it will take numbers as keys in order to access them using [key] you must supply key as a string. However .get() performs the same coercion so e.g. .get(1) will work with the caveat that an error will not be raised if the key is not found.

In order to prevent accidental clobbering of keys a KeyError will be raised if the number of keys is not the same as that supplied. i.e. TableRow({'1', 1, 1: 2}) will throw an error. However TableRow({'a': 1, 'a': 2}) will also throw an error, where creating a dict of the same would result in {'a': 2}.

Example

```
>>> tr = TableRow(OrderedDict({'a': 0, 'b': 1}))
>>> tr['a']
0
>>> tr[0]
0
>>> tr
TableRow({'a': 0, 'b': 1})
>>> tr = TableRow([(1, 2), ('2', 3)])
>>> tr['2']
3
>>> tr[0]
2
>>> tr[1]
3
>>> tr.get(1)
2
>>> tr.get(3)
None
>>> tr = TableRow({'a': 0, 'a': 1})
KeyError
```

```
>>> tr = TableRow([(2, 2), ('2', 3)])
KeyError
```

seed.functional.tests.page.**table_factory**(*table*)

Constructs a Table instance from a table web element.

It can be difficult to locate elements in tables in selenium, unless you have a name or id. Typically you would have to locate e.g. a table cell by using an XPath. These can be long and difficult to figure out since you need to locate row and column by a number corresponding to their position and this is difficult to figure out since the table structure can vary.

This avoids this issue by taking a table Web Element and then examining if for table headers. Based on what it finds in iterates through the table rows and converts these into ordered dictionaries where the key is the value of the table header and the value is the Web Element that represents the corresponding table cell.Where possible it tries to determine the table_header by examining the inner text contained in a header cell. If it is unable to do so it will instead use Col_0 etc

Returns instance of Table

Pages

This module defines Page objects representing view in SEED.

Example

Defining a page object

Class definition:

```
class Home(Page):
    def __init__(self, test_obj):
        url = "/index.html"
        self.locator = Locator('NAME', 'my-button')
        super(Home, self).__init__(test_obj, locator, url=url)
    self.load_page()
```

Warning: a locator must be defined and passed to super(Class, self)__init__

Calling the page object in a test:

```
from seed.functional.tests.browser_definitions import import BROWSERS
from seed.functional.tests.base import LOGGED_IN_CLASSES
from seed.functional.tests.pages import Home


def my_tests_generator():
    for browser in BROWSERS:

        class Tests((LOGGED_OUT_CLASSES[browser.name]):

        def my_test(self):
            home_page = Home(self)
            my_element = home_page.get_element_by_name('example')
            assert my_element.text = 'example text'
```

Example

Defining the Page Object for a page with a table

Class definition:

```
class Home(Page):
    def __init__(self, test_obj):
        url = "index.html"
        locator = Locator('NAME', 'my-button')
        # will cause ensure_table_is_loaded method to be added
        self.table_locator = Locator('XPATH', '//table')
        super(Home, self).__init__(test_obj, locator, url=url)
        self.load_page()
```

Calling the table object in a test:

```
from seed.functional.tests.browser_definitions import import BROWSERS
from seed.functional.tests.base import LOGGED_IN_CLASSES
from seed.functional.tests.pages import Home


def my_tests_generator():
    for browser in BROWSERS:

        class Tests((LOGGED_OUT_CLASSES[browser.name]):

        def my_test(self):
            home_page = Home(self)
            table = home.page.ensure_table_is_loaded()
            assert table[0][0].text = 'example text'
```

author Paul Munday<paul@paulmunday.net>

class seed.functional.tests.pages.**AccountsPage**(*test_obj*, *use_url=None*, *create_import=None*, *import_file=None*, *building=None*, *sub_org=None*)

Bases: *seed.functional.tests.page.Page*

Page object for the Organizations/account page. app/#/accounts

An import record and building will be created if url is True. An accounts page object will have an org attribute. This is self.org from the test case. It may also have a 'sub_org' attribute. It is only present if a sub_org exists. The sub_org will either be test_obj.sub_org if it already exists when the object is initialized, otherwise the first sub_org created.

Param use_url: if True load by url, if False assume page is loaded

Create_import create an Import record (with building) if true

Parameters

- **import_file** – define additional attributes of the import file
- **building** – define additional attributes of the building snapshot

Sub_org create a SubOrganization if True or string(name) or list (names).

Type url: bool ,None.

Type create_import: bool, None

Type import_file: dict

Type building: dict

Type sub_org: bool, string

create_sub_org(*name*)
Create a sub organization

get_managed_org_tables()
Return a list of managed(owned) organizations.

Each organization is a named tuple of type Organization. Organization.org is a TableRow for the organization. Organization.sub_orgs is a list containing sub-organizations if any.

If there are sub orgs/the org is parent org, or capable of being so, the first row of sub orgs is a TableRow whose first cell is the (header) text 'Sub-Organizations', and whose second contains the 'Create new sub-organization' link.

Then any sub orgs will be listed. All TableRows have two parts: the first cell contains the organization name (or 'Sub-Organizations') the second the controls i.e. links to 'Settings' and 'Members' if its a sub/child organization, 'Settings', 'Sharing', 'Data', 'Cleansing', 'Sub-Organizations', 'Members' if it's a parent organization.

The keys are 'ORGANIZATION' and 'Col_1' respectively.

get_member_orgs_table()
Return a the table of organizations the user belongs to.

class seed.functional.tests.pages.**BuildingInfo**(*test_obj*, *building_id=None*, *create_building=None*, *create_project=None*, *import_file=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the building details page.

The page will load directly(by url) if create_record or a building_id is set.

Pass a dictionary to create_building to define additional attributes of building record/snapshot.

building_id and create_building are mutually exclusive. building_id will take priority. If create project is true and there is a building (i.e. create_building or building id is True), it will be added to the project.

Param building_id: Canonical building id, page will load by url if set.

Param create_building: Create a building when instantiating.

Param create_project: Create a project . Supply string for project name.

Parameters **import_file** – define additional attributes of the import file.

Type building_id: int

Type create_building: bool, None or dict

Type create_project: bool, string, None

Type import_file: dict

class seed.functional.tests.pages.**BuildingLabels**(*test_obj*, *use_url=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the building reports page.

Param use_url: load page directly by url

Type use_url: bool

class seed.functional.tests.pages.**BuildingListSettings**(*test_obj*, *use_url=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the building list settings page.

Param use_url: load page directly by url

Type use_url: bool

ensure_table_is_loaded()
Page uses stacked table to get fixed header, so needs own method.

class seed.functional.tests.pages.**BuildingProjects**(*test_obj*, *id=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the building reports page.

Param id: building_id, page will load directly(by url) if supplied.

Type id: int

class seed.functional.tests.pages.**BuildingReports**(*test_obj*, *use_url=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the building reports page.

Param use_url: load page directly by url

Type use_url: bool

class seed.functional.tests.pages.**BuildingsList**(*test_obj*, *url=None*, *import_file=None*, *building=None*, *num_buildings=0*)
Bases: *seed.functional.tests.page.Page*

Page object for the buildings list page.

An import record and building will be created if url is True.

Param url: if True load by url, if False assume page is loaded

Parameters

- **import_file** – define additional attributes of the import file
- **building** – define additional attributes of the building snapshot

Num_buildings number of additional buildings to create (assumes use_url)

Type url: bool or string, if string append to self.url.

Type import_file: dict

Type building: dict

Type num_buildings: int

class seed.functional.tests.pages.**DataMapping**(*test_obj*, *dataset_id=None*, *create_import=None*, *import_record=None*, *import_file=None*, *building=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the data mapping page

dataset_id and create_import are mutually exclusive. dataset_id will take priority. The page will load directly (by url) if dataset_id or create_import are set. If import_record, import_record or building are supplied create_import will be set to True.

Param dataset_id: id of dataset (used in url)

Parameters

- **create_import** – create an import record before loading
- **import_record** – define additional attributes of the import record
- **import_file** – define additional attributes of the import file
- **building** – Add building if true, use dict for additional attributes

Type dataset_id: int

Type use_url: bool

Type create_import: bool

Type import_file: dict

Type import_record: dict

Type building: bool or dict

class seed.functional.tests.pages.**DataSetInfo**(*test_obj*, *dataset_id=None*, *create_import=None*, *import_record=None*, *import_file=None*, *building=None*)

Bases: *seed.functional.tests.page.Page*

Page object for the data set info page

dataset_id and create_import are mutually exclusive. dataset_id will take priority. The page will load directly (by url) if dataset_id or create_import are set. If import_record, import_record or building are supplied create_import will be set to True.

