SEED Platform Documentation

Release 2.2.0

Sep 29, 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

1.1 Development Setup

1.1.1 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 *CELERY_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
    }
}
CELERY_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 does not 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 not start Celery or Redis if they already seem to be running.

1.1.2 Installation using Docker

Docker works natively on Linux, Mac OSX, and Windows 10. If you are using an older version of Windows (and some older versions of Mac OSX), you will need to install Docker Toolbox.

Choose either *Docker Toolbox*, *Docker Native (Windows/OSX)*, or *Docker Native (Ubuntu)* to install Docker.

Docker Toolbox

Install Docker-Toolbox, which installs several applications including 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
# export environment variables
eval $(docker-machine env dev)
```

- Get the Docker IP address (*docker-machine ip dev*)

Docker Native (Ubuntu)

Follow instructions [here](https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/).

- [Install Docker Compose](https://docs.docker.com/compose/install/)

Docker Native (Windows/OSX)

Following instructions (for Mac)[https://docs.docker.com/docker-for-mac/install/] or (for Windows)[https://docs.docker.com/docker-for-windows/install/].

- [Install Docker Compose](https://docs.docker.com/compose/install/)

Building and Configuring Containers

- 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

- Login to container

  The docker-compose file creates a default user and password. Below are the defaults but can be overridden by setting environment variables.

```
username: user@seed-platform.org
password: super-secret-password
```

Note: Don't forget that you need to reset your default username and password if you are going to use these Docker images in production mode!

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.

2.1 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.

2.1.1 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

2.1.2 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
```

2.1.3 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
```

2.1.4 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.

2.1.5 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 `CELERY_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
    }
}
CELERY_BROKER_URL = 'redis://seed-core-cache.ntmprk.0001.usw2.cache.amazonaws.
↪com:6379/1'
```

2.1.6 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
```

2.1.7 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
```

2.1.8 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
```

2.2 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.

2.2.1 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

2.2.2 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

2.2.3 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
```

2.2.4 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
```

2.2.5 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
```

2.2.6 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 `CELERY_BROKER_URL` settings.

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

2.2.7 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.

2.2.8 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
```

2.2.9 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
```

2.2.10 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.

2.2.11 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
```

2.2.12 SMTP service

In the AWS setup, we can use SES to provide an email service for Django. The service is configured in the 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'
```

2.2.13 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
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
    }
}
CELERY_BROKER_URL = 'redis://127.0.0.1:6379/1'

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

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

2.3 Monitoring

2.3.1 Sentry

Sentry is used for development. The front end tests are run on Sentry for every commit through travis.

CHAPTER 3

API

3.1 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 ':'. The string must be base 64 encoded per the Basic Auth requirement.

Using Python, use the requests and base 64 library:

```
import requests
import base64

auth_string = base64.urlsafe_b64encode('{}:{}'.format(user_email, api_key)
auth_string = 'Basic {}'.format(auth_string)
header = {
    'Authorization': auth_string,
}

>>> header
>>> {'Authorization': 'Basic␣
↪dXNlckBzZWVkLXBsYXRmb3JtLm9yZzpiNThmMTJjMzU4NjA2MTYzYzdmZjFlNTUxMjJjNzUxN2ZkMzJhZjRi
↪'}

result = requests.get('https://seed-platform.org/api/v2/version/', headers=header)
print result.json()
```

Using curl, pass the header information in the request (use base64 result from above):

```
curl -H Authorization:"Basic␣
↪bmljaG9sYXMubG9uZ0BucmVsLmdvdjpiNThmMTJjMzU4NjA2MTYzYzdmZjFlNTUxMjJjNzUxN2ZkMzJhZjRi
↪" http://seed-platform.org/api/v2/version/
```

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

3.2 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 <auth-header> https://seed-platform.org/api/v2/organizations/12/
```

Or in a JSON payload:

```
curl -H <auth-header> \
  -d '{"organization_id":6, "role": "viewer"}' \
  https://seed-platform.org/api/v2/users/12/update_role/
```

Using Python:

```
params = {'organization_id': 6, 'role': 'viewer'}
result = requests.post('https://seed-platform.org/api/v2/users/12/update_role/',
                       data=json.dumps(params),
                       headers=header)
print result.json()
```

3.3 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):

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

3.4 API Endpoints

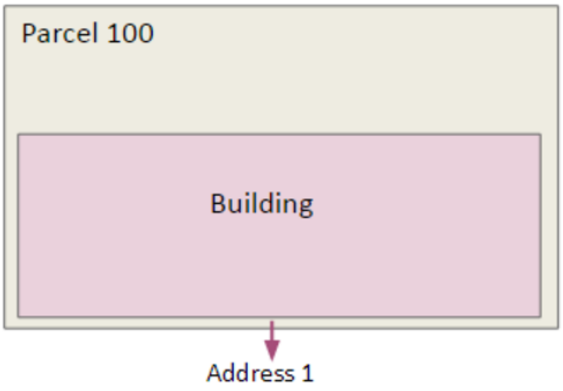
A list of interactive endpoints are available by accessing the API menu item on the left navigation pane within you account on your SEED instance.

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

CHAPTER 4

Data Model

Case A: 1 Building to 1 Parcel

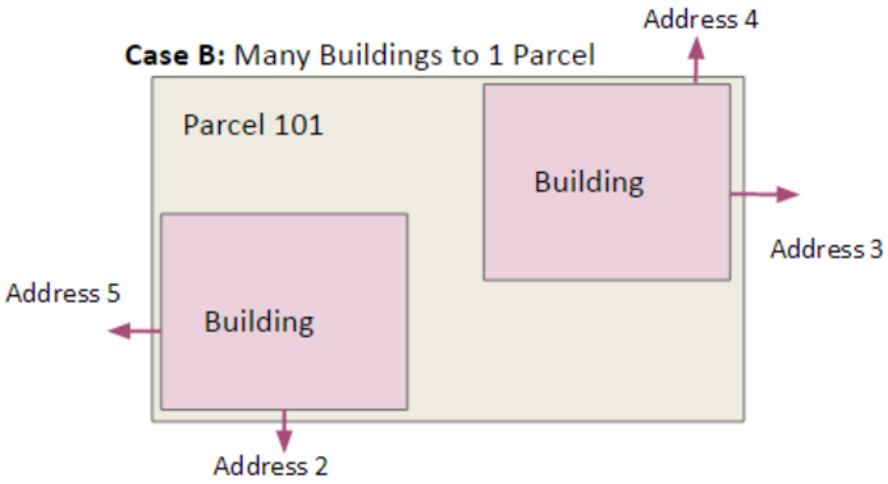


Source Data

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

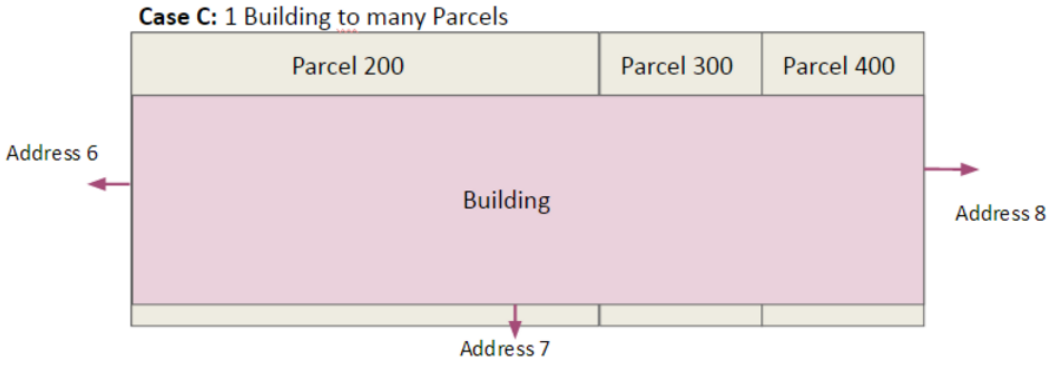


Source Data

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

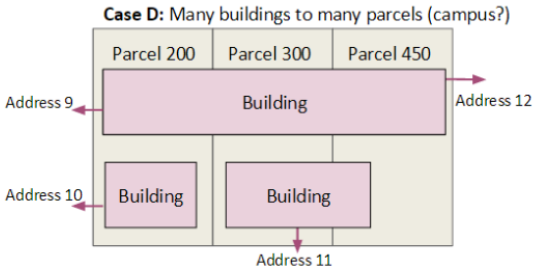


Source Data

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

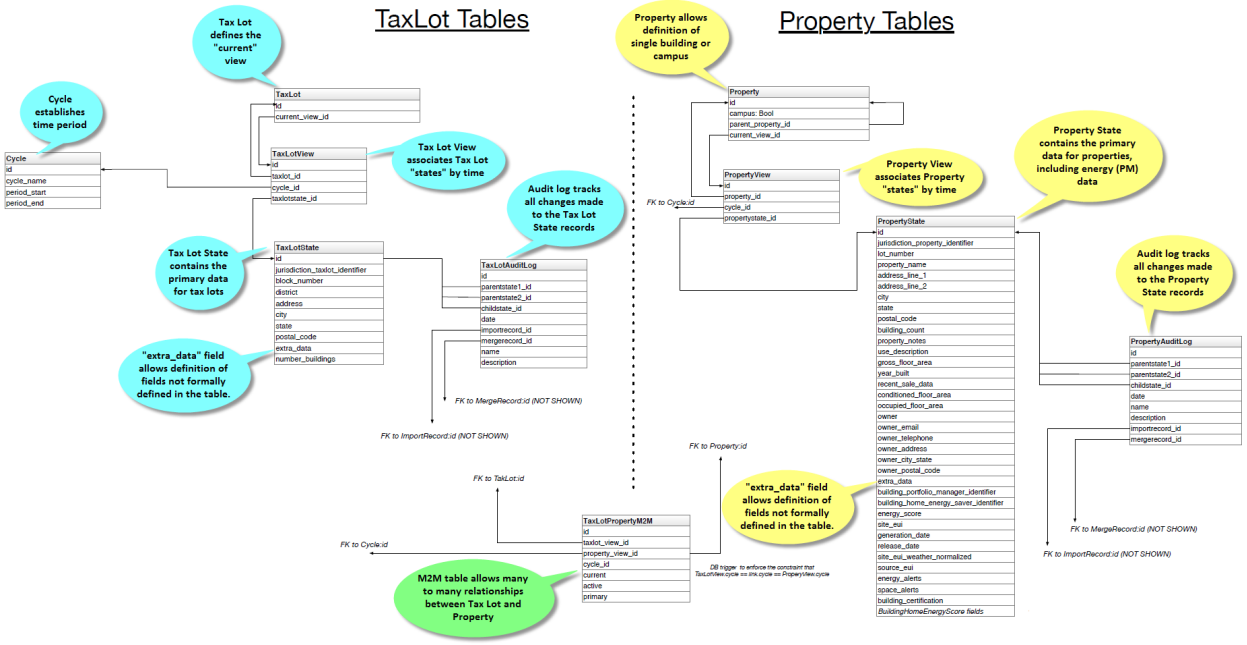


Source Data

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
```

