SEED Documentation

Release 0.1.0

config

January 13, 2016

Contents

Github project page

Getting Started:

Install Guide

SEED is intended to be installed on Linux instances in the cloud(AWS), and on local hardware. For Windows installation, see the Django notes.

1.1 AWS Setup

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

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

1.1.1 Pre-requisites

Ubuntu server 13.10 or newer, with the following list of *aptitude packages* installed. prerequisites.txt

Copy the *prerequisites.txt* files to the server and install the dependencies:

```
$ sudo dpkg --set-selections < ./prerequisites.txt
$ sudo apt-get dselect-upgrade
```

or with a single command as su

```
# aptitude install $(cat ./prerequisites.txt | awk '{print $1}')
```

Note: postgresql server is not included above, and it is assumed that the system will use the AWS RDS postgresql service

---

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

A smaller list of packages to get going:

```
$ sudo apt-get install python-pip python-dev libatlas-base-dev gfortran \
python-dev build-essential g++ npm libxml2-dev libxslt1-dev \
postgresql-devel postgresql-9.3 postgresql-server-dev-9.3 libpq-dev \
libmemcached-dev openjdk-7-jre-headless
```

Amazon Web Services (AWS) Dependencies

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

- RDS (PostgreSQL >=9.3)

- ElastiCache (redis)

- SES

- S3

1.1.2 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.txt

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

1.1.3 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 `grunt-cli` will be installed globally by the `install_javascript_dependencies` script. The Ubuntu version `13.10` requires a cusomt install of nodejs/npm, and an install scrpt (`bin/node-and-npm-in-30s.sh`) is provided to download a stable release and install `npm` assuming the prerequisites are met.

```
$ sudo apt-get install build-essential
$ sudo apt-get install libssl-dev
$ sudo apt-get install curl
$ . bin/node-and-npm-in-30s.sh

$ bin/install_javascript_dependencies.sh
```

1.1.4 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 9.3 database instance you have manually installed within your infrastructure.

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

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.

create the database within the postgres `psql` shell:

```
postgres-user=# CREATE DATABASE seed;
```

or from the command line:

```
$ createdb seed
```

create the database tables and migrations:

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

Note: running migrations can be shortened into a one-liner `./manage.py syncdb --migrate`

---

create a superuser to access the system

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

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.

1.1.5 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 `BROKER_URL` settings.

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

Note: The popular `memcached` can also be used as a cache back-end, but is not supported and redis has a different cache key format, which could cause breakage and isn't tested. Likewise, `rabbitmq` or AWS `SQS` are alternative message brokers, which could cause breakage and is not tested.

1.1.6 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:

```
$ python manage.py celery worker -B -c 2 --loglevel=INFO -E --maxtasksperchild=1000
```

1.1.7 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.

```
$ python manage.py runserver
```

1.1.8 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
```

1.2 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.

1.2.1 Pre-requisites

Ubuntu server 14.04 or newer

We need to install the base packages needed to run the app:

```
$ 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

1.2.2 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

1.2.3 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.txt

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

1.2.4 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 `grunt-cli` will be installed globally by the `install_javascript_dependencies` script. The Ubuntu version `14.04` requires a cusomt install of nodejs/npm, and an install scrpt (`bin/node-and-npm-in-30s.sh`) is provided to download a stable release and install `npm` assuming the prerequisites are met.

```
$ . bin/node-and-npm-in-30s.sh
```

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

1.2.5 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 9.3 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 conect to the configured database.

create the database tables and migrations:

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

Note: running migrations can be shortened into a one-liner `./manage.py syncdb --migrate`

1.2.6 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 `BROKER_URL` settings.

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

Note: The popular `memcached` can also be used as a cache back-end, but is not supported and redis has a different cache key format, which could cause breakage and isn't tested. Likewise, `rabbitmq` or AWS `SQS` are alternative message brokers, which could cause breakage and is not tested.

1.2.7 Creating the initial user

create a superuser to access the system

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

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.