Param dataset_id: id of dataset (used in url)

Parameters

- **create_import** – create an import record before loading
- **import_record** – define additional attributes of the import record
- **import_file** – define additional attributes of the import file
- **building** – Add building if true, use dict for additional attributes

Type dataset_id: int

Type use_url: bool

Type create_import: bool

Type import_file: dict

Type import_record: dict

Type building: bool or dict

class seed.functional.tests.pages.**DataSetsList**(*test_obj*, *create_import=None*, *import_file=None*, *building=None*)

Bases: *seed.functional.tests.page.Page*

Page object for the data sets list page.

The page will load directly (by url) if create_import is set.

Parameters

- **create_import** – create an import record before loading

- **import_file** – define additional attributes of the import file
- **building** – Add building if true, use dict for additional attributes

Type use_url: bool

Type create_import: bool

Type import_file: dict

Type building: bool or dict

class seed.functional.tests.pages.**LandingPage**(*test_obj*, *use_url=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the Landing (Front/Home) Page.

Param use_url: if True load by url, if False assume page is loaded

Type use_url: bool

class seed.functional.tests.pages.**MainPage**(*test_obj*, *use_url=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the Main Page. /app/#/

Param use_url: if True load by url, if False assume page is loaded

Type use_url: bool

class seed.functional.tests.pages.**ProfilePage**(*test_obj*, *use_url=None*, *section=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the Profile Page. /app/#/profile

Param use_url: if True load by url, if False assume page is loaded

Param section: Which tab will be loaded. Default = profile

Type use_url: bool.

Type section: string one of profile/security/developer, case insensitive.

get_api_key_table()
Return API Key table.

reload(*section=None*)
Set section(use_text) before reloading. :param: section: Which tab will be loaded. Default = previous.

Type section: None/string one of profile/security/developer.

class seed.functional.tests.pages.**ProjectBuildingInfo**(*test_obj*, *use_url=False*, *name=None*, *building_id=None*, *create_building=None*, *create_project=None*, *import_file=None*)
Bases: *seed.functional.tests.page.Page*

Page object for the Project building information page.

This is largely the same as the BuildingInfo page

The page will load directly(by url) if create_project or use_url is True.

If use_url is set you *must* supply name and building id. If use_url is not set you *must* supply name.

Pass a dictionary to create_building to define additional attributes of building record/snapshot.

building_id and create_building are mutually exclusive. building_id will take priority. If create project is true a building will be added to the project. Therefore either create_building must be True or building_id must be set.

Param use_url: if True load by url, if False assume page is loaded

Param name: Project name.

Param building_id: Canonical building id

Param create_building: Create a building when instantiating.

Param create_project: Create a project.

Parameters **import_file** – define additional attributes of the import file.

Type use_url: bool

Type name: str

Type building_id: int

Type create_building: bool, None or dict

Type create_project: bool, None

Type import_file: dict

class seed.functional.tests.pages.**ProjectPage**(*test_obj*, *name=None*, *building_id=None*, *create_building=None*, *create_project=None*, *import_file=None*)

Bases: *seed.functional.tests.page.Page*

Page object for the Projects Page. /app/#/projects

Pass a dictionary to create_building to define additional attributes of building record/snapshot.

building_id and create_building are mutually exclusive. building_id will take priority. If create project is true and there is a building (i.e. create_building or building id is True), it will be added to the project.

Param name: Project name. If set will load by url appending name to url

Param building_id: Canonical building id to add to project.

Param create_building: Create a building when instantiating.

Param create_project: Create a project.

Parameters **import_file** – define additional attributes of the import file.

Type name: str

Type building_id: int

Type create_building: bool, None or dict

Type create_project: bool, None

Type import_file: dict

class seed.functional.tests.pages.**ProjectsList**(*test_obj*, *use_url=None*, *building_id=None*, *create_building=None*, *create_project=None*, *import_file=None*)

Bases: *seed.functional.tests.page.Page*

Page object for the Projects Page. /app/#/projects

Param use_url: if True load by url, if False assume page is loaded

Param building_id: Canonical building id to add to project.

Param create_building: Create a building when instantiating.

Param create_project: Create a project. Supply string to set name

Parameters **import_file** – define additional attributes of the import file.

Type use_url: bool

Type building_id: int

Type create_building: bool, None or dict

Type create_project: bool, None

Type import_file: dict

Admin Views

class seed.tests.test_admin_views.**AdminViewsTest**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_add_org()
Happy path test for creating a new org.

test_add_org_dupe()
Trying to create an org with a dupe name fails.

test_add_user_existing_org()
Test creating a new user, adding them to an existing org in the process.

test_add_user_new_org()
Create a new user and a new org at the same time.

test_add_user_no_org()
Shouldn't be able to create a new user without either selecting or creating an org at the same time.

test_signup_process()
Simulates the entire new user signup process, from initial account creation by an admin to receiving the signup email to confirming the account and setting a password.

test_signup_process_force_lowercase_email()
Simulates the signup and login forcing login username to lowercase

Decorators

class seed.tests.test_decorators.**ClassDecoratorTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

test_ajax_request_class_dict()

test_ajax_request_class_dict_status_error()

test_ajax_request_class_dict_status_false()

test_ajax_request_class_format_type()

test_require_organization_id_class_no_org_id()

test_require_organization_id_class_org_id()

test_require_organization_id_class_org_id_not_int()

class seed.tests.test_decorators.**RequireOrganizationIDTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_require_organization_id_fail_no_key()

test_require_organization_id_fail_not_numeric()

test_require_organization_id_success_integer()

test_require_organization_id_success_string()

class seed.tests.test_decorators.**TestDecorators**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests for locking tasks and reporting progress.

locked = 1

pk = 34

setUp()

test_get_prog_key()
We format our cache key properly.

test_increment_cache()
Sum our progress by increments properly.

test_locking()
Make sure we indicate we're locked if and only if we're inside the function.

test_locking_w_exception()
Make sure we release our lock if we've had an exception.

test_progress()
When a task finishes, it increments the progress counter properly.

unlocked = 0

exception seed.tests.test_decorators.**TestException**
Bases: exceptions.Exception

Exporters

class seed.tests.test_exporters.**TestExporters**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests for exporting data to various formats.

setUp()

tearDown()

test_csv_export()
Ensures exported CSV data matches source data

test_csv_export_extra_data()
Ensures exported CSV data matches source data

test_data_model_assumptions()
Some parts of export make certain assumptions about the data model, this test ensures that those assumptions are true.

test_xls_export()
Ensures exported XLS data matches source data

Models

Tasks

class seed.tests.test_tasks.**TestTasks**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests for dealing with SEED related tasks.

setUp()

test_delete_organization()

test_delete_organization_buildings(**args*, ***kwargs*)
tests the delete buildings for an organization

test_delete_organization_doesnt_delete_user_if_multiple_memberships()
Deleting an org shouldn't delete the orgs users if the user belongs to many orgs.

Views

class seed.tests.test_views.**BuildingDetailViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests of the SEED Building Detail page

setUp()

test_get_match_tree(**args*, ***kwargs*)
tests get_match_tree

test_get_match_tree_from_child(**args*, ***kwargs*)
tests get_match_tree from the child

test_get_property(**args*, ***kwargs*)

test_get_taxlot(**args*, ***kwargs*)

test_save_match_audit_log(**args*, ***kwargs*)
tests that a building match logs an audit_log

test_save_match_invalid_org()
tests that a building match checks perm of org id

test_save_match_wrong_perms_different_building_orgs()
tests that a building match is valid for BS orgs

test_save_match_wrong_perms_org_id()
tests that a building match is valid for the org id

test_save_unmatch_audit_log(**args*, ***kwargs*)
tests that a building unmatch logs an audit_log

test_update_building_audit_log(**args*, ***kwargs*)
tests that a building update logs an audit_log

class seed.tests.test_views.**DefaultColumnsViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests of the SEED default custom saved columns

setUp()

tearDown()

test_get_columns()

test_get_default_columns_initial_state()

test_get_default_columns_with_set_columns()

test_set_default_columns()

class seed.tests.test_views.**GetDatasetsViewsTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_delete_dataset()

test_get_dataset()

test_get_datasets()

test_get_datasets_count()

test_get_datasets_count_invalid()

test_update_dataset()

class seed.tests.test_views.**ImportFileViewsTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_delete_duplicates_from_import_file()

test_delete_file()

test_get_import_file()

test_get_matching_results()

class seed.tests.test_views.**InventoryViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

