OpenBlock Documentation

Release 1.1.0

The OpenBlock Team

August 26, 2015

Contents

OpenBlock is a web application that allows users to browse and search their local area for “hyper-local news” - to see what’s going on recently in the immediate geographic area.

CHAPTER 1

Brief Overview: Concepts and Terminology

OpenBlock is a *hyperlocal news platform*. What we mean by that is that, at its essence, OpenBlock is a web application (and web service) that stores two kinds of information:

- *Local news*. What's happening nearby? This could be your original content, or aggregated from any number of sources on the web.
- *Local geographic data*. What places do we care about? Neighborhoods, zip codes, school districts, police precincts?

OpenBlock allows you to explore that data in various ways: by geographic area, small or large; by type of news; by various categories relevant to the types of news you have; by text search; or by any combination of the above.

News in OpenBlock is stored as *NewsItems*. In essence, a NewsItem is just something that happened at one *time* and one *place*. Each NewsItem stores a timestamp, a geographic point, a title, a description, and a few other generic fields.

Each NewsItem also has a type, which we call its *Schema*. Schemas are used to classify NewsItems and allow them to have extra searchable, type-specific data.

For example: an OpenBlock site might provide both police reports and restaurant inspections. There would be one *Schema* representing police reports, allowing each police report to store information about what kinds of crime were committed. There would be a second *Schema* representing restaurant inspections, and it would allow each inspection report to say whether the restaurant failed or passed, what violations were observed, and so on.

All this can be configured by the site administrator without writing code.

A user is then able to browse *NewsItems* that are only police reports, or only restaurant inspections, or both; or browse only failed restaurant inspections; or browse crimes of a certain type, in a certain location, during a certain time period; et cetera.

There is a demo site where you can experiment with similar searches.

CHAPTER 2

History

OpenBlock began life as the open-source code released by Everyblock.com in June 2009. Originally created by Adrian Holovaty and the Everyblock team in 2007, it has been rebranded OpenBlock to avoid trademark infringement, and is now developed as an open-source (GPL) project at http://openblockproject.org.

Funding for both the initial creation of Everyblock and the ongoing development of OpenBlock was provided by the Knight Foundation.

CHAPTER 3

Copyright / License

OpenBlock code is licensed under the GNU General Public License version 3.

A few modules were borrowed from other packages with other free software licenses, e.g. “Modified BSD”-style licenses; these are identified as such in the source code.

This documentation is licensed under the Creative Commons BY-SA 3.0 license.

For Developers

This is a Django application, so it’s highly recommended that you have familiarity with the Django Web framework. The best places to learn are the official Django documentation and the free Django Book.

Before you dive in, it’s *highly* recommended you spend a little bit of time browsing around http://demo.openblockproject.org and/or http://EveryBlock.com to get a feel for what this software does. (But note that the code has diverged considerably since 2009, so everyblock.com has features not present in OpenBlock, and vice versa; the visual design is quite different as well.)

Also, for a light conceptual background on some of this, particularly the data storage aspect of ebpub, watch the video “Behind the scenes of EveryBlock.com” here: http://blip.tv/file/1957362

CHAPTER 5

All Contents

5.1 Installation and Setup

5.1.1 Installing OpenBlock on Amazon AWS

This is the fastest way to try out OpenBlock. You can launch an instance cloned from our AMI and start feeding in data in minutes.

Warning, experimental!

Our AMI is still new and hasn't been widely tested. Please report any issue to the mailing list.

Get an Account

If you haven't registered for Amazon AWS, you might try out a free account. OpenBlock will happily run on the smallest (micro) EC2 instance size.

Launch an Instance ...

In the AWS EC2 management console, click "Launch Instance". Click "Community AMIs", and in the search box, type in "ami-69d91600". (You can also try searching for "openblock".)

Select the OpenBlock AMI, then continue through the wizard until your instance is launched. You can leave all options at their default values (except the security group) unless you know what you're doing.

Security groups

Note that the default security group has *all* ports closed. You can modify your default security group, or create a new one; either way you *must* add the SSH and HTTP rules to open ports 22 and 80. Otherwise you won't be able to connect to your new instance at all.