4.1 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
```

4.1.1 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

4.2 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.

4.3 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

4.4 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.

4.5 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.

4.6 *_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.

4.7 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

4.8 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

5.1 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*.

5.2 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.

5.3 Matching

Todo: document

5.4 Pairing

Todo: document

CHAPTER 6

Modules

6.1 Audit Logs Package

6.1.1 Submodules

6.1.2 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 MultipleObjectsReturned

Bases: django.core.exceptions.MultipleObjectsReturned

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.

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.

get_audit_type_display(**moreargs*, ***morekwargs*)

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

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

to_dict()
serializes an audit_log

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_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

6.1.3 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

6.1.4 URLs

6.1.5 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'
```

6.2 Configuration

6.2.1 Submodules

6.2.2 Storage

6.2.3 Template Context

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

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

6.2.4 Tests

6.2.5 Utils

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

6.2.6 Views

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

6.2.7 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.

6.3 Data Package

6.3.1 Submodules

6.3.2 BEDES

6.3.3 Module contents

6.4 Data Importer Package

6.4.1 Submodules

6.4.2 Managers

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

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

use_for_related_fields = True

6.4.3 Models

6.4.4 URLs

6.4.5 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**(*iterlist*, *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.

6.4.6 Views

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

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

available_matches(*request*, **args*, ***kwargs*)

data_quality_progress(*request*, **args*, ***kwargs*)
Return the progress of the data quality check. — type:

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

progress: type: integer description: status of background data quality 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_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

get_data_quality_results(*request*, **args*, ***kwargs*)
Retrieve the details of the data quality check. — 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 data quality check

parameter_strategy: replace parameters:

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

static has_coparent (*state_id*, *inventory_type*, *fields=None*)
Return the coparent of the current state id based on the inventory type. If fields are given (as a list), then it will only return the fields specified of the state model object as a dictionary.

Parameters

- **state_id** – int, ID of PropertyState or TaxLotState
- **inventory_type** – string, either properties | taxlots
- **fields** – list, either None or list of fields to return

Returns dict or state object, If fields is not None then will return state_object

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

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

match (*request*, **args*, ***kwargs*)

matching_results (*request*, **args*, ***kwargs*)
Retrieves the number of matched and unmatched properties & tax lots for a given ImportFile record. Specifically for new imports

GET Expects import_file_id corresponding to the ImportFile in question.