tearDown()

test_get_cycles()

test_get_properties()

test_get_properties_cycle_id()

test_get_properties_empty_page()

test_get_properties_page_not_an_integer()

test_get_properties_property_extra_data()

test_get_properties_taxlot_extra_data()

test_get_properties_with_taxlots()

test_get_property()

test_get_property_columns()

test_get_property_history()

test_get_property_multiple_taxlots()

test_get_taxlot()

test_get_taxlot_columns()

test_get_taxlots()

test_get_taxlots_empty_page()

test_get_taxlots_extra_data()

test_get_taxlots_missing_jurisdiction_tax_lot_id()

test_get_taxlots_multiple_taxlots()

test_get_taxlots_no_cycle_id()

test_get_taxlots_page_not_an_integer()

class seed.tests.test_views.**MainViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_export_buildings()

test_export_buildings_empty()

test_export_buildings_progress()

test_home()

class seed.tests.test_views.**MatchTreeTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Currently only tests _parent_tree_coparents

setUp()

test_get_coparents()

test_parent_tree_coparents()
Tests that _parent_tree_coparents returns what we expect

class seed.tests.test_views.**ReportViewsTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_get_aggregated_building_report_data(**args*, ***kwargs*)

test_get_aggregated_inventory_report_data(**args*, ***kwargs*)

test_get_building_report_data(**args*, ***kwargs*)

test_get_building_summary_report_data()

test_get_inventory_report_data(**args*, ***kwargs*)

class seed.tests.test_views.**SearchBuildingSnapshotsViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_search_mapping_results(**args*, ***kwargs*)

class seed.tests.test_views.**SearchViewTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Tests of the SEED search_buildings

setUp()

test_search_active_canonicalbuildings(**args*, ***kwargs*)
tests the search_buildings method used throughout the app for only returning active CanonicalBuilding BuildingSnapshot instances.

test_search_case_insensitive_exact_match(**args*, ***kwargs*)
Tests search_buildings method when called with a case insensitive exact match.

test_search_empty_column(**args*, ***kwargs*)
Tests search_buildings method when called with an empty column query.

test_search_exact_exclude_filter(**args*, ***kwargs*)

test_search_exact_match(**args*, ***kwargs*)
Tests search_buildings method when called with an exact match.

test_search_exclude_filter(**args*, ***kwargs*)

test_search_extra_data(**args*, ***kwargs*)
tests the search_buildings method used throughout the app for only returning active CanonicalBuilding BuildingSnapshot instances.

test_search_extra_data_empty_column(**args*, ***kwargs*)
Empty column query on extra_data key should match key's value being empty in JsonField.

test_search_extra_data_exact_exclude_filter(**args*, ***kwargs*)

test_search_extra_data_exact_match(**args*, ***kwargs*)
Exact match on extra_data json keys

test_search_extra_data_exclude_filter(**args*, ***kwargs*)

test_search_extra_data_non_empty_column(**args*, ***kwargs*)
Not-empty column query on extra_data key.

test_search_extra_data_non_existent_column(**args*, ***kwargs*)
Empty column query on extra_data key should match key not existing in JsonField.

test_search_filter_date_range_ISO8601(**args*, ***kwargs*)

test_search_filter_range()
Tests search_buildings method when called with a range.

test_search_not_empty_column(**args*, ***kwargs*)
Tests search_buildings method when called with a not-empty column query.

test_search_sort(**args*, ***kwargs*)
tests the search_buildings method used throughout the app for only returning active CanonicalBuilding BuildingSnapshot instances.

test_sort_extra_data(**args*, ***kwargs*)
Tests that sorting on extra data takes the column type into account.

class seed.tests.test_views.**TestMCMViews**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

assert_expected_mappings(*actual*, *expected*)
For each k,v pair of form column_name: [dest_col, confidence] in actual, assert that expected contains the same column_name and dest_col mapping.

expected_mappings = {u'year built': [u'year_built', 50], u'building id': [u'Building air leakage', 64], u'name': [u'Na

raw_columns_expected = {u'status': u'success', u'raw_columns': [u'name', u'address', u'year built', u'building id']}

setUp()

test_create_dataset()
tests the create_dataset view, allows duplicate dataset names

test_get_column_mapping_suggestions()

test_get_column_mapping_suggestions_pm_file()

test_get_column_mapping_suggestions_with_columns()

test_get_raw_column_names()
Good case for `get_raw_column_names`.

test_progress()
Make sure we retrieve data from cache properly.

test_reset_mapped_w_matching_done(**args*, ***kwargs*)
Make sure we don't delete buildings that have been merged.

test_reset_mapped_w_previous_matches(**args*, ***kwargs*)
Ensure we ignore mapped buildings with children BuildingSnapshots.

test_save_column_mappings()

test_save_column_mappings_idempotent()
We need to make successive calls to save_column_mappings.

Tests

class seed.tests.tests.**ComplianceTestCase**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

test_basic_compliance_creation()

class seed.tests.tests.**ProjectTestCase**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

test_basic_project_creation()

class seed.tests.tests.**UtilsTests**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

setUp()

test_get_buildings_count_for_user()

Utils

class seed.tests.util.**FakeClient**
Bases: object

An extremely light-weight test client.

get (*view_func*, *data*, *headers=None*, ***kwargs*)

post (*view_func*, *data*, *headers=None*, ***kwargs*)

class seed.tests.util.**FakeRequest** (*data=None*, *headers=None*, *user=None*, *method='POST'*, ***kwargs*)

Bases: object

A simple request stub.

GET = {}

META = {'REMOTE_ADDR': '127.0.0.1'}

POST = {}

body = None

path = 'fake_login_path'

seed.tests.util.**make_fake_property** (*import_file*, *init_data*, *bs_type*, *is_canon=False*, *org=None*)

For making fake mapped PropertyState to test matching against.

Inheritance

Submodules

Decorators

seed.decorators.**DecoratorMixin** (*decorator*)

Converts a decorator written for a function view into a mixin for a class-based view.

Example:

```
LoginRequiredMixin = DecoratorMixin(login_required)
class MyView(LoginRequiredMixin):
    pass

class SomeView(DecoratorMixin(some_decorator), DecoratorMixin(something_else)):
    pass
```

seed.decorators.**ajax_request** (*func*)

- Copied from django-annoying, with a small modification. Now we also check for 'status' or 'success' keys and return correct status codes

If view returned serializable dict, returns response in a format requested by HTTP_ACCEPT header. Defaults to JSON if none requested or match.

Currently supports JSON or YAML (if installed), but can easily be extended.

Example:

```
@ajax_request
def my_view(request):
    news = News.objects.all()
    news_titles = [entry.title for entry in news]
    return { 'news_titles': news_titles }
```

seed.decorators.**ajax_request_class** (*func*)

•Copied from django-annoying, with a small modification. Now we also check for 'status' or

'success' keys and return correct status codes

If view returned serializable dict, returns response in a format requested by HTTP_ACCEPT header. Defaults to JSON if none requested or match.

Currently supports JSON or YAML (if installed), but can easily be extended.

Example:

```
@ajax_request
def my_view(self, request):
    news = News.objects.all()
    news_titles = [entry.title for entry in news]
    return { 'news_titles': news_titles }
```

seed.decorators.**get_prog_key**(*func_name*, *import_file_pk*)
Return the progress key for the cache

seed.decorators.**lock_and_track**(*fn*, **args*, ***kwargs*)
Decorator to lock tasks to single executor and provide progress url.

seed.decorators.**require_organization_id**(*func*)
Validate that organization_id is in the GET params and it's an int.

seed.decorators.**require_organization_id_class**(*fn*)
Validate that organization_id is in the GET params and it's an int.

seed.decorators.**require_organization_membership**(*fn*)
Validate that the organization_id passed in GET is valid for request user.

Factory

class seed.factory.**SEEDFactory**
Bases: *seed.test_helpers.factory.helpers.DjangoFunctionalFactory*

model factory for SEED

classmethod building_snapshot(*canonical_building=None*, **args*, ***kwargs*)
creates an BuildingSnapshot inst.

if canonical_building (CanonicalBuilding inst.) is None, then a CanonicalBuilding inst. is created and a BuildingSnapshot inst. is created and linked to the CanonicalBuilding inst.