As part of the setup, you'll be prompted to create a key pair (if you don't have one already). It's very important that you save the PEM file that it prompts you to download. You'll use this to ssh to your server later.

Or if you prefer, you can use Amazon's `ec2-run-instances` command-line tool, which is beyond the scope of this document.

Don’t forget to stop it!

Remember that AWS bills by the hour. Especially if you’re not on the free plan, be sure to stop or terminate any instances you’re not using when done with them.

Does It Work?

If all goes well, you will be immediately able to browse OpenBlock on your new instance. In the AWS EC2 console, click on your running instance, and look for its “*Public DNS*”. This will be a domain name such as “ec2-50-17-54-xyz.compute-1.amazonaws.com”. Paste or type that into your browser, and you should see the OpenBlock front page.

If the connection times out, make sure your security group allows connecting to port 80.

Until you finish configuration and data loading, your site will have some boring defaults, such as the title “OpenBlock: Your City”, and there will be no locations and no news.

What You Have

- Openblock 1.1. A checkout of the stable branch is installed in a virtualenv at */home/openblock/openblock*.
- A “custom” app named “myblock” as per the docs, installed at `/home/openblock/openblock/src/myblock/myblock`
- Ubuntu 11.04 (“Natty”), Python 2.7, Postgresql 8.4 and PostGIS 1.5, Apache2, mod_wsgi.

The code and its database are set up as if you had already followed the Preparing Your System, Installing the Openblock Software, and Creating a Custom Site Based on OpenBlock instructions.

A few other nice details are taken care of for you:

- `cron` jobs are configured in `/etc/cron.d/openblock`. Notably, this cron config has some commented-out examples of running scraper scripts. It also periodically runs any *Background Tasks*. It also sends the alerts email messages.
- `logrotate` is already configured to rotate the apache and openblock logs, so they won’t fill up your storage.

Get ssh access

Next you’ll need to log in to your server to do some configuration. The username will be `ubuntu` and you’ll need to use the PEM file that you were prompted to save when you launched your instance, and your public DNS that you can find in the EC2 console.

If you have a command-line `ssh` tool such as openssh, you can log in like so:

```
$ ssh -i <PATH TO YOUR PEM FILE> ubuntu@<YOUR PUBLIC DNS HERE>
```

If you’re using another ssh tool such as PuTTY, try searching the web for instructions on how to use it with AWS.

Once you’re in...

You’ll be logged in as the `ubuntu` user, but openblock is installed by the `openblock` user. So typically the first thing you will do is run these commands:

```
$ sudo su - openblock
$ cd /home/openblock/openblock
$ source bin/activate
$ export DJANGO_SETTINGS_MODULE=myblock.settings
```

Users and Permissions on Your EC2 Instance

Note that the `openblock` user can do most anything that needs doing in its home directory, but has no password and has limited privileges beyond that, eg. cannot use `sudo`. I often keep a second terminal logged in as `ubuntu` for those times that I need to use `sudo`.

Change Settings

The OpenBlock config file will be at `/home/openblock/openblock/src/myblock/myblock/settings.py`. Edit that file as per Configuring OpenBlock.

(Text editors *nano* and *vim* are installed; you can of course install *emacs* or whatever else you like.)

Security warning: it is especially important that you change the `PASSWORD_CREATE_SALT` and `PASSWORD_RESET_SALT` settings.

Note that anytime you change settings, or updater your openblock code, you'll want to run this command before you can see your changes take effect on your site:

```
$  touch /home/openblock/openblock/wsgi/myblock.wsgi
```

Warning about email!

OpenBlock uses outgoing email for two features: account registration, and email alert subscriptions. **You can't really send email from an EC2 host.** Due to spam concerns, Amazon strictly limits the amount of email you can send, and many ISPs block email from EC2 anyway. The solution is to use another email server to send your outgoing email. If you don't have an SMTP server available, you may be able to use a gmail account or similar; for example, see this blog post. Or you might try Amazon's own email service: https://aws.amazon.com/ses/

Make an Admin User

Your instance does not come with an administrative django user, because of course we don't want other people who clone the AMI to know your password. You can create one with this command:

```
$ django-admin.py createsuperuser
```

Now you can log in at `http://<your public DNS>/admin`.