1.2.8 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:

```
$ python manage.py celery worker -B -c 2 --loglevel=INFO -E --maxtasksperchild=1000
```

1.2.9 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
```

1.2.10 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`:

```python
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/u
```

> **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.

1.2.11 environmental variables

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

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

1.2.12 SMTP service

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

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

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

Django can likewsie 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'
```

1.2.13 local_untracked.py

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

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

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

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

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

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

API Usage

Authentication

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

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

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e https://seeddom
```

Or using the Python Requests library:

```
headers = {'authorization': 'user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e'}
result = requests.get('https://seeddomain.com/app/api/get_api_schema/',
                      headers=headers)
print result.json()
```

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

Payloads

Many requests require a json-encoded payload and/or parameters in the query string of the url. A frequent requirement is including the organization_id of the org you belong to. E.g.:

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e \
        https://seeddomain.com/app/accounts/get_organization?organization_id={your org id here}
```

Or in a json payload:

```
curl -H Authorization:user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e \
        -d '{"organization_id":6, "user_id": 12, "role": "viewer"}' \
        https://seeddomain/app/accounts/update_role/
```

Using Python:

```
headers = {'authorization': 'user@email_address.com:5edfd7f1f0696d4139118f8b95ab1f05d0dd418e'}
params = json.dumps({'organization_id': 6, 'user_id': 12, 'role': 'viewer'})
result = requests.post('https://seeddomain.com/app/accounts/update_role/',
                       data=params,
                       headers=headers)
print result.json()
```

Responses

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

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

Sample Client

A python-based API client is included in seed.utils.api_client and documented here: `api_client`

6.1 Api-related endpoints

`seed.views.api.`**`get_api_schema`**`()`

>   **URI** /app/api/get_api_schema/

Returns a hash of all API endpoints and their descriptions.

Returns:

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

TODO: Should this require authentication? Should it limit the return to endpoints the user has authorization for?

TODO: Format docstrings better.

6.2 Account management endpoints

`seed.views.accounts.`**`add_org`**`()`

>   **URI** /app/accounts/add_org/

Creates a new organization.

Payload:

```
{
 'organization_name': The name of the new org,
 'user_id': the user id of the owner of the new org,
}
```

Returns:

```
{
'status': 'success' or 'error',
'message': message, if any,
'organization_id': The ID of the new org, if created.
}
```

seed.views.accounts.**add_user**()

> **URI** /app/accounts/add_user/

Creates a new SEED user. One of 'organization_id' or 'org_name' is needed. Sends invitation email to the new user.

Payload:

```
{
    'organization_id': ID of an existing org to add the new user to,
    'org_name': Name of a new org to create with user as owner
    'first_name': First name of new user
    'last_name': Last name of new user
    'role': {
        'value': The permission level of new user within this org
            (one of member, viewer, owner)
    },
    'email': Email address of new user.
}
```

Returns:

```
{
 'status': 'success',
 'message': email address of new user,
 'org': name of the new org (or existing org),
 'org_created': True if new org created,
 'username': Username of new user
}
```

seed.views.accounts.**add_user_to_organization**()

> **URI** /app/accounts/add_user_to_organization/

Adds an existing user to an organization.

Payload:

```
{
 'organization_id': The ID of the organization,
 'user_id': the user id of the owner of the new org,
}
```

Returns:

```
{
'status': 'success' or 'error',
'message': message, if any,
}
```

seed.views.accounts.**create_sub_org**()

> **URI** /app/accounts/create_sub_org/

Creates a child org of a parent org.

Payload:

```
{
    'parent_org_id': ID of the parent org,
    'sub_org': {
        'name': Name of new sub org,
        'email': Email address of owner of sub org, which
                must already exist
    }
}
```

Returns:

```
{
 'status': 'success' or 'error',
 'message': Error message, if any,
 'organization_id': ID of newly-created org
}
```

seed.views.accounts.**get_organization**()

> URI /app/accounts/get_organization/

Retrieves a single organization by id.

> GET Expects ?organization_id=(:org_id)

Returns:

```
{'status': 'success or error', 'message': 'error message, if any',
 'organization':
    {'name': org name,
     'org_id': org's identifier (used with Authorization header),
     'id': org's identifier,
     'number_of_users': count of members of org,
     'user_is_owner': True if the user is owner of this org,
     'user_role': The role of user in this org (owner, viewer, member),
     'owners': [
         {
         'first_name': the owner's first name,
         'last_name': the owner's last name,
         'email': the owner's email address,
         'id': the owner's identifier (int)
         }
        ]
     'sub_orgs': [ a list of orgs having this org as parent, in
                 the same format...],
     'is_parent': True if this org contains suborgs,
     'num_buildings': Count of buildings belonging to this org
    }
}
```

seed.views.accounts.**get_organizations**()

> URI /app/accounts/get_organizations/

Retrieves all orgs the user has access to.