Models

Search

Search methods pertaining to buildings.

seed.search.**build_json_params**(*order_by*, *sort_reverse*)
returns db_columns, extra_data_sort, and updated order_by

Parameters **order_by** (*str*) – field to order_by

Returns tuple: db_columns: dict of known DB columns i.e. non-JsonField, extra_data_sort bool if order_by is in extra_data JsonField, order_by str if sort_reverse and DB column prepend a '-' for the django order_by

seed.search.**build_shared_buildings_orgs**(*orgs*)
returns a list of sibling and parent orgs

seed.search.**convert_to_js_timestamp**(*timestamp*)
converts a django/python datetime object to milliseconds since epoch duplicated code with seed utils due to circular imports

seed.search.**create_building_queryset**(*orgs*, *exclude*, *order_by*, *other_orgs=None*, *extra_data_sort=False*)
creates a queryset of buildings within orgs. If other_orgs, buildings in both orgs and other_orgs will be represented in the queryset.

Parameters

- **orgs** – queryset of Organization inst.
- **exclude** – django query exclude dict.
- **order_by** – django query order_by str.
- **other_orgs** – list of other orgs to or the query

seed.search.**create_inventory_queryset**(*inventory_type*, *orgs*, *exclude*, *order_by*, *other_orgs=None*)
creates a queryset of properties or taxlots within orgs. If other_orgs, properties/taxlots in both orgs and other_orgs will be represented in the queryset.

Parameters

- **inventory_type** – property or taxlot.
- **orgs** – queryset of Organization inst.
- **exclude** – django query exclude dict.
- **order_by** – django query order_by str.
- **other_orgs** – list of other orgs to or the query

seed.search.**filter_other_params**(*queryset*, *other_params*, *db_columns*)
applies a dictionary filter to the query set. Does some domain specific parsing, mostly to remove extra query params and deal with ranges. Ranges should be passed in as '<field name>__lte' or '<field name>__gte' e.g. other_params = {'gross_floor_area__lte': 50000}

Parameters

- **Queryset queryset** (*Django*) – queryset to be filtered
- **other_params** (*dict*) – dictionary to be parsed and applied to filter.
- **db_columns** (*dict*) – list of column names, extra_data blob outside these

Returns Django Queryset:

seed.search.**generate_paginated_results**(*queryset*, *number_per_page=25*, *page=1*, *whitelist_orgs=None*, *below_threshold=False*, *matching=True*)
Return a page of results as a list from the queryset for the given fields

Parameters

- **queryset** – optional queryset to filter from
- **number_per_page** (*int*) – optional number of results per page
- **page** (*int*) – optional page of results to get

- **whitelist_orgs** – a queryset returning the organizations in which all building fields can be returned, otherwise only the parent organization's `exportable_fields` should be returned. The `whitelist_orgs` are the orgs the request user belongs.
- **below_threshold** – True if less than the parent org's query threshold is greater than the number of queryset results. If True, only return buildings within whitelist_orgs.
- **matching** – Toggle expanded parent and children data, including coparent and confidence

Usage:

```
generate_paginated_results(q, 1)
```

Returns:

```
[
    {
        'gross_floor_area': 1710,
        'site_eui': 123,
        'tax_lot_id': 'a-tax-lot-id',
        'year_built': 2001
    }
]
```

seed.search.**get_building_fieldnames**()
returns a list of field names for the BuildingSnapshot class/model that will be searched against

seed.search.**get_inventory_fieldnames**(*inventory_type*)
returns a list of field names that will be searched against

seed.search.**get_orgs_w_public_fields**()
returns a list of orgs that have publicly shared fields

seed.search.**inventory_search_filter_sort**(*inventory_type*, *params*, *user*)
Given a parsed set of params, perform the search, filter, and sort for Properties or Taxlots

seed.search.**is_not_whitelist_building**(*parent_org*, *building*, *whitelist_orgs*)
returns false if a building is part of the whitelist_orgs

Parameters

- **parent_org** – the umbrella parent Organization instance.
- **building** – the BuildingSnapshot inst.
- **whitelist_orgs** – queryset of Organization instances.

Returns bool

seed.search.**mask_results**(*search_results*)
masks (deletes dict keys) for non-shared public fields

seed.search.**orchestrate_search_filter_sort**(*params*, *user*, *skip_sort=False*)
Given a parsed set of params, perform the search, filter, and sort for BuildingSnapshot's

seed.search.**paginate_results**(*request*, *search_results*)
returns a paginated list of dict results

seed.search.**parse_body**(*request*)
parses the request body for search params, q, etc

Parameters **request** – django wsgi request object

Returns dict

Example:

```
{
    'exclude': dict, exclude dict for django queryset
    'order_by': str, query order_by, defaults to 'tax_lot_id'
    'sort_reverse': bool, True if ASC, False if DSC
    'page': int, pagination page
    'number_per_page': int, number per pagination page
    'show_shared_buildings': bool, whether to search across all user's orgs
    'q': str, global search param
    'other_search_params': dict, filter params
    'project_id': str, project id if exists in body
}
```

seed.search.**process_search_params**(*params*, *user*, *is_api_request=False*)

Given a python representation of a search query, process it into the internal format that is used for searching, filtering, sorting, and pagination.

Parameters

- **params** – a python object representing the search query
- **user** – the user this search is for
- **is_api_request** – bool, boolean whether this search is being done as an api request.

Returns dict

Example:

```
{
    'exclude': dict, exclude dict for django queryset
    'order_by': str, query order_by, defaults to 'tax_lot_id'
    'sort_reverse': bool, True if ASC, False if DSC
    'page': int, pagination page
    'number_per_page': int, number per pagination page
    'show_shared_buildings': bool, whether to search across all user's orgs
    'q': str, global search param
    'other_search_params': dict, filter params
    'project_id': str, project id if exists in body
}
```

seed.search.**remove_results_below_q_threshold**(*search_results*)

removes buildings if total count of buildings grouped by org is less than their org's public query threshold

Parameters **search_results** (*list/queryset*) – search results

Returns list or queryset

seed.search.**search_buildings**(*q*, *fieldnames=None*, *queryset=None*)

returns a queryset for matching buildings :param str or unicode q: search string :param list fieldnames: list of BuildingSnapshot model fieldnames

(defaults to those generated by get_building_field_names())

Parameters **queryset** – optional queryset to filter from, defaults to BuildingSnapshot.objects.none()

Returns

queryset queryset of matching buildings

seed.search.**search_inventory**(*inventory_type*, *q*, *fieldnames=None*, *queryset=None*)
returns a queryset for matching Taxlot(View)/Property(View) :param str or unicode q: search string :param list fieldnames: list of model fieldnames :param queryset: optional queryset to filter from, defaults to

BuildingSnapshot.objects.none()

Returns

queryset queryset of matching buildings

seed.search.**search_properties**(*q*, *fieldnames=None*, *queryset=None*)

seed.search.**search_public_buildings**(*request*, *orgs*)
returns a queryset or list of buildings matching the search params and count

Parameters

- **request** – wsgi request (Django) for parsing params
- **orgs** – list of Organization instances to search within

Returns tuple (search_results_list, result count)

seed.search.**search_taxlots**(*q*, *fieldnames=None*, *queryset=None*)

Tasks

Token Generator

token_generator.py - taken from django core master branch

needed a token check that would not expire after three days for sending a signup email

class seed.token_generators.**SignupTokenGenerator**
Bases: object

Strategy object used to generate and check tokens for the password reset mechanism.

check_token(*user*, *token*, *token_expires=True*)
Check that a password reset token is correct for a given user.

make_token(*user*)
Returns a token that can be used once to do a password reset for the given user.

URLs

Utils

Views

Module contents

Serializers Package

Submodules

Serializers

class seed.serializers.celery.**CeleryDatetimeSerializer**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

default(*obj*)

static seed_decoder(*obj*)

static seed_dumps(*obj*)

static seed_loads(*obj*)

Labels

class seed.serializers.labels.**LabelSerializer**(**args*, ***kwargs*)

Bases: rest_framework.serializers.ModelSerializer

class Meta

extra_kwargs = {'super_organization': {'write_only': True}}

fields = ('id', 'name', 'color', 'organization_id', 'super_organization', 'is_applied')

model

alias of StatusLabel

LabelSerializer.**get_is_applied**(*obj*)

Module contents

URLs Package

Submodules

Accounts

APIs

Main

Projects

Utilities Package

Submodules

APIs

class seed.utils.api.**APIBypassCSRFMiddleware**
Bases: object

This middleware turns off CSRF protection for API clients.

It must come before CsrfViewMiddleware in settings.MIDDLEWARE_CLASSES.

process_view(*request*, **args*, ***kwargs*)
If this request is an API request, bypass CSRF protection.

seed.utils.api.**api_endpoint**(*fn*)
Decorator function to mark a view as allowed to authenticate via API key.