What's Next

You'll want to start Loading Geographic Data.

Then you'll want to add some custom content types to your site, and write some scraper scripts to populate them.

5.1.2 Preparing Your System

These instructions cover manual installation of the prerequisites for OpenBlock and installation of the base Openblock software.

(You can skip this if you are cloning our AMI image.)

If you have problems...

Please drop a line to the ebcode google group or visit the IRC channel `#openblock` on freenode with any problems you encounter. We're glad to help.

System Requirements

Note: See System Specific Prerequisites for specific lists of packages to install based on your distribution or OS.

Linux, OSX, or some other Unix flavor is required. **Windows is not supported** by the OpenBlock team, and may never be; but patches are welcome :)

Generally, you need:

- python 2.6 (2.7 might work; 2.5 is too old)
- Postgresql 8.3, 8.4, or 9.0
- PostGIS 1.4 or 1.5
- libxml2 and libxslt
- libgdal
- git
- subversion
- wget
- unzip
- virtualenv

Optionally, it may be helpful to install prebuilt packages for the following if your distribution provides them:

- python lxml bindings
- python gdal bindings

GeoDjango's platform-specific instructions may have some useful information as well, as the majority of the requirements are just those of GeoDjango + PostGIS.

Don't forget ldconfig!

Typically after installing libraries, you will need to run this command:

```
$ sudo ldconfig
```

... in order for new libraries to be found while building software.

Database Setup

GeoDjango requires a spatial database; more specifically, OpenBlock requires PostGIS. This documentation generally assumes you are installing OpenBlock and Postgres on the same server. If you are using a remote server, please read `remote_postgis_server` and make adjustments accordingly.

OpenBlock is known to work with Postgresql 8.3, 8.4, or 9.0, and PostGIS 1.4 or 1.5.

PostGIS template setup

Regardless of whether you run postgresql locally or on another host, you'll want a PostGIS template database. Some platforms install this automatically for you, some don't.

You (or your database admin) should follow the instructions for Creating a Spatial Database Template for PostGIS in the GeoDjango documentation and be sure to heed the **Note** about varying names and locations of the relevant files.

Database Access Settings

The following instructions (and the default settings) assume that there is an `openblock` database user which can create a database for use with openblock. You can create an openblock user by running:

```
$ sudo -u postgres createuser --createrole --createdb openblock
```

Depending on your database security setup, you may need to adjust the instructions, settings of postgres and/or settings of openblock.

Postgres administration is beyond the scope of these instructions, but as a quickstart, you can disable postgres security for local users by changing the `pg_hba.conf` file under `etc` (the precise location varies, but for postgresql 8.4 on Ubuntu it's `/etc/postgresql/8.4/main/pg_hba.conf`), comment out any line that starts with `local all`, and add a line like this:

```
local    all     all   trust
```

Then restart postgresql. **This is not suitable for production**.

See Postgres pg_hba.conf documentation or the postgres wiki for more information.

Testing Database Access If the `openblock` user is configured correctly, you should be able to execute:

```
$ createdb -U openblock test_ob_access
$ dropdb -U openblock test_ob_access
```

Next Steps

Now that your system is prepped, you are ready to move on to Installing the Openblock Software.

5.1.3 Installing the Openblock Software

These steps assume you have fulfilled the requirements and followed the instructions in the section Preparing Your System.

(You can skip this if you are cloning an OpenBlock AMI.)

Creating a virtualenv

Create a "virtualenv" that will contain the OpenBlock software and its python dependencies. (You probably do *not* want to do this as root or with sudo):

```
$ virtualenv openblock
$ cd openblock
```

"Activate" your virtualenv - this makes sure that all python commands will use your new virtual environment:

```
$ source bin/activate
```

Activating also sets the $VIRTUAL_ENV environment variable, which we can use as a convenient base to be sure that we run commands in the right directory.