Returns:

```
{
    'status': 'success',
    'properties': {
        'matched': Number of PropertyStates that have been matched,
        'unmatched': Number of PropertyStates that are unmatched new imports
    },
    'tax_lots': {
        'matched': Number of TaxLotStates that have been matched,
        'unmatched': Number of TaxLotStates that are unmatched new imports
```

```
    }
}
```

matching_status(*request*, **args*, ***kwargs*)
Retrieves the number and ids of matched and unmatched properties & tax lots for a given ImportFile record. Specifically for hand-matching

GET Expects import_file_id corresponding to the ImportFile in question.

Returns:

```
{
    'status': 'success',
    'properties': {
        'matched': Number of PropertyStates that have been matched,
        'matched_ids': Array of matched PropertyState ids,
        'unmatched': Number of PropertyStates that are unmatched records,
        'unmatched_ids': Array of unmatched PropertyState ids
    },
    'tax_lots': {
        'matched': Number of TaxLotStates that have been matched,
        'matched_ids': Array of matched TaxLotState ids,
        'unmatched': Number of TaxLotStates that are unmatched records,
        'unmatched_ids': Array of unmatched TaxLotState ids
    }
}
```

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
            'from_units': 'kBtu/ft**2/year', # pint-parseable units, optional
            'to_field': 'energy_use_intensity',
            'to_table_name': 'PropertyState',
        },
        {
            'from_field': 'gfa',
            'from_units': 'ft**2', # pint-parseable units, optional
            '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

unmatch(*request*, **args*, ***kwargs*)

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

seed.data_importer.views.**get_upload_details**(*request*, **args*, ***kwargs*)
Retrieves details about how to upload files to this instance.

Returns:

```
If S3 mode:
```

```
{
    'upload_mode': 'S3',
    'upload_complete': A url to notify that upload is complete,
    'signature': The url to post file details to for auth to upload to S3.
}

If local file system mode:

{
    'upload_mode': 'filesystem',
    'upload_path': The url to POST files to (see local_uploader)
}
```

seed.data_importer.views.**handle_s3_upload_complete**(*request*, **args*, ***kwargs*)

Notify the system that an upload to S3 has been completed. This is a necessary step after uploading to S3 or the SEED instance will not be aware the file exists.

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

GET Expects the following in the query string:

key: The full path to the file, within the S3 bucket. E.g. data_importer/buildings.csv

source_type: The source of the file. E.g. 'Assessed Raw' or 'Portfolio Raw'

source_program: Optional value from common.mapper.Programs source_version: e.g. "4.1"

import_record: The ID of the ImportRecord this file belongs to.

Returns:

```
{
    'success': True,
    'import_file_id': The ID of the newly-created ImportFile object.
}
```

seed.data_importer.views.**sign_policy_document**(*request*, **args*, ***kwargs*)

Sign and return the policy document for a simple upload. http://aws.amazon.com/articles/1434/#signyours3postform

Payload:

```
{
 "expiration": ISO-encoded timestamp for when signature should expire,
               e.g. "2014-07-16T00:20:56.277Z",
 "conditions":
     [
         {"acl":"private"},
         {"bucket": The name of the bucket from get_upload_details},
         {"Content-Type":"text/csv"},
         {"success_action_status":"200"},
         {"key": filename of upload, prefixed with 'data_imports/',
                 suffixed with a unique timestamp.
                 e.g. 'data_imports/my_buildings.csv.1405469756'},
         {"x-amz-meta-category":"data_imports"},
         {"x-amz-meta-qqfilename": original filename}
     ]
}
```

Returns:

```
{
    "policy": A hash of the policy document. Using during upload to S3.
    "signature": A signature of the policy document.  Also used during upload to␣
↪S3.
}
```

6.4.7 Module contents

6.5 Features Package

6.5.1 Submodules

6.5.2 Module contents

6.6 Green Button Package

6.6.1 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

6.6.2 Submodules

6.6.3 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'

6.6.4 Module contents

6.7 Landing Package

6.7.1 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

6.7.2 Submodules

6.7.3 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

6.7.4 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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

REQUIRED_FIELDS = ['email']

USERNAME_FIELD = 'username'

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.

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.

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.

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.

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.

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

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

get_absolute_url()

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

get_next_by_date_joined(**moreargs*, ***morekwargs*)

get_previous_by_date_joined(**moreargs*, ***morekwargs*)

get_short_name()
Returns the short name for the user.

greenassessmentpropertyauditlog_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.