Decorator must be used before login_required or has_perm to set request.user for those decorators.

seed.utils.api.**api_endpoint_class**(*fn*)
Decorator function to mark a view as allowed to authenticate via API key.

Decorator must be used before login_required or has_perm to set request.user for those decorators.

seed.utils.api.**clean_api_regex**(*url*)
Given a django-style url regex pattern, strip it down to a human-readable url.

TODO: If pks ever appear in the url, this will need to account for that.

seed.utils.api.**drf_api_endpoint**(*fn*)
Decorator to register a Django Rest Framework view with the list of API endpoints. Marks it with *is_api_endpoint = True* as well as appending it to the global *endpoints* list.

seed.utils.api.**format_api_docstring**(*docstring*)
Cleans up a python method docstring for human consumption.

seed.utils.api.**get_all_urls**(*urllist*, *prefix=''*)
Recursive generator that traverses entire tree of URLs, starting with urllist, yielding a tuple of (url_pattern, view_function) for each one.

seed.utils.api.**get_api_endpoints**()
Examines all views and returns those with is_api_endpoint set to true (done by the @api_endpoint decorator).

..:todo: this isn't particularly expensive now, but if the number of URLs grows a lot, it may be worth caching this somewhere.

seed.utils.api.**get_api_request_user**(*request*)
Determines if this is an API request and returns the corresponding user if so.

Buildings

seed.utils.buildings.**get_buildings_for_user_count**(*user*)
returns the number of buildings in a user's orgs

seed.utils.buildings.**get_columns**(*org_id*, *all_fields=False*)
Get default columns, to be overridden in future

Returns:

```
title: HTML presented title of column
sort_column: semantic name used by js and for searching DB
class: HTML CSS class for row td elements
title_class: HTML CSS class for column td elements
type: 'string', 'number', 'date'
min, max: the django filter key e.g. gross_floor_area__gte
field_type: assessor, pm, or compliance (currently not used)
sortable: determines if the column is sortable
checked: initial state of "edit columns" modal
static: True if option can be toggle (ID is false because it is
    always needed to link to the building detail page)
link: signifies that the cell's data should link to a building detail
    page
```

seed.utils.buildings.**get_search_query**(*user*, *params*)

seed.utils.buildings.**get_source_type**(*import_file*, *source_type=''*)
Used for converting ImportFile source_type into an int.

seed.utils.buildings.**serialize_building_snapshot**(*b*, *pm_cb*, *building*)
returns a dict that's safe to JSON serialize

Constants

Mappings

seed.utils.mapping.**get_mappable_columns**(*exclude_fields=None*)
Get a list of all the columns we're able to map to that are fields in the database already

seed.utils.mapping.**get_mappable_types**(*exclude_fields=None*)
Like get_mappable_columns, but with type information.

Organizations

seed.utils.organizations.**create_organization**(*user*, *org_name=''*, **args*, ***kwargs*)
Helper script to create a user/org relationship from scratch.

Parameters

- **user** – user inst.
- **org_name** – str, name of Organization we'd like to create.
- **kwargs** (*(optional)*) – 'role', int; 'status', str.

Projects

seed.utils.projects.**copy_buildings**(*source_project*, *target_project*, *buildings*, *select_all*, *search_params*, *user*)

copies buildings from source project to target project

Parameters

- **source_project_slug** – str, a slug to get a Project inst.
- **target_project_slug** – str, a slug to get a Project inst.
- **buildings** – list, list of source_facility_id as str to get BuildingSnapshot inst.
- **select_all** – bool, if the select all checkbox was checked. i.e. should we transfer all buildings in a project or just the buildings in the list

Search_params dict, params needed to generate a queryset of buildings, with keys (q, other_params, project_slug)

User User inst., django user instance needed for select all queryset

seed.utils.projects.**delete_matching_buildings**(*project*, *buildings*, *select_all*, *search_params*, *user*)

deletes buildings in a project that match search search params

Parameters

- **project_slug** – str, a slug to get a Project inst.
- **buildings** – list, list of source_facility_id as str to get BuildingSnapshot inst.
- **select_all** – bool, if the select all checkbox was checked. i.e. should we transfer all buildings in a project or just the buildings in the list

Search_params dict, params needed to generate a queryset of buildings, with keys (q, other_params, project_slug)

User User inst., django user instance needed for select all queryset

seed.utils.projects.**get_projects**(*building*, *organization*)

return an JSON friendly list of the building's projects

Parameters

- **building** – the BuildingSnapshot inst.
- **organization** – the Organization inst.

Returns list of projects

seed.utils.projects.**get_transfer_buildings**(*source_project*, *target_project*, *buildings*, *select_all*, *search_params*, *user*)

generates move or copy buildings queryset

Parameters

- **source_project_slug** – str, a slug to get a Project inst.

- **target_project_slug** – str, a slug to get a Project inst.
- **buildings** – list, list of source_facility_id as str to get BuildingSnapshot inst.
- **select_all** – bool, if the select all checkbox was checked. i.e. should we transfer all buildings in a project or just the buildings in the list

Search_params dict, params needed to generate a queryset of buildings, with keys (q, other_params, project_slug)

User User inst., django user instance needed for select all queryset

Rtype Queryset a django queryset of buildings to move or copy

seed.utils.projects.**move_buildings**(*source_project*, *target_project*, *buildings*, *select_all*, *search_params*, *user*)

moves buildings from source project to target project

Parameters

- **source_project_slug** – str, a slug to get a Project inst.
- **target_project_slug** – str, a slug to get a Project inst.
- **buildings** – list, list of source_facility_id as str to get BuildingSnapshot inst.
- **select_all** – bool, if the select all checkbox was checked. i.e. should we transfer all buildings in a project or just the buildings in the list

Search_params dict, params needed to generate a queryset of buildings, with keys (q, other_params, project_slug)

User User inst., django user instance needed for select all queryset

seed.utils.projects.**transfer_buildings**(*source_project_slug*, *target_project_slug*, *buildings*, *select_all*, *search_params*, *user*, *copy_flag=False*)

copies or moves buildings from one project to another

Parameters

- **source_project_slug** – str, a slug to get a Project inst.
- **target_project_slug** – str, a slug to get a Project inst.
- **buildings** – list, list of source_facility_id as str to get BuildingSnapshot inst.
- **select_all** – bool, if the select all checkbox was checked. i.e. should we transfer all buildings in a project or just the buildings in the list

Search_params dict, params needed to generate a queryset of buildings, with keys (q, other_params, project_slug)

User User inst., django user instance needed for select all queryset and to update the projects' "last changed" and "last changed by"

Copy_flag bool, True - copy buildings, False - move buildings

Time

seed.utils.time.**convert_datestr**(*datestr*)

Converts dates like *12/31/2010* into datetime objects.

seed.utils.time.**convert_to_js_timestamp**(*timestamp*)

converts a django/python datetime object to milliseconds since epoch

seed.utils.time.**parse_datetime**(*maybe_datetime*)
Process a datetime value that may be None, timestamp, strftime.

Views Package

Submodules

Accounts

APIs

Main

class seed.views.main.**DataFileViewSet**(***kwargs*)
Bases: rest_framework.viewsets.ViewSet

authentication_classes = (<class 'rest_framework.authentication.SessionAuthentication'>, <class 'seed.authentica

mapping_suggestions(*request*, **args*, ***kwargs*)
Returns suggested mappings from an uploaded file's headers to known data fields. — type:

status: required: true type: string description: Either success or error

suggested_column_mappings: required: true type: dictionary description: Dictionary where (key, value) = (the column header from the file,

array of tuples (destination column, score))

building_columns: required: true type: array description: A list of all possible columns

building_column_types: required: true type: array description: A list of column types corresponding to the building_columns array

parameter_strategy: replace parameters:

•name: pk description: import_file_id required: true paramType: path

•name: organization_id description: The organization_id for this user's organization required: true paramType: query

raise_exception = True

suffix = None

seed.views.main.**angular_js_tests**(*request*)
Jasmine JS unit test code covering AngularJS unit tests and ran by ./manage.py harvest

seed.views.main.**delete_buildings**(*request*, **args*, ***kwargs*)
Deletes all BuildingSnapshots the user has selected.

Does not delete selected_buildings where the user is not a member or owner of the organization the selected building belongs. Since search shows buildings across all the orgs a user belongs, it's possible for a building to belong to an org outside of *org_id*.