We'll be using pip to install some software, so make sure it's installed. Recent versions of virtualenv do this for you, but virtualenv < 1.4.1 does not, so we need to make sure. We also recommend that you ensure that the latest versions of pip and distribute are installed:

```
$ easy_install --upgrade pip distribute
$ hash -r
```

Note that it's *very* important that pip is installed *in the virtualenv*. If you only have pip installed globally on your system, *it won't work* and you will get confusing build errors such as version conflicts, permission failures, etc.

Installing OpenBlock Packages

You can install either stable releases of the OpenBlock software, or check out the latest development code.

Installing Stable Packages

The latest stable releases of ebpub, ebdata, and obadmin can be found on the Python Package Index. To install from these packages, we will publish a consolidated pip requirements file that will install *all* the necessary python packages. These requirements files will be listed for each release at http://openplans.github.com/openblock/ .

For example, the 1.1 release is at: http://openplans.github.com/openblock/requirements/openblock-requirements-1.1.0.txt and can be installed with this command:

```
$ $VIRTUAL_ENV/bin/pip install -r http://openplans.github.com/openblock/requirements/openblock-requi
```

If you encounter errors during package installation, please see Common Installation Problems.

Installing Development Code

Download the openblock software:

```
$ cd $VIRTUAL_ENV
$ mkdir -p src/
$ git clone git://github.com/openplans/openblock.git src/openblock
```

It takes a few more Pip commands to install for development, like so commands:

```
$ cd $VIRTUAL_ENV/src/openblock
$ pip install -r ebpub/requirements.txt
$ pip install -e ebpub
$ pip install -r ebdata/requirements.txt
$ pip install -e ebdata
$ pip install -r obadmin/requirements.txt
$ pip install -e obadmin
```

If you encounter errors during package installation, please see Common Installation Problems.

Next Steps: Install the Demo, or Create a Custom App

If you want to run the OpenBlock demo app (just like http://demo.openblockproject.org), proceed with *Step-By-Step Demo Installation*.

Or, you can dive right in to Creating a Custom Site Based on OpenBlock.

5.1.4 Common Installation Problems

This page covers some common installation problems.

Please feel free to drop a line to the ebcode google group or visit the IRC channel `#openblock` on freenode with any problems you encounter. We're glad to help.

Python Package Conflicts / Failures

Some quick things to check if you have any problems installing any of the Python package dependencies:

- Make sure your virtualenv is activated. `$VIRTUAL_ENV` should point to the right directory.
- You should have at least version 1.0 of pip. Check `pip --version`
- Make sure pip is installed *in the virtualenv*. Check the output of `which pip`.
- You should have a recent version of distribute. Try `easy_install --version`. If it says at least 'distribute 0.6.14', you're OK.
- Don't try to combine the `pip install -r` and `pip install -e` commands. Doing so can result in the wrong version of a dependency. (This is a pip bug.)

Problems Installing lxml

Installing the easy way

It's easiest to install your platform's package for lxml globally, if it has one. For example, on ubuntu:

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

(Note that if you want to take this approach, you *must not* run virtualenv with the `--no-site-packages` option, as that will prevent your virtualenv from being able to use this package.)

The slightly harder way

If your platform doesn't have a ready-made lxml package, or if you prefer to build your own, you'll need the libxml2 and libxslt development libraries, and then install lxml yourself. For example, on ubuntu you can do:

```
$ sudo apt-get install libxml2 libxml2-dev libxslt libxslt-dev
```

And once you have those, on any platform you can do:

```
$ sudo ldconfig
$ sudo pip install lxml
```

Problems Installing GDAL

Installing the easy way

GDAL installation isn't covered in detail by the GeoDjango install docs.

The easiest thing to do is check if your operating system already provides a ready-made python GDAL package. For example, on Ubuntu, this will work:

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

(Note that if you want to take this approach, you *must not* run virtualenv with the `--no-site-packages` option, as that will prevent your virtualenv from being able to use this package.)

GDAL the hard way

TODO: see if we can contribute this upstream?

Installing GDAL by hand can be a little tricky, because you have to be careful about which version you install, and in some cases it may not install properly without a few extra arguments.