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.

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.

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.

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.

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.

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

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.

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.

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.

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.

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.

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.

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

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.

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.

6.7.5 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.

6.7.6 URLs

6.7.7 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*)

6.7.8 Module contents

6.8 Library Packages

6.8.1 Submodules

6.8.2 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*)

6.8.3 Module contents

6.9 Mapping Package

6.9.1 Submodules

6.9.2 seed.mappings.mapper module

6.9.3 seed.mappings.seed_mappings module

6.9.4 Module contents

6.10 Managers Package

6.10.1 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

6.10.2 Submodules

6.10.3 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*)

6.10.4 Module contents

6.11 Models

6.11.1 Submodules

6.11.2 AuditLog

6.11.3 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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

static create_mappings(*mappings*, *organization*, *user*, *import_file_id=None*)
Create the mappings for an organization and a user based on a simple array of array object.

Parameters

- **mappings** – dict, dictionary containing mapping information
- **organization** – inst, organization object
- **user** – inst, User object
- **import_file_id** – integer, If passed, will cache the column mappings data into the import_file_id object.

:return Boolean, True is data are saved in the ColumnMapping table in the database

static create_mappings_from_file(*filename*, *organization*, *user*, *import_file_id=None*)
Load the mappings in from a file in a very specific file format. The columns in the file must be:

1. raw field
2. table name
3. field name
4. field display name
5. field data type
6. field unit type

Parameters

- **filename** – string, absolute path and name of file to load
- **organization** – id, organization id
- **user** – id, user id
- **import_file_id** – Integer, If passed, will cache the column mappings data into the import_file_id object.

Returns ColumnMapping, True

static delete_all(*organization*)

Delete all the columns for an organization. Note that this will invalidate all the data that is in the extra_data fields of the inventory and is irreversible.

Parameters **organization** – instance, Organization

Returns [int, int] Number of columns, column_mappings records that were deleted

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.

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.

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.

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

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.

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 **retrieve_all**(*org_id*, *inventory_type*)
Retrieve all the columns for an organization. First, grab the columns from the # VIEW_COLUMNS_PROPERTY schema which defines the database columns with added data for # various reasons. Then query the database for all extra data columns and add in the # data as appropriate ensuring that duplicates that are taken care of (albeit crudely).

Note: this method should retrieve the columns from MappingData and then have a method # to return for JavaScript (i.e. UI-Grid) or native (standard JSON)

Parameters

- **org_id** – Organization ID
- **inventory_type** – Inventory Type (property|taxlot)

Returns dict

static **retrieve_db_fields**()
return the fields in the database regardless of properties or taxlots

[“address_line_1”, “gross_floor_area”, . . .] :return: list

static **retrieve_db_types**()
return the data types for the database columns in the format of:

Example: {

“field_name”: “data_type”, “field_name_2”: “data_type_2”, “address_line_1”: “string”,

}

Returns dict

static **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 have ever seen for a particular organization.

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

to_dict()
Convert the column object to a dictionary

Returns dict

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

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 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 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 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

get_source_type_display(**moreargs*, ***morekwargs*)

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

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.

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

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'.

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!

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.

to_dict()
Convert the ColumnMapping object to a dictionary

Returns dict

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).

Returns list of mapped items, float representation of confidence.

6.11.4 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 MultipleObjectsReturned

Bases: django.core.exceptions.MultipleObjectsReturned

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_end(**moreargs*, ***morekwargs*)

get_next_by_start(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_end(**moreargs*, ***morekwargs*)

get_previous_by_start(**moreargs*, ***morekwargs*)

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.

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

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.

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.

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_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.

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.

6.11.5 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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

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

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.

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.

6.11.6 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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

get_value_source_display(**moreargs*, ***morekwargs*)

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

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

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

get_compliance_type_display(**moreargs*, ***morekwargs*)

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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_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.

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

get_energy_type_display(**moreargs*, ***morekwargs*)

get_energy_units_display(**moreargs*, ***morekwargs*)

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

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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

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

GRAY_CHOICE = 'gray'

GREEN_CHOICE = 'green'

LIGHT_BLUE_CHOICE = 'light blue'

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

ORANGE_CHOICE = 'orange'

RED_CHOICE = 'red'

WHITE_CHOICE = 'white'

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.

get_color_display(**moreargs*, ***morekwargs*)

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

rule_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.

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.

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.