DELETE Expects 'org_id' for the organization, and the search payload similar to add_buildings/create_project

Payload:

```
{
    'organization_id': 2,
    'search_payload': {
        'selected_buildings': [2, 3, 4],
        'select_all_checkbox': False,
        'filter_params': ... // see search_buildings
    }
}
```

Returns:

```
{
    'status': 'success' or 'error'
}
```

seed.views.main.**delete_duplicates_from_import_file**(*request*, **args*, ***kwargs*)
Retrieves the number of matched and unmatched BuildingSnapshots for a given ImportFile record.

GET Expects import_file_id corresponding to the ImportFile in question.

Returns:

```
{
    "status": "success",
    "deleted": "Number of duplicates deleted"
}
```

seed.views.main.**delete_file**(*request*, **args*, ***kwargs*)
Deletes an ImportFile from a dataset.

Payload:

```
{
    "file_id": "ImportFile id",
    "organization_id": "current user organization id as integer"
}
```

Returns:

```
{
    'status': 'success' or 'error',
    'message': 'error message, if any'
}
```

seed.views.main.**delete_organization_buildings**(*request*, **args*, ***kwargs*)
Starts a background task to delete all BuildingSnapshots in an org.

GET Expects 'org_id' for the organization.

Returns:

```
{
    'status': 'success' or 'error',
    'progress_key': ID of background job, for retrieving job progress
}
```

seed.views.main.**delete_organization_inventory**(*request*, **args*, ***kwargs*)
Starts a background task to delete all properties & taxlots in an org.

GET Expects 'org_id' for the organization.

Returns:

```
{
    'status': 'success' or 'error',
    'progress_key': ID of background job, for retrieving job progress
}
```

seed.views.main.**export_buildings**(*request*, **args*, ***kwargs*)

Begins a building export process.

Payload:

```
{
    "export_name": "My Export",
    "export_type": "csv",
    "selected_buildings": [1234,], (optional list of building ids)
    "selected_fields": optional list of fields to export
    "select_all_checkbox": True // optional, defaults to False
}
```

Returns:

```
{
    "success": True,
    "status": "success",
    "export_id": export_id; see export_buildings_download,
    "total_buildings": count of buildings,
}
```

seed.views.main.**export_buildings_download**(*request*, **args*, ***kwargs*)

Provides the url to a building export file.

Payload:

```
{
    "export_id": export_id from export_buildings
}
```

Returns:

```
{
    'success': True or False,
    'status': 'success or error',
    'message': 'error message, if any',
    'url': The url to the exported file.
}
```

seed.views.main.**export_buildings_progress**(*request*, **args*, ***kwargs*)

Returns current progress on building export process.

Payload:

```
{
    "export_id": export_id from export_buildings
}
```

Returns:

```
{
    'success': True,
    'status': 'success or error',
    'message': 'error message, if any',
    'buildings_processed': number of buildings exported
}
```

seed.views.main.**get_building**(*request*)

Retrieves a building. If user doesn't belong to the building's org, fields will be masked to only those shared within the parent org's structure.

GET Expects building_id and organization_id in query string. building_id should be the *canonical_building* ID for the building, not the BuildingSnapshot id.

Returns:

```
{
     'status': 'success or error',
     'message': 'error message, if any',
     'building': {'id': the building's id,
                  'canonical_building': the canonical building ID,
                  other fields this user has access to...
     },
     'imported_buildings': [ A list of buildings imported to create
                             this building's record, in the same
                             format as 'building'
                           ],
     'projects': [
        // A list of the building's projects
        {
            "building": {
                "approved_date":07/30/2014,
                "compliant": null,
                "approver": "demo@seed.lbl.gov"
                "approved_date": "07/30/2014"
                "compliant": null
                "label": {
                    "color": "red",
                    "name": "non compliant",
                    id: 1
                }
            }
            "description": null
            "id": 3
            "is_compliance": false
            "last_modified_by_id": 1
            "name": "project 1"
            "owner_id": 1
            "slug": "project-1"
            "status": 1
            "super_organization_id": 1
        },
        . . .
    ],
     'user_role': role of user in this org,
     'user_org_id': the org id this user belongs to
}
```

seed.views.main.**get_columns**(*request*, **args*, ***kwargs*)
returns a JSON list of columns a user can select as his/her default

GET Expects organization_id in the query string.

seed.views.main.**get_coparents**(*request*, **args*, ***kwargs*)
Returns the nodes in the BuildingSnapshot tree that can be unmatched.

GET Expects organization_id and building_id in the query string

Returns:

```
{
    'status': 'success',
    'coparents': [
        {
            "id": 333,
            "coparent": 223,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1123
        },
        {
            "id": 223,
            "coparent": 333,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1124
        },
        ...
    ]
}
```

seed.views.main.**get_default_building_detail_columns**(*request*, **args*, ***kwargs*)
Get default columns for building detail view.

front end is expecting a JSON object with an array of field names

Returns:

```
{
    "columns": ["project_id", "name", "gross_floor_area"]
}
```

seed.views.main.**get_default_columns**(*request*, **args*, ***kwargs*)
Get default columns for building list view.

front end is expecting a JSON object with an array of field names

Returns:

```
{
    "columns": ["project_id", "name", "gross_floor_area"]
}
```

seed.views.main.**get_total_number_of_buildings_for_user**(*request*, **args*, ***kwargs*)
gets a count of all buildings in the user's organizations

seed.views.main.**home**(*request*, **args*, ***kwargs*)
the main view for the app Sets in the context for the django template:

•**app_urls**: a json object of all the URLs that is loaded in the JS global namespace

- **username**: the request user's username (first and last name)
- **AWS_UPLOAD_BUCKET_NAME**: S3 direct upload bucket
- **AWS_CLIENT_ACCESS_KEY**: S3 direct upload client key
- **FILE_UPLOAD_DESTINATION**: 'S3' or 'filesystem'

seed.views.main.**public_search**(*request*, **args*, ***kwargs*)

the public API unauthenticated endpoint

see `search_buildings` for the non-public version

seed.views.main.**save_match**(*request*)

seed.views.main.**search_buildings**(*request*, **args*, ***kwargs*)

Retrieves a paginated list of CanonicalBuildings matching search params.

Payload:

```
{
    'q': a string to search on (optional),
    'show_shared_buildings': True to include buildings from other orgs in this␣
↪user's org tree,
    'order_by': which field to order by (e.g. pm_property_id),
    'import_file_id': ID of an import to limit search to,
    'filter_params': {
        a hash of Django-like filter parameters to limit query.  See seed.search.
↪filter_other_params.
        If 'project__slug' is included and set to a project's slug, buildings␣
↪will include associated labels
        for that project.
    }
    'page': Which page of results to retrieve (default: 1),
    'number_per_page': Number of buildings to retrieve per page (default: 10),
}
```

Returns:

```
{
    'status': 'success',
    'buildings': [
        {
            all fields for buildings the request user has access to, e.g.:
                'canonical_building': the CanonicalBuilding ID of the building,
                'pm_property_id': ID of building (from Portfolio Manager),
                'address_line_1': First line of building's address,
                'property_name': Building's name, if any
            ...
        }...
    ]
    'number_matching_search': Total number of buildings matching search,
    'number_returned': Number of buildings returned for this page
}
```

seed.views.main.**set_default_building_detail_columns**(*request*, **args*, ***kwargs*)

seed.views.main.**set_default_columns**(*request*, **args*, ***kwargs*)

Meters

seed.views.meters.**add_meter_to_building**(*request*, **args*, ***kwargs*)
Will add a building to an existing meter.

Payload:

```
{
    'organization_id': 435,
    'building_id': 342,
    'meter_name': 'Unit 34.',
    'energy_type': 'Electricity',
    'energy_units': 'kWh'
}
```

seed.views.meters.**add_timeseries**(*request*, **args*, ***kwargs*)
Add time series data for a meter.

Payload:

```
{
    'organization_id': 435,
    'meter_id': 34,
    'timeseries': [
        {
            'begin_time': 2342342232,
            'end_time': 23423433433,
            'cost': 232.23,
        }...
    ]
}
```

seed.views.meters.**get_meters**(*request*, **args*, ***kwargs*)
Returns all of the meters for a building.

Expected GET params:

building_id: int, unique identifier for a (canonical) building.

seed.views.meters.**get_timeseries**(*request*, **args*, ***kwargs*)
Return all time series data for a building, grouped by meter.

Expected GET params:

meter_id: int, unique identifier for the meter. offset: int, the offset from the most recent meter data to begin showing. num: int, the number of results to show.

Projects

class seed.views.projects.**ProjectViewSet**(***kwargs*)
Bases: seed.decorators.DecoratorMixindrf_api_endpoint, rest_framework.viewsets.ModelViewSet

ProjectViewModels = {'property': <class 'seed.models.projects.ProjectPropertyView'>, 'taxlot': <class 'seed.models.