First, get the GDAL development library. On Ubuntu, this can be installed like:

```
$ sudo apt-get install libgdal libdal1-dev
$ sudo ldconfig
```

Next, make sure you are in your openblock environment and it is activated:

```
$ cd <path_to_openblock>
$ source bin/activate
```

Next, determine which version of the Python GDAL package you need. Try this command:

```
$ gdal-config --version
```

The output will be a version number like "1.6.3". Your Python GDAL package version number needs to match the first two digits. So if `gdal-config --version` tells you "1.6.3", then you would need a version of Python GDAL that's at least 1.6.0, but less than 1.7. Or if gdal-config tells you that you have 1.7.0, then you would need version 1.7.something of the Python GDAL package. You get the idea. You can use `pip` to find an appropriate version, like this:

```
$ pip install --no-install "GDAL>=1.6,<1.7a"  # adjust version as needed
```

Next, remove the bogus setup.cfg file, if any:

```
$ rm -f $VIRTUAL_ENV/build/GDAL/setup.cfg
```

Build the python package with some extra options, determined as described below:

```
$ cd $VIRTUAL_ENV/build/GDAL
$ python setup.py build_ext --gdal-config=gdal-config \
    --library-dirs=/usr/lib \
    --libraries=gdal1.6.0 \
    --include-dirs=/usr/include/gdal \
  install
```

The correct value for –library-dirs can be determined by running `gdal-config --libs` and looking for any output starting with `-L`. The correct value for –libraries can be determined with the same command but looking for output beginning with -l. The correct value for --include-dirs can be determined by running gdal-config --cflags and looking for output beginning with -I.

Still no luck?

If you get an error like /usr/include/gdal/ogr_p.h:94: fatal error: swq.h: No such file or directory, that's because of a bug in GDAL. (See http://trac.osgeo.org/gdal/ticket/3468 .)

The workaround is to manually install swq.h in the same directory that contains ogr_p.h, typically somewhere like /usr/include/gdal. You can get swq.h for GDAL 1.7 here: http://svn.osgeo.org/gdal/branches/1.7/gdal/ogr/swq.h

Then try the preceding setup.py build_ext command again.

5.1.5 Installing and Setting Up the Demo Site

These instructions will install the software in a similar configuration to the OpenBlock demo site. To start a custom application instead, please see Creating a Custom Site Based on OpenBlock.

(You can skip this if you are cloning an OpenBlock AMI.)

Step-By-Step Demo Installation

Basic Setup

First, follow **all** the instructions in the Preparing Your System document and Installing the Openblock Software

If you followed the Installing the Openblock Software instructions properly, you've already got a virtualenv ready. Go into it and activate it, if you haven't yet:

```
$ cd path/to/your/virtualenv
$ source bin/activate
```

Installing obdemo - stable release

obdemo is included as part of *Installing Stable Packages*.

You'll then want to make a copy of the skeleton settings file for editing, which lives at $VIRTUAL_ENV/lib/python2.*/site-packages/obdemo/settings.py.in.

Installing obdemo for development

You can work on the latest development code of obdemo and its dependencies like this, assuming you've already taken care of *Installing Development Code*:

```
$ cd $VIRTUAL_ENV/src/openblock
$ pip install -r obdemo/requirements.txt
$ pip install -e obdemo
```

Editing Settings

You'll want to edit the demo's django settings at this point, or at least look at it to get an idea of what can be configured. There is also some configuration documentation you should look at.

`obdemo` doesn't come with a settings.py; it comes with a `settings.py.in` template that you can copy and edit.

(If you've installed OpenBlock sources from git, this file will be at `$VIRTUAL_ENV/src/openblock/obdemo/obdemo/settin`
If you've installed a stable release from pypi, it will be at `$VIRTUAL_ENV/lib/python2.*/site-packages/obdemo/setti`

```
$ cd path/to/obdemo
$ cp settings.py.in settings.py
$ favorite_editor settings.py
```

At minimum, you should change the values of:

- `PASSWORD_CREATE_SALT` - this is used when users create a new account.
- `PASSWORD_RESET_SALT` - this is used when users reset their passwords.
- `STAFF_COOKIE_VALUE` - this is used for allowing staff members to see some parts of the site that other users cannot, such as types of news items that you're still working on.