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

get_unit_type_display(**moreargs*, ***morekwargs*)

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
}
```

6.11.7 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 MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

adding_buildings_status_percentage_cache_key

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.

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.

get_compliance()

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(*moreargs, **morekwargs)

get_previous_by_modified(*moreargs, **morekwargs)

get_status_display(*moreargs, **morekwargs)

has_compliance

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.

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

organization
For compliance with organization names in new data model.

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_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_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_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.

property_count

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.

removing_buildings_status_percentage_cache_key

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.

taxlot_count

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.

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 MultipleObjectsReturned

Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

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.

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

noncanonicalprojectbuildings
Accessor to the related object on the reverse side of a one-to-one relation.

In the example:

```
class Restaurant(Model):
    place = OneToOneField(Place, related_name='restaurant')
```

`place.restaurant` is a `ReverseOneToOneDescriptor` instance.

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

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.

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

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.

6.11.8 Properties

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

The Property is the parent property that ties together all the views of the property. For example, if a building has multiple changes overtime, then this Property will always remain the same. The PropertyView will point to the unchanged property as the PropertyState and Property view are updated.

If the property can be a campus. The property can also reference a parent property.

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

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

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

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.

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_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.

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*, *parent_state1*, *parent_state2*, *state*, *view*, *name*, *description*, *import_filename*, *record_type*, *created*)

Bases: `django.db.models.base.Model`

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

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

get_record_type_display(**moreargs*, ***morekwargs*)

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

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.

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.

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.

parent_state1

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.

parent_state2

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 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__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.

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.

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

clean()

classmethod coparent(*state_id*)
Return the coparent of the PropertyState. This will query the PropertyAuditLog table to determine if there is a coparent and return it if it is found. The state_id needs to be the base ID of when the original record was imported

Parameters **state_id** – integer, state id to find coparent.

Returns dict

get_data_state_display(**moreargs*, ***morekwargs*)

get_merge_state_display(**moreargs*, ***morekwargs*)

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.

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

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.

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)

propertyauditlog__parent_state1
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__parent_state2
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 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_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.

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

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.

A PropertyView contains a reference to a property (which should not change) and to a cycle (time period), and a state (characteristics).

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

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

gapauditlog__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.

greenassessmentproperty_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.

import_filename
Get the import file name form the audit logs

initialize_audit_logs(***kwargs*)

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

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_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.

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.

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.

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.

tax_lot_states()
Return a list of TaxLotStates associated with this PropertyView and Cycle

Returns list of TaxLotStates

tax_lot_views()
Return a list of TaxLotViews that are associated with this PropertyView and Cycle

Returns list of TaxLotViews

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.

6.11.9 TaxLots

class seed.models.tax_lots.**TaxLot**(*id*, *organization*)
Bases: django.db.models.base.Model

exception DoesNotExist
Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned
Bases: django.core.exceptions.MultipleObjectsReturned

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.

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

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.

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*, *parent_state1*, *parent_state2*, *state*, *view*, *name*, *description*, *import_filename*, *record_type*, *created*)

Bases: `django.db.models.base.Model`

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

exception MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

get_record_type_display(**moreargs*, ***morekwargs*)

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

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.

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.

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.

parent_state1
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.

parent_state2

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.

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__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__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.

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*, *address_line_2*, *normalized_address*, *city*, *state*, *postal_code*, *number_properties*, *extra_data*)

Bases: django.db.models.base.Model

exception DoesNotExist

Bases: django.core.exceptions.ObjectDoesNotExist

exception MultipleObjectsReturned

Bases: django.core.exceptions.MultipleObjectsReturned

classmethod coparent(*state_id*)

Return the coparent of the TaxLotState. This will query the TaxLotAuditLog table to determine if there is a coparent and return it if it is found. The state_id needs to be the base ID of when the original record was imported

Parameters **state_id** – integer, state id to find coparent.

Returns dict

get_data_state_display(**moreargs*, ***morekwargs*)

get_merge_state_display(**moreargs*, ***morekwargs*)

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.

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

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.

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)

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

taxlotauditlog__parent_state1

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__parent_state2
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__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.

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.

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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

import_filename