Returns:

```
{'organizations': [
    {'name': org name,
```

```
            'org_id': org's identifier (used with Authorization header),
            'id': org's identifier,
            'number_of_users': count of members of org,
            'user_is_owner': True if the user is owner of this org,
            'user_role': The role of user in this org (owner, viewer, member),
            'owners': [
                        {
                         'first_name': the owner's first name,
                         'last_name': the owner's last name,
                         'email': the owner's email address,
                         'id': the owner's identifier (int)
                        }
                      ]
            'sub_orgs': [ a list of orgs having this org as parent, in
                        the same format...],
            'is_parent': True if this org contains suborgs,
            'num_buildings': Count of buildings belonging to this org
          }...
        ]
    }
```

seed.views.accounts.**get_organizations_users**()

> **URI** /app/accounts/get_organizations_users/

Retrieve all users belonging to an org.

Payload:

```
{'organization_id': org_id}
```

Returns:

```
{'status': 'success',
 'users': [
     {
       'first_name': the user's first name,
       'last_name': the user's last name,
       'email': the user's email address,
       'id': the user's identifier (int),
       'role': the user's role ('owner', 'member', 'viewer')
     }
   ]
}
```

> TODO(ALECK/GAVIN): check permissions that request.user is owner or admin and get more info about the users.

seed.views.accounts.**get_query_threshold**()

> **URI** /app/accounts/get_query_threshold/

Returns the "query_threshold" for an org. Searches from members of sibling orgs must return at least this many buildings from orgs they do not belong to, or else buildings from orgs they don't belong to will be removed from the results.

> **GET** Expects organization_id in the query string.

Returns:

```
{
  'status': 'success',
```

```
    'query_threshold': The minimum number of buildings that must be
        returned from a search to avoid squelching non-member-org results.
}
```

seed.views.accounts.**get_shared_fields**()

> **URI** /app/accounts/get_shared_fields/

Retrieves all fields marked as shared for this org tree.

> **GET** Expects organization_id in the query string.

Returns:

```
{
 'status': 'success',
 'shared_fields': [
     {
      "title": Display name of field,
      "sort_column": database/search name of field,
      "class": css used for field,
      "title_class": css used for title,
      "type": data type of field,
          (One of: 'date', 'floor_area', 'link', 'string', 'number')
      "field_type": classification of field.  One of:
          'contact_information', 'building_information',
          'assessor', 'pm',
      "sortable": True if buildings can be sorted on this field,
     }
     ...
   ],
   'public_fields': [
       {
          "title": Display name of field,
          "sort_column": database/search name of field,
          "class": css used for field,
          "title_class": css used for title,
          "type": data type of field,
            (One of: 'date', 'floor_area', 'link', 'string', 'number')
          "field_type": classification of field.  One of:
              'contact_information', 'building_information',
              'assessor', 'pm',
          "sortable": True if buildings can be sorted on this field,
         }
         ...
     ]
}
```

seed.views.accounts.**get_user_profile**()

> **URI** /app/accounts/get_user_profile/

Retrieves the request's user's first_name, last_name, email and api key if exists.

Returns:

```
{
    'status': 'success',
    'user': {
        'first_name': user's first name,
        'last_name': user's last name,
```

```
            'email': user's email,
            'api_key': user's API key
        }
    }
```

seed.views.accounts.**remove_user_from_org**()

> **URI** /app/accounts/remove_user_from_org/

Removes a user from an organization.

Payload:

```
{
    'organization_id': ID of the org,
    'user_id': ID of the user
}
```

Returns:

```
{
 'status': 'success' or 'error',
 'message': 'error message, if any'
}
```

seed.views.accounts.**save_org_settings**()

> **URI** /app/accounts/save_org_settings/

Saves an organzation's settings: name, query threshold, shared fields

Payload:

```
{
    'organization_id: 2,
    'organization': {
        'query_threshold': 2,
        'name': 'demo org',
        'fields': [  # All internal sibling org shared fields
            {
                'sort_column': database/search field name,
                    e.g. 'pm_property_id',
            }
        ],
        'public_fields': [  # All publicly shared fields
            {
                'sort_column': database/search field name,
                    e.g. 'pm_property_id',
            }
        ],
    }
}
```

Returns:

```
{
    'status': 'success or error',
    'message': 'error message, if any'
}
```

seed.views.accounts.**update_role**()

URI /app/accounts/update_role/

Sets a user's role within an organization.