ViewModels = {'property': <class 'seed.models.properties.PropertyView'>, 'taxlot': <class 'seed.models.tax_lots.TaxLot

add(*request*, **args*, ***kwargs*)
Add inventory to project :PUT: Expects organization_id in query string. — parameters:

•name: organization_id description: ID of organization to associate new project with type: integer required: true

•name: inventory_type description: type of inventory to add: 'property' or 'taxlot' type: string required: true paramType: query

•name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path

•name: selected description: ids of property or taxlot views to add type: array[int] required: true

Returns:

{ 'status': 'success', 'added': [list of property/taxlot view ids added]

}

authentication_classes = (<class 'rest_framework.authentication.SessionAuthentication'>, <class 'seed.authentica

count (*request*, **args*, ***kwargs*)

Returns the number of projects within the org tree to which a user belongs. Counts projects in parent orgs and sibling orgs.

GET Expects organization_id in query string.

— parameters:

•name: organization_id description: The organization_id for this user's organization required: true paramType: query

type:

status: type: string description: success, or error

count: type: integer description: number of projects

create (*request*, **args*, ***kwargs*)

Creates a new project

POST Expects organization_id in query string.

— parameters:

•name: organization_id description: ID of organization to associate new project with type: integer required: true paramType: query

•name: name description: name of the new project type: string required: true

•name: is_compliance description: add compliance data if true type: bool required: true

•name: compliance_type description: description of type of compliance type: string required: true if is_compliance else false

•name: description description: description of new project type: string required: true if is_compliance else false

•name: end_date description: Timestamp for when project ends type: string required: true if is_compliance else false

•name: deadline_date description: Timestamp for compliance deadline type: string required: true if is_compliance else false

Returns::

{ 'status': 'success', 'project': {

'id': project's primary key, 'name': project's name, 'slug': project's identifier, 'status': 'active', 'number_of_buildings': Count of buildings associated with project 'last_modified': Timestamp when project last changed 'last_modified_by': {

'first_name': first name of user that made last change, 'last_name': last name, 'email': email address,

}, 'is_compliance': True if project is a compliance project, 'compliance_type': Description of compliance type, 'deadline_date': Timestamp of when compliance is due, 'end_date': Timestamp of end of project, 'property_count': 0, 'taxlot_count': 0,

}

}

destroy(*request*, **args*, ***kwargs*)

Delete a project.

DELETE Expects organization_id in query string.

— parameter_strategy: replace parameters:

- name: organization_id description: The organization_id for this user's organization required: true paramType: query
- name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path

Returns::

{ 'status': 'success',

}

get_error(*error*, *key=None*, *val=None*)

Return error message and corresponding http status code.

get_key(*pk*)

Determine where to use slug or pk to identify project.

get_organization()

Get org id from query param or request.user.

get_params(*keys*)

Get required params from post etc body.

Returns dict of params and list of missing params.

get_project(*key*, *pk*)

Get project for view.

get_queryset()

get_status(*status*)

Get status from string or int

list(*request*, **args*, ***kwargs*)

Retrieves all projects for a given organization.

GET Expects organization_id in query string.

parameters:

- name: organization_id description: The organization_id for this user's organization required: true paramType: query

Returns:

```
{
    'status': 'success',
    'projects': [
        {
            'id': project's primary key,
            'name': project's name,
            'slug': project's identifier,
            'status': 'active',
            'number_of_buildings': Count of buildings associated with project
            'last_modified': Timestamp when project last changed
            'last_modified_by': {
                'first_name': first name of user that made last change,
                'last_name': last name,
                'email': email address,
            },
            'is_compliance': True if project is a compliance project,
            'compliance_type': Description of compliance type,
            'deadline_date': Timestamp of when compliance is due,
            'end_date': Timestamp of end of project,
            'property_count': number of property views associated with␣
↪project,
            'taxlot_count':  number of taxlot views associated with project,
        }...
    ]
}
```

parser_classes = (<class 'rest_framework.parsers.JSONParser'>,)

partial_update(*request*, **args*, ***kwargs*)

Updates a project. Allows partial update, i.e. only updated param s need be supplied.

PUT Expects organization_id in query string.

— parameters:

•name: organization_id description: ID of organization to associate new project with type: integer required: true paramType: query

•name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path

•name: name description: name of the new project type: string required: false

•name: is_compliance description: add compliance data if true type: bool required: false

•name: compliance_type description: description of type of compliance type: string required: true if is_compliance else false

•name: description description: description of new project type: string required: true if is_compliance else false

•name: end_date description: Timestamp for when project ends type: string required: true if is_compliance else false

•name: deadline_date description: Timestamp for compliance deadline type: string required: true if is_compliance else false

Returns::

{ 'status': 'success', 'project': {

'id': project's primary key, 'name': project's name, 'slug': project's identifier, 'status': 'active', 'number_of_buildings': Count of buildings associated with project 'last_modified': Timestamp when project last changed 'last_modified_by': {

'first_name': first name of user that made last change, 'last_name': last name, 'email': email address,

}, 'is_compliance': True if project is a compliance project, 'compliance_type': Description of compliance type, 'deadline_date': Timestamp of when compliance is due, 'end_date': Timestamp of end of project, 'property_count': number of property views associated with project, 'taxlot_count': number of taxlot views associated with project,

}

}

project_view_factory(*inventory_type*, *project_id*, *view_id*)
ProjectPropertyView/ProjectTaxLotView factory.

query_set = []

remove(*request*, **args*, ***kwargs*)
Remove inventory from project :PUT: Expects organization_id in query string. — parameters:

- name: organization_id description: ID of organization to associate new project with type: integer required: true
- name: inventory_type description: type of inventory to add: 'property' or 'taxlot' type: string required: true paramType: query
- name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path
- name: selected description: ids of property or taxlot views to add type: array[int] required: true

Returns:

{ 'status': 'success', 'removed': [list of property/taxlot view ids removed]

}

renderer_classes = (<class 'rest_framework.renderers.JSONRenderer'>,)

retrieve(*request*, **args*, ***kwargs*)
Retrieves details about a project.

GET Expects organization_id in query string.

— parameter_strategy: replace parameters:

- name: organization_id description: The organization_id for this user's organization required: true paramType: query
- name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path

Returns:

```
{
 'id': project's primary key,
 'name': project's name,
 'slug': project's identifier,
 'status': 'active',
 'number_of_buildings': Count of buildings associated with project
 'last_modified': Timestamp when project last changed
 'last_modified_by': {
    'first_name': first name of user that made last change,
    'last_name': last name,
    'email': email address,
    },
 'is_compliance': True if project is a compliance project,
 'compliance_type': Description of compliance type,
 'deadline_date': Timestamp of when compliance is due,
 'end_date': Timestamp of end of project
 'property_count': number of property views associated with project,
 'taxlot_count':  number of taxlot views associated with project,
 'property_views': [list of serialized property views associated with the␣
↪project...],
 'taxlot_views': [list of serialized taxlot views associated with the␣
↪project...],
}
```

serializer_class
alias of `ProjectSerializer`

suffix = None

transfer (*request*, **args*, ***kwargs*)
Move or copy inventory from one project to another

PUT Expects organization_id in query string.

— parameter_strategy: replace parameters:

•name: organization_id description: The organization_id for this user's organization required: true type: integer paramType: query

•name: inventory_type description: type of inventory to add: 'property' or 'taxlot' required: true type: string paramType: query

•name: copy or move description: Whether to move or copy inventory required: true paramType: path required: true

-name: target type: string or int description: target project slug/id to move/copy to. required: true

•name: selected description: JSON array, list of property/taxlot views to be transferred paramType: array[int] required: true

update (*request*, **args*, ***kwargs*)
Updates a project

PUT Expects organization_id in query string.

— parameters:

•name: organization_id description: ID of organization to associate new project with type: integer required: true paramType: query

•name: project slug or pk description: The project slug identifier or primary key for this project required: true paramType: path

•name: name description: name of the new project type: string required: true

•name: is_compliance description: add compliance data if true type: bool required: true

•name: compliance_type description: description of type of compliance type: string required: true if is_compliance else false

•name: description description: description of new project type: string required: true if is_compliance else false

•name: end_date description: Timestamp for when project ends type: string required: true if is_compliance else false

•name: deadline_date description: Timestamp for compliance deadline type: string required: true if is_compliance else false

Returns::

{ 'status': 'success', 'project': {

'id': project's primary key, 'name': project's name, 'slug': project's identifier, 'status': 'active', 'number_of_buildings': Count of buildings associated with project 'last_modified': Timestamp when project last changed 'last_modified_by': {

'first_name': first name of user that made last change, 'last_name': last name, 'email': email address,

}, 'is_compliance': True if project is a compliance project, 'compliance_type': Description of compliance type, 'deadline_date': Timestamp of when compliance is due, 'end_date': Timestamp of end of project, 'property_count': number of property views associated with project, 'taxlot_count': number of taxlot views associated with project,

}

}

update_details(*request*, **args*, ***kwargs*)

Updates extra information about the inventory/project relationship. In particular, whether the property/taxlot is compliant and who approved it.