Get the import file name form the audit logs

initialize_audit_logs(***kwargs*)

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

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.

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.

property_states()
Return a list of PropertyStates associated with this TaxLotView and Cycle

Returns list of PropertyStates

property_views()
Return a list of PropertyViews that are associated with this TaxLotView and Cycle

Returns list of PropertyViews

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.

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.

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.

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.

6.11.10 Module contents

6.12 Public Package

6.12.1 Submodules

6.12.2 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 MultipleObjectsReturned
Bases: `django.core.exceptions.MultipleObjectsReturned`

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.

get_field_type_display(**moreargs*, ***morekwargs*)

get_next_by_created(**moreargs*, ***morekwargs*)

get_next_by_modified(**moreargs*, ***morekwargs*)

get_previous_by_created(**moreargs*, ***morekwargs*)

get_previous_by_modified(**moreargs*, ***morekwargs*)

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

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.

6.12.3 Module contents

6.13 SEED Package

6.13.1 Subpackages

Management Package

Subpackages

Management Packages

Submodules

S3

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

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()
Should not 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 have 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_doesnt_delete_user_if_multiple_memberships()
Deleting an org should not delete the orgs users if the user belongs to many orgs.

Views

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_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_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_empty()

test_export_buildings_progress()

test_home()

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_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.

6.13.2 Inheritance

6.13.3 Submodules

6.13.4 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.

6.13.5 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.

6.13.6 Models

6.13.7 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.**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*)

6.13.8 Tasks

6.13.9 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.

6.13.10 URLs

6.13.11 Utils

6.13.12 Views

6.13.13 Module contents

6.14 Serializers Package

6.14.1 Submodules

6.14.2 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*)

6.14.3 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

get_is_applied(*obj*)

6.14.4 Module contents

6.15 URLs Package

6.15.1 Submodules

6.15.2 Accounts

6.15.3 APIs

6.15.4 Main

6.15.5 Projects

6.16 Utilities Package

6.16.1 Submodules

6.16.2 APIs

required approvals from the U.S. Department of Energy) and contributors. All rights reserved. # NOQA :author

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.

class seed.utils.api.**OrgCreateMixin**
Bases: *seed.utils.api.OrgMixin*

Mixin to add organization when creating model instance

perform_create(*serializer*)
Override to add org

class seed.utils.api.**OrgCreateUpdateMixin**
Bases: *seed.utils.api.OrgCreateMixin*, *seed.utils.api.OrgUpdateMixin*

Mixin to add organization when creating/updating model instance

class seed.utils.api.**OrgMixin**
Bases: object

Provides get_organization and get_parent_org method

get_organization(*request*, *return_obj=None*)
Get org from query param or request.user. :param request: request object. :param return_obj: bool. Set to True if obj vs pk is desired. :return: int representing a valid organization pk or

organization object.

get_parent_org(*request*)
Gets parent organization of org from query param or request. :param request: Request object. :return: organization object.

class seed.utils.api.**OrgQuerySetMixin**
Bases: *seed.utils.api.OrgMixin*

Mixin proving a get_queryset method that filters on organization.

In order to use this mixin you must specify the model attributes on the View[Set] class. By default it assumes there is an organization field on the model. You can override this by setting the orgfilter attribute to the appropriate fieldname. This also allows nested fields e.g. foreign_key.organization By default this retrieves organization from query string param OR the default_organization or first returned organization of the logged in user. You can force it to return the appropriate "parent" organization by setting the force_parent attribute to True.

get_queryset()
"get_queryset filtered on organization

class seed.utils.api.**OrgUpdateMixin**
Bases: *seed.utils.api.OrgMixin*

Mixin to add organization when updating model instance

perform_update(*serializer*)
Override to add org

class seed.utils.api.**OrgValidateMixin**
Bases: object

Mixin to provide a validate() method organization to ensure users belongs to the same org as the instance referenced by a foreign key..

You must set org_validators on the Serializer that uses this Mixin. This is a list of OrgValidator named tuples (where key is the key on request data representing the foreign key, and field the foreign key that represents the organization on the corresponding model.

my_validator = OrgValidator(key='foreign_key, field='organization_id')

..example:

class MySerializer(OrgValidateMixin, serializers.ModelSerializer):

foreign_key= serializers.PrimaryKeyRelatedField(query_set=MyModel.objects.all()

) org_validators = [my_validator]

This ensures request.user belongs to the org MyModel.organization

You can traverse foreign key relationships by using a double underscore in validator.field

In the example above setting validator field to be 'property__org_id' is equivalent to MyModel.property.org_id

If you use this Mixin and write a validate method, you must call super to ensure validation takes place.

validate(*data*)
Object level validation. Checks for self.org_validators on Serializers and ensures users belongs to org corresponding to the foreign key being set.

validate_org(*instance*, *user*, *validator*)
Raise error if orgs do not match. :param instance: value in request.data.get(key) to check against :type instance: model instance :param: org_id of user, from get_org_id(request) :type org_id: int :param validator: validator to user :type: OrgValidator named tuple

class seed.utils.api.**OrgValidator**(*key*, *field*)
Bases: tuple

field
Alias for field number 1

key
Alias for field number 0

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).

seed.utils.api.**get_api_request_user**(*request*)
Determines if this is an API request and returns the corresponding user if so.

seed.utils.api.**get_org_id_from_validator**(*instance*, *field*)
For querysets Django enables you to do things like:

note double underscore. However you can't do:

This presents an issue as getattr only works 1 level deep:

getattr(obj, 'org.id') does not work either.

This can be worked around using rgetattr (above). This functions mimics getattr(obj, 'org__id') by splitting field on __ and calling rgetattr on the result.

seed.utils.api.**rgetattr**(*obj*, *lst*)
This enables recursive getattr look ups. given obj, ['a', 'b', 'c'] as params it will look up: obj.a, a.b, b.c returning b.c unless one of the previous values was None, in which case it returns None immediately.

Parameters

- **obj** (*object*) – initial object to examine
- **lst** (list) – list of successive attributes to look up

6.16.3 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

6.16.4 Constants

6.16.5 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.

seed.utils.mapping.**get_table_and_column_names**(*column_mapping*, *attr_name='column_raw'*)
Turns the Column.column_names into a serializable list of str.

6.16.6 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.

6.16.7 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

6.16.8 Time

seed.utils.time.**convert_datestr**(*datestr*, *make_tz_aware=False*)
Converts dates like *12/31/2010* into datetime objects. Dates are returned in UTC time

TODO: reconcile this with seed/lib/mcm/cleaners.py#L85-L85

Parameters

- **datestr** – string, value to convert
- **make_tz_aware** – bool, if set to true, then will convert the timezone into UTC time

Returns datetime or None

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.

6.17 Views Package

6.17.1 Submodules

6.17.2 Accounts

6.17.3 APIs

seed.views.api.**get_api_schema**(*request*, **args*, ***kwargs*)
Returns a hash of all API endpoints and their descriptions.

Returns:

```
{
    '/example/url/here': {
        'name': endpoint name,
        'description': endpoint description
    }...
}
```

Todo: Format docstrings better.

6.17.4 Main

seed.views.main.**angular_js_tests**(*request*)
Jasmine JS unit test code covering AngularJS unit tests

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_inventory**(*request*, **args*, ***kwargs*)
Starts a background task to delete all properties & taxlots in an org.

DELETE Expects 'org_id' for the organization.

Returns:

```
{
    'status': 'success' or 'error',
    'progress_key': ID of background job, for retrieving job progress
}
```

seed.views.main.**error404**(*request*)

seed.views.main.**error500**(*request*)

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_columns**(*request*, **args*, ***kwargs*)
Returns a JSON list of columns a user can select as his/her default

Requires the organization_id as a query parameter

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.**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.**progress**(*request*, **args*, ***kwargs*)
Get the progress (percent complete) for a task.

Payload:

```
{
    'progress_key': The progress key from starting a background task
}
```

Returns:

```
{
    'progress_key': The same progress key,
    'progress': Percent completion
}
```

seed.views.main.**public_search**(*request*, **args*, ***kwargs*)
the public API unauthenticated endpoint

see `search_buildings` for the non-public version

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*)

seed.views.main.**version**(*request*, **args*, ***kwargs*)
Returns the SEED version and current git sha

6.17.5 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.

6.17.6 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.TaxLo

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*)

6.17.7 Module contents

CHAPTER 7

Developer Resources

7.1 General Notes

7.1.1 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
```

7.2 Django Notes

Both Django and AngurlarJS are used for url routing. Django routes are in *seed/urls/main.py*

7.2.1 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'
```

7.3 AngularJS Integration Notes

7.3.1 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', ['$interpolateProvider'], function (
↪$interpolateProvider) {
        $interpolateProvider.startSymbol("{$");
        $interpolateProvider.endSymbol("$}");
    }
);
```

7.3.2 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'];
});
```

7.3.3 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/**'
    ]);
});
```

7.4 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

7.5 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. Currently, the list of Columns is dynamically populated by importing data.

There are default mappings for ESPM are located here:

https://github.com/SEED-platform/seed/blob/develop/seed/lib/mappings/data/pm-mapping.json

7.6 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
```

7.7 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

9.1 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

9.2 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

Python Module Index

C

S

Index

C

D

E

F

G

H

I

P

Q

R

S

T

U

V

W