Payload:

```
{
    'organization_id': organization's id,
    'user_id': user's id,
    'role': one of 'owner', 'member', 'viewer'
}
```

Returns:

```
{'status': 'success or error',
 'message': 'error message, if any'}
```

`seed.views.accounts.`**`update_user`**`()`

URI /app/accounts/update_user/

Updates the request's user's first name, last name, and email

Payload:

```
{
 'user': {
         'first_name': :first_name,
         'last_name': :last_name,
         'email': :email
       }
}
```

Returns:

```
{
    'status': 'success',
    'user': {
        'first_name': user's first name,
        'last_name': user's last name,
        'email': user's email,
        'api_key': user's API key
    }
}
```

6.3 File upload endpoints

These endpoints behave drastically differently depending on whether the system is configured to upload files to S3 or to a local filesystem.

6.4 Seed (building and project) endpoints

`seed.views.main.`**`create_dataset`**`()`

URI /app/create_dataset/

Creates a new empty dataset (ImportRecord).

Payload:

```
{
 "name": Name of new dataset, e.g. "2013 city compliance dataset"
 "organization_id": ID of the org this dataset belongs to
}
```

Returns:

```
{'status': 'success',
 'id': The ID of the newly-created ImportRecord,
 'name': The name of the newly-created ImportRecord
}
```

seed.views.main.**create_pm_mapping**()

> **URI** /app/create_pm_mapping/

Create a mapping for PortfolioManager input columns.

Payload:

```
{
    columns: [ "name1", "name2", ... , "nameN"],
}
```

Returns:

```
{
    success: true,
    mapping: [
        ["name1", "mapped1", {bedes: true|false, numeric: true|false}],
        ["name2", "mapped2", {bedes: true|false, numeric: true|false}],
        ...
        ["nameN", "mappedN", {bedes: true|false, numeric: true|false}]
    ]
}
-- OR --
{
    success: false,
    reason: "message goes here"
}
```

seed.views.main.**delete_buildings**()

> **URI** /app/delete_buildings/

Deletes all BuildingSnapshots the user has selected.

Does not delete selected_buildings where the user is not a member or owner of the organization the selected building belongs. Since search shows buildings across all the orgs a user belongs, it's possible for a building to belong to an org outside of *org_id*.

> **DELETE** Expects 'org_id' for the organization, and the search payload

similar to add_buildings/create_project

> **{** 'organization_id': 2, 'search_payload': {
>
> > 'selected_buildings': [2, 3, 4], 'select_all_checkbox': False, 'filter_params': ... **//** see search_buildings

```
        }

    }
```

Returns:

```
{'status': 'success' or 'error'}
```

seed.views.main.**delete_dataset**()

> URI /app/delete_dataset/

Deletes all files from a dataset and the dataset itself.

> DELETE Expects 'dataset_id' for an ImportRecord in the query string.

Returns:

```
{'status': 'success' or 'error',
 'message': 'error message, if any'
}
```

seed.views.main.**delete_duplicates_from_import_file**()

> URI /app/delete_duplicates_from_import_file/

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

> GET Expects import_file_id corresponding to the ImportFile in question.

Returns:

```
{'status': 'success',
 'deleted': Number of duplicates deleted
}
```

seed.views.main.**delete_file**()

> URI /app/delete_file/

Deletes an ImportFile from a dataset.

Payload:: {

> "file_id": ImportFile id, "organization_id": current user organization id

}

Returns:

```
{'status': 'success' or 'error',
 'message': 'error message, if any'
}
```

seed.views.main.**delete_organization_buildings**()

> URI /app/delete_organization_buildings/

Starts a background task to delete all BuildingSnapshots in an org.

> GET Expects 'org_id' for the organization.

Returns:

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

```
seed.views.main.export_buildings()
```

> **URI** /app/export_buildings/

Begins a building export process.

Payload:

```
{
  "export_name": "My Export",
  "export_type": "csv",
  "selected_building": [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()
```