PUT Expects organization_id in query string.

— parameter_strategy: replace parameters:

•name: organization_id description: The organization_id for this user's organization required: true type: integer paramType: query

•name: inventory_type description: type of inventory to add: 'property' or 'taxlot' required: true type: string paramType: query

•name: id description: id of property/taxlot view to update required: true type: integer paramType: string

•name: compliant description: is compliant required: true type: bool paramType: string

Returns::

{ 'status': 'success', 'approved_date': Timestamp of change (now), 'approver': Email address of user making change

}

seed.views.projects.**convert_dates**(*data*, *keys*)

seed.views.projects.**update_model**(*model*, *data*)

Module contents

CHAPTER 7

Developer Resources

General Notes

Flake Settings

Flake is used to statically verify code syntax. If the developer is running flake from the command line, they should ignore the following checks in order to emulate the same checks as the CI machine.

Code	Description
E402	module level import not at top of file
E501	line too long (82 characters) or max-line = 100
E731	do not assign a lambda expression, use a def
W503	line break occurred before a binary operator

To run flake locally call:

```
tox -e flake8
```

Django Notes

Both Django and AngurlarJS are used for url routing. Django routes are in *seed/urls/main.py*

AWS S3

Amazon AWS S3 Expires headers should be set on the AngularJS partials if using S3 with the management command: set_s3_expires_headers_for_angularjs_partials

Example:

```
python manage.py set_s3_expires_headers_for_angularjs_partials --verbosity=3
```

The default user invite reply-to email can be overridden in the config/settings/common.py file. The *SERVER_EMAIL* settings var is the reply-to email sent along with new account emails.

```
# config/settings/common.py
PASSWORD_RESET_EMAIL = 'reset@seed.lbl.gov'
SERVER_EMAIL = 'no-reply@seed.lbl.gov'
```

AngularJS Integration Notes

Template Tags

Angular and Django both use *{{* and *}}* as variable delimiters, and thus the AngularJS variable delimiters are renamed *{$* and *$}*.

```
window.BE.apps.seed = angular.module('BE.seed', ['ngRoute', "ngCookies"], function (
↪$interpolateProvider) {
        $interpolateProvider.startSymbol("{$");
        $interpolateProvider.endSymbol("$}");
    }
);
```

Django CSRF Token and AJAX Requests

For ease of making angular *$http* requests, we automatically add the CSRF token to all *$http* requests as recommended by http://django-angular.readthedocs.io/en/latest/integration.html#xmlhttprequest

```
window.BE.apps.seed.run(function ($http, $cookies) {
    $http.defaults.headers.common['X-CSRFToken'] = $cookies['csrftoken'];
});
```

Routes and Partials or Views

Routes in *static/seed/js/seed.js* (the normal angularjs *app.js*)

```
window.BE.apps.seed.config(['$routeProvider', function ($routeProvider) {
        $routeProvider
            .when('/', {
                templateUrl: static_url + '/seed/partials/home.html'
            })
            .when('/projects', {
                controller: 'project_list_controller',
                templateUrl: static_url + '/seed/partials/projects.html'
            })
            .when('/buildings', {
                templateUrl: static_url + '/seed/partials/buildings.html'
            })
            .when('/admin', {
                controller: 'seed_admin_controller',
                templateUrl: static_url + '/seed/partials/admin.html'
            })
            .otherwise({ redirectTo: '/' });
    }]);
```

HTML partials in *static/seed/partials/*

on production and staging servers on AWS, or for the partial html templates loaded on S3, or a CDN, the external resource should be added to the white list in *static/seed/js/seed/js*

```
// white list for s3
window.BE.apps.seed.config(function( $sceDelegateProvider ) {
$sceDelegateProvider.resourceUrlWhitelist([
```

```
    // localhost
    'self',
    // AWS s3
    'https://be-*.amazonaws.com/**'
    ]);
});
```

Logging

Information about error logging can be found here - https://docs.djangoproject.com/en/1.7/topics/logging/

Below is a standard set of error messages from Django.

A logger is configured to have a log level. This log level describes the severity of the messages that the logger will handle. Python defines the following log levels:

```
DEBUG: Low level system information for debugging purposes
INFO: General system information
WARNING: Information describing a minor problem that has occurred.
ERROR: Information describing a major problem that has occurred.
CRITICAL: Information describing a critical problem that has occurred.
```

Each message that is written to the logger is a Log Record. The log record is stored in the web server & Celery

BEDES Compliance and Managing Columns

Columns that do not represent hardcoded fields in the application are represented using a Django database model defined in the seed.models module.

The goal of adding new columns to the database is to create seed.models.Column records in the database for each column to import. In the Django application, Columns are linked together via a Schema model. Currently there is a schema containing the ESPM fields with a name of "BEDES". When creating columns they should be added to either an existing schema or a new one should be created if that is the desired result.

When creating Column records, if the type of the column is a string (should be treated as a string for searching and filtering), then just creating a Column record is sufficient for importing that column. If the column type is not string, then a seed.models.Unit record must be created and linked to the Column model via a foreign key.

The initial fields from the ESPM schema were imported using a standard Django migration using the migration library south. You can find the initial migration here:

Note: This seems out of state

https://github.com/seed-platform/seed/blob/master/seed/migrations/0025_add_espm_column_names.py

---

In that example, the migration is using the data from the 'flat_schema' and 'types' keys of the python dictionary defined here:

Note: Need to reevaluate how this is stored https://github.com/seed-platform/seed-mcm/blob/master/mcm/data/ESPM/espm.py

---

Note: The fields to import do not need to be in a separate file, and the format could differ from what is shown here as long as the migration logic accounted for the different format.

Resetting the Database

This is a brief description of how to drop and re-create the database for the seed application.

The first two commands below are commands distributed with the Postgres database, and are not part of the seed application. The third command below will create the required database tables for seed and setup initial data that the application expects (initial columns for BEDES). The last command below (spanning multiple lines) will create a new superuser and organization that you can use to login to the application, and from there create any other users or organizations that you require.

Below are the commands for resetting the database and creating a new user:

```
psql -c 'DROP DATABASE "seeddb"'
psql -c 'CREATE DATABASE "seeddb" WITH OWNER = "seeduser";'
psql -c 'GRANT ALL PRIVILEGES ON DATABASE "seeddb" TO seeduser;'
psql -c 'ALTER USER seeduser CREATEDB;'

psql -c 'ALTER USER seeduser CREATEROLE;'
./manage.py migrate
./manage.py create_default_user \
    --username=testuser@seed.org \
    --password=password \
    --organization=testorg
```

Testing

JS tests can be run with Jasmine at the url *app/angular_js_tests/*.

Python unit tests are run with

```
python manage.py test --settings=config.settings.test
```

Run coverage using

```
coverage run manage.py test --settings=config.settings.test
coverage report --fail-under=83
```

Python compliance uses PEP8 with flake8

```
flake8
# or
tox -e flake8
```

JS Compliance uses jshint

```
jshint seed/static/seed/js
```

CHAPTER 8

License

CHAPTER 9

Help

For SEED-Platform Users

Please visit our User Support website for tutorials and documentation to help you learn how to use SEED-Platform.

https://sites.google.com/a/lbl.gov/seed/

There is also a link to the SEED-Platform Users forum, where you can connect with other users.

https://groups.google.com/forum/#!forum/seed-platform-users

For direct help on a specific problem, please email: SEED-Support@lists.lbl.gov

For SEED-Platform Developers

The Open Source code is available on the Github organization SEED-Platform:

https://github.com/SEED-platform

Please join the SEED-Platform Dev forum where you can connect with other developers.

https://groups.google.com/forum/#!forum/seed-platform-dev

CHAPTER 10

Updating this documentation

This python code documentation was generated by running the following:

```
$ pip install -r requirements/local.txt
$ sphinx-apidoc -o docs/source/modules . seed/lib/mcm seed/lib/superperms
$ cd docs
$ make html
```

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

C

S

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W