> **URI** /app/export_buildings/download/

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()
```

> **URI** /app/export_buildings/progress/

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_PM_filter_by_counts**()

>> **URI** /app/get_PM_filter_by_counts/

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

>> **GET** Expects import_file_id corresponding to the ImportFile in question.

Returns:

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

seed.views.main.**get_aggregated_building_report_data**()

>> **URI** /app/get_aggregated_building_report_data/

This method returns a set of aggregated building data for graphing. It expects as parameters

>> **GET**

>> •start_date: The starting date for the data series with the format *YYYY-MM-DDThh:mm:ss+hhmm* # NOQA

>> •end_date: The starting date for the data series with the format *YYYY-MM-DDThh:mm:ss+hhmm* # NOQA

>> •x_var: The variable name to be assigned to the "x" value in the returned data series # NOQA

>> •y_var: The variable name to be assigned to the "y" value in the returned data series # NOQA

>> •organization_id: The organization to be used when querying data.

The x_var values should be from the following set of variable names:

>> •site_eui

>> •source_eui

>> •site_eui_weather_normalized

>> •source_eui_weather_normalized

>> •energy_score

The y_var values should be from the following set of variable names:

>> •gross_floor_area

>> •use_description

>> •year_built

This method includes building record count information as part of the result JSON in a property called "building_counts."

This property provides data on the total number of buildings available in each 'year ending' group, as well as the subset of those buildings that have actual data to graph. By sending these values in the result we allow the client to easily build a message like "200 of 250 buildings in this group have data."

Returns:: The returned JSON document that has the following structure. ""

>> **{** "status": "success", "chart_data": [

>>>> **{** "yr_e": x - group by year ending "x": x, - median value in group "y": y - average value thing

>>>> }, {

> "yr_e": x "x": x, "y": y

>], "building_counts": [

>> { "yr_e": string for year ending - group by "num_buildings": number of buildings in query results "num_buildings_w_data": number of buildings with valid data in this group, BOTH x and y? # NOQA

>] "num_buildings": total number of buildings in query results, "num_buildings_w_data": total number of buildings with valid data in query results

> }

"""

—

parameters:

> - name: x_var description: Name of column in building snapshot database to be used for "x" axis required: true type: string paramType: query

> - name: y_var description: Name of column in building snapshot database to be used for "y" axis required: true type: string paramType: query

> - start_date: description: The start date for the entire dataset. required: true type: string paramType: query

> - end_date: description: The end date for the entire dataset. required: true type: string paramType: query

> - name: organization_id description: User's organization which should be used to filter building query results required: true type: string paramType: query

type:

> **status:** required: true type: string

> **chart_data:** required: true type: array

> **building_counts:** required: true type: array

> **num_buildings:** required: true type: string

> **num_buildings_w_data:** required: true type: string

responseMessages:

> - code: 400 message: Bad request, only GET method is available

> - code: 401 message: Not authenticated

> - code: 403 message: Insufficient rights to call this procedure

seed.views.main.**get_building**()

> **URI** /app/get_building/

Retrieves a building. If user doesn't belong to the building's org, fields will be masked to only those shared within the parent org's structure.

> **GET** Expects building_id and organization_id in query string.

building_id should be the *caninical_building* ID for the building, not the BuildingSnapshot id.

Returns:

```
{
    'status': 'success or error',
    'message': 'error message, if any',
    'building': {'id': the building's id,
                 'canonical_building': the canonical building ID,
                 other fields this user has access to...
    },
    'imported_buildings': [ A list of buildings imported to create
                            this building's record, in the same
                            format as 'building'
                          ],
    'projects': [
        // A list of the building's projects
        {
            "building": {
                "approved_date":07/30/2014,
                "compliant": null,
                "approver": "demo@buildingenergy.com"
                "approved_date": "07/30/2014"
                "compliant": null
                "label": {
                    "color": "red",
                    "name": "non compliant",
                    id: 1
                }
            }
            "description": null
            "id": 3
            "is_compliance": false
            "last_modified_by_id": 1
            "name": "project 1"
            "owner_id": 1
            "slug": "project-1"
            "status": 1
            "super_organization_id": 1
        },
        . . .
    ],
     'user_role': role of user in this org,
     'user_org_id': the org id this user belongs to
}
```

seed.views.main.**get_building_report_data**()

> URI /app/get_building_report_data/

This method returns a set of x,y building data for graphing. It expects as parameters

> **GET**

> •start_date: The starting date for the data series with the format *YYYY-MM-DD*

> •end_date: The starting date for the data series with the format *YYYY-MM-DD*

> •x_var: The variable name to be assigned to the "x" value in the returned data series # NOQA

> •y_var: The variable name to be assigned to the "y" value in the returned data series # NOQA

> •organization_id: The organization to be used when querying data.

The x_var values should be from the following set of variable names:

- site_eui

- source_eui

- site_eui_weather_normalized

- source_eui_weather_normalized

- energy_score

The y_var values should be from the following set of variable names:

- gross_floor_area

- use_description

- year_built

This method includes building record count information as part of the result JSON in a property called "building_counts."

This property provides data on the total number of buildings available in each 'year ending' group, as well as the subset of those buildings that have actual data to graph. By sending these values in the result we allow the client to easily build a message like "200 of 250 buildings in this group have data."

Returns:: The returned JSON document that has the following structure. """

{ "status": "success", "chart_data": [

{ "id" the id of the building, "yr_e": the year ending value for this data point "x": value for x var, "y": value for y var,

], "building_counts": [

{ "yr_e": string for year ending "num_buildings": number of buildings in query results "num_buildings_w_data": number of buildings with valid data in query results

] "num_buildings": total number of buildings in query results, "num_buildings_w_data": total number of buildings with valid data in the query results # NOQA

}

"""

—

parameters:

- name: x_var description: Name of column in building snapshot database to be used for "x" axis required: true type: string paramType: query

- name: y_var description: Name of column in building snapshot database to be used for "y" axis required: true type: string paramType: query

- start_date: description: The start date for the entire dataset. required: true type: string paramType: query

- end_date: description: The end date for the entire dataset. required: true type: string paramType: query

- name: organization_id description: User's organization which should be used to filter building query results required: true type: string paramType: query

- name: aggregate description: Aggregates data based on internal rules (given x and y var) required: true type: string paramType: query

type:

status: required: true type: string

chart_data: required: true type: array

num_buildings: required: true type: string

num_buildings_w_data: required: true type: string

responseMessages:

- code: 400 message: Bad request, only GET method is available

- code: 401 message: Not authenticated

- code: 403 message: Insufficient rights to call this procedure

`seed.views.main.`**`get_building_summary_report_data`**`()`

    URI /app/get_building_summary_report_data/

This method returns basic, high-level data about a set of buildings, fitered by organization ID.

It expects as parameters

    GET

•start_date: The starting date for the data series with the format *YYYY-MM-DD*

•end_date: The starting date for the data series with the format *YYYY-MM-DD*

Returns:: The returned JSON document that has the following structure. ""

    { "status": "success", "summary_data": {

        "num_buildings": number of buildings returned from query, "avg_eui": average EUI for returned buildings, "avg_energy_score": average energy score for returned buildings

      }

    }

""

Units for return values are as follows:

` | property | units | |-----------------------|------------| | avg_eui | kBtu-ft2 | `

—

parameters:

- name: organization_id description: User's organization which should be used to filter building query results required: true type: string paramType: query

- start_date: description: The start date for the entire dataset. required: true type: string paramType: query

- end_date: description: The end date for the entire dataset. required: true type: string paramType: query

type:

status: required: true type: string

summary_data: required: true type: object

responseMessages:

- code: 400 message: Bad request, only GET method is available

- code: 401 message: Not authenticated

- code: 403 message: Insufficient rights to call this procedure

seed.views.main.**get_column_mapping_suggestions**()

> **URI** /app/get_column_mapping_suggestions/

Returns suggested mappings from an uploaded file's headers to known data fields.

Payload:

```
{'import_file_id': The ID of the ImportRecord to examine,
 'org_id': The ID of the user's organization}
```

Returns:

```
{'status': 'success',
 'suggested_column_mappings':
       {
        column header from file: [ (destination_column, score) ...]
        ...
       }
 'building_columns': [ a list of all possible columns ],
 'building_column_types': [a list of column types corresponding to
                          building_columns],
     ]
}
```

seed.views.main.**get_coparents**()

> **URI** /app/get_coparents/

Returns the nodes in the BuildingSnapshot tree that can be unmatched.

> **GET** Expects organization_id and building_id in the query string

Returns:

```
{
    'status': 'success',
    'coparents': [
        {
            "id": 333,
            "coparent": 223,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1123
        },
        {
            "id": 223,
            "coparent": 333,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1124
        },
        ...
    ]
}
```

seed.views.main.**get_dataset**()

> **URI** /app/get_dataset/

Retrieves ImportFile objects for one ImportRecord.

> **GET** Expects dataset_id for an ImportRecord in the query string.

Returns:

```
{'status': 'success',
 'dataset':
    {'name': Name of ImportRecord,
     'number_of_buildings': Total number of buildings in
                            all ImportFiles for this dataset,
     'id': ID of ImportRecord,
     'updated_at': Timestamp of when ImportRecord was last modified,
     'last_modified_by': Email address of user making last change,
     'importfiles': [
         {'name': Name of associated ImportFile, e.g. 'buildings.csv',
          'number_of_buildings': Count of buildings in this file,
          'number_of_mappings': Number of mapped headers to fields,
          'number_of_cleanings': Number of fields cleaned,
          'source_type': Type of file (see source_types),
          'id': ID of ImportFile (needed for most operations)
         }
        ],
        ...
       },
       ...
   }
```

seed.views.main.**get_datasets**()

> **URI** /app/get_datasets/

Retrieves all datasets for the user's organization.

> **GET** Expects 'organization_id' of org to retrieve datasets from in query string.

Returns:

```
{'status': 'success',
 'datasets':  [
    {'name': Name of ImportRecord,
     'number_of_buildings': Total number of buildings in
                            all ImportFiles,
     'id': ID of ImportRecord,
     'updated_at': Timestamp of when ImportRecord was last modified,
     'last_modified_by': Email address of user making last change,
     'importfiles': [
         {'name': Name of associated ImportFile, e.g. 'buildings.csv',
          'number_of_buildings': Count of buildings in this file,
          'number_of_mappings': Number of mapped headers to fields,
          'number_of_cleanings': Number of fields cleaned,
          'source_type': Type of file (see source_types),
          'id': ID of ImportFile (needed for most operations)
         }
        ],
        ...
       },
       ...
      ]
   }
```

seed.views.main.**get_datasets_count**()

> **URI** /app/get_datasets_count/

Retrieves the number of datasets for an org.

> **GET** Expects organization_id in the query string.

Returns:

```
{'status': 'success',
 'datasets_count': Number of datasets belonging to this org.
}
```

seed.views.main.**get_first_five_rows**()

> **URI** /app/get_first_five_rows/

Retrieves the first five rows of an ImportFile.

Payload:

```
{'import_file_id': The ID of the ImportFile}
```

Returns:

```
{'status': 'success',
 'first_five_rows': [
     [list of strings of header row],
     [list of strings of first data row],
     ...
     [list of strings of fourth data row]
 ]
}
```

seed.views.main.**get_import_file**()

> **URI** /app/get_import_file/

Retrieves details about an ImportFile.

> **GET** Expects import_file_id in the query string.

Returns:

```
{'status': 'success',
 'import_file': {
     "name": Name of the uploaded file,
     "number_of_buildings": number of buildings in the file,
     "number_of_mappings": number of mapped columns,
     "number_of_cleanings": number of cleaned fields,
     "source_type": type of data in file, e.g. 'Assessed Raw'
     "number_of_matchings": Number of matched buildings in file,
     "id": ImportFile ID,
     'dataset': {
         'name': Name of ImportRecord file belongs to,
         'id': ID of ImportRecord file belongs to,
         'importfiles': [  # All ImportFiles in this ImportRecord, with
             # requested ImportFile first:
             {'name': Name of file,
              'id': ID of ImportFile
             }
             ...
         ]
```

```
            }
        }
    }
```

seed.views.main.**get_match_tree**()

> URI /app/get_match_tree/

returns the BuildingSnapshot tree

> GET Expects organization_id and building_id in the query string

Returns:

```
{
    'status': 'success',
    'match_tree': [ // array of all the members of the tree
        {
            "id": 333,
            "coparent": 223,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1123
        },
        {
            "id": 223,
            "coparent": 333,
            "child": 443,
            "parents": [],
            "canonical_building_id": 1124
        },
        {
            "id": 443,
            "coparent": null,
            "child": 9933,
            "parents": [333, 223],
            "canonical_building_id": 1123
        },
        {
            "id": 9933,
            "coparent": null,
            "child": null,
            "parents": [443],
            "canonical_building_id": 1123
        },
        ...
    ]
}
```

seed.views.main.**get_raw_column_names**()

> URI /app/get_raw_column_names/

Retrieves a list of all column names from an ImportFile.

Payload:

{'import_file_id': The ID of the ImportFile}

Returns:

```
{'status': 'success',
 'raw_columns': [
     list of strings of the header row of the ImportFile
     ]
}
```

seed.views.main.**progress**()

> **URI** /app/progress/

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.**remap_buildings**()

> **URI** /app/remap_buildings/

Re-run the background task to remap buildings as if it hadn't happened at all. Deletes mapped buildings for a given ImportRecord, resets status.

NB: will not work if buildings have been merged into CanonicalBuilings.

Payload:

```
{'file_id': The ID of the ImportFile to be remapped}
```

Returns:

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

seed.views.main.**save_column_mappings**()

> **URI** /app/save_column_mappings/

Saves the mappings between the raw headers of an ImportFile and the destination fields in the BuildingSnapshot model.

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

Payload:

```
{
    "import_file_id": ID of the ImportFile record,
    "mappings": [
        ["destination_field": "raw_field"],  #direct mapping
        ["destination_field2":
            ["raw_field1", "raw_field2"],  #concatenated mapping
        ...
    ]
}
```

Returns:

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

seed.views.main.**save_match**()

> **URI** /app/save_match/

Adds or removes a match between two BuildingSnapshots. Creating a match creates a new BuildingSnapshot with merged data.

Payload:

```
{
 'organization_id': current user organization id,
 'source_building_id': ID of first BuildingSnapshot,
 'target_building_id': ID of second BuildingSnapshot,
 'create_match': True to create match, False to remove it,
 'organization_id': ID of user's organization
}
```

Returns:

```
{
    'status': 'success',
    'child_id': The ID of the newly-created BuildingSnapshot
                containing merged data from the two parents.
}
```

seed.views.main.**save_raw_data**()

> **URI** /app/save_raw_data/

Starts a background task to import raw data from an ImportFile into BuildingSnapshot objects.

Payload:

```
{ 'file_id': The ID of the ImportFile to be saved }
```

Returns:

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

seed.views.main.**search_building_snapshots**()

> **URI** /app/search_building_snapshots/

Retrieves a paginated list of BuildingSnapshots matching search params.

Payload:

```
{
 'q': a string to search on (optional),
 '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.
   }
 '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': [
  {
   '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.**search_buildings**()

> **URI** /app/search_buildings/

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.**start_mapping**()

> **URI** /app/start_mapping/

Starts a background task to convert imported raw data into BuildingSnapshots, using user's column mappings.

Payload:

```
{'file_id': The ID of the ImportFile to be mapped}
```

Returns:

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

seed.views.main.**start_system_matching**()

> URI /app/start_system_matching/

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

Payload:

```
{'file_id': The ID of the ImportFile to be matched}
```

Returns:

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

seed.views.main.**update_building**()

> URI /app/update_building/

Manually updates a building's record. Creates a new BuildingSnapshot for the resulting changes.

> PUT { 'organization_id': organization id, 'building':
>
> > { 'canonical_building': The canonical building ID 'fieldname': 'value'... The rest of the fields in the
> >
> > > BuildingSnapshot; see get_columns() endpoint for complete list.
> >
> > }
>
> }

Returns:

```
{'status': 'success',
 'child_id': The ID of the newly-created BuildingSnapshot
}
```

seed.views.main.**update_dataset**()

> URI /app/update_dataset/

Updates the name of a dataset.

Payload:

```
{'dataset':
    {'id': The ID of the Import Record,
     'name': The new name for the ImportRecord
    }
}
```

Returns:

```
{'status': 'success' or 'error',
 'message': 'error message, if any'
}
```

Data Model

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 primary BuildingSnapshot, BS0, and the secondary BuildingSnapshot, BS1, merged together. If both the primary and secondary BuildingSnapshot have data for a given field, the primary's fields are preferred and merged into the child, B3.

All BuildingSnapshot instances point to a CanonicalBuilding.

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

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

7.1 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 database as a many-to-many relation from BuildingSnapshot to BuildingSnapshot. 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_building_snapshots` 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
```

7.1.1 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	**14**		14
id4	**13**	14	13

7.2 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)
```
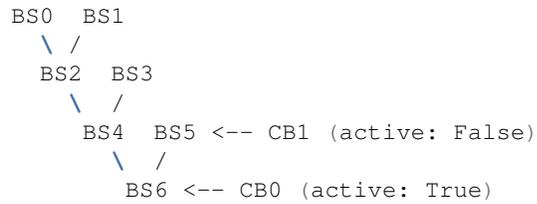
Even though BS5 is pointed to by a CanonicalBuilding, CB1, BS5 will not be returned by the normal `search_buildings` endpoint because the CanonicalBuilding pointing to it has its field `active` set to `False`.

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

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

8.1 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 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*.

8.2 Mapping

After the data is saved, the UI invokes *seed.views.main.get_column_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.

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

Updating this documentation

This python code documentation was generated by running the following:

```
$ pip install Sphinx==1.2.2
$ sphinx-apidoc -o docs/source/  .
$ cd docs
$ make html
```

Indices and tables

- *genindex*
- *modindex*
- *search*

U