You'll also want to think about *Choosing Your Map Base Layer*.

Database Initialization

You should already have taken care of *Database Setup*. Create the (empty) database with this command:

```
$ sudo -u postgres createdb -U openblock --template template_postgis openblock
```

Now initialize your database tables:

```
$ export DJANGO_SETTINGS_MODULE=obdemo.settings
$ django-admin.py syncdb --migrate
```

(The –migrate option is important; it loads some initial data that openblock depends on including stored procedures, and some default Schemas that you can try out, modify, and delete as needed.)

This will also prompt you to create an administrative user, which is a good idea.

Starting the Test Server

Run these commands to start the test server:

```
$ export DJANGO_SETTINGS_MODULE=obdemo.settings
$ django-admin.py runserver
  ...
  Development server is running at http://127.0.0.1:8000/
```

You can now visit http://127.0.0.1:8000/ in your Web browser to see the site in action (with no data). You can log in to view the administrative site at http://127.0.0.1:8000/admin/ .

Loading Demo Data

OpenBlock is pretty boring without data! You'll want to load some geographic data and some local news. We've included some example data for Boston, MA, and scraper scripts you can use to start with if you don't have all of your local data on hand yet.

Set your DJANGO_SETTINGS_MODULE environment variable before you begin:

```
$ export DJANGO_SETTINGS_MODULE=obdemo.settings
```

First you'll want to load Boston geographies. This will take several minutes:

```
$ django-admin.py import_boston_zips
$ django-admin.py import_boston_hoods
$ django-admin.py import_boston_blocks
```

Then fetch some news from the web, this will take several minutes:

```
$ django-admin.py import_boston_news
```

For testing with random data you might also want to get the `misc` directory from the OpenBlock source code, and try the `random-news` script like so:

```
$ src/openblock/misc/bin/random_news.py 10 local-news
```

... where 10 is the number of random articles to generate, and 'local-news' is a Schema slug. You must first have some blocks in the database; it will assign randomly generated local news articles to randomly chosen blocks.

Next Steps

Now that you have the demo running, you might want to add some more custom content types to it, and write some scraper scripts to populate them.

5.1.6 Creating a Custom Site Based on OpenBlock

If you want to do something much different than obdemo, you're better off starting from scratch with a custom Django project. We provide a script that will get you started with a skeleton app that you can edit.

(You can skip this if you are cloning an OpenBlock AMI.)

Setting up the app

Basic Setup

First, follow **all** the instructions in the Preparing Your System document and Installing the Openblock Software

If you followed the Installing the Openblock Software instructions properly, you've already got a virtualenv ready. Go into it and activate it, if you haven't yet:

```
$ cd path/to/your/virtualenv
$ source bin/activate
```

Create Custom App Package

Now do the following to create a new OpenBlock project. **Note**: Your project name should be suitable for use as a python module name; i.e. no spaces etc. Here we assume the project name is *myblock*:

```
$ cd $VIRTUAL_ENV/src
$ paster create -t openblock myblock
```

After answering a few questions, this will create a bare-bones Django project in the folder you specified. Next, install the project into your environment:

```
$ cd myblock
$ python setup.py develop
...
```

What You Get

The created package is a minimal Django project that includes:

- `settings.py` you can *edit*.
- `urls.py` that wraps ebpub's URLs; you can override individual views here, add custom views and other Django apps, etc.
- `manage.sh`, a tiny wrapper around `manage.py` / `django-admin.py` that saves you the trouble of activating your virtualenv or exporting `DJANGO_SETTINGS_MODULE`.
- `templates/homepage.html`, an example of overriding one of `ebpub`'s templates, edit as you like.
- `wsgi/<projectname>.wsgi`, suitable for deploying your project under Apache and mod_wsgi. It takes care of finding the containing virtualenv and the `DJANGO_SETTINGS_MODULE` automatically.

Adjust Django Settings

Your django settings are located in settings.py within your project. You should review these and make adjustments based on your setup:

```
$ favorite_editor myblock/settings.py
...
```

Read more about important settings you can/should customize.

If you plan to use a remote database or have other changes to database connection information, make sure you change them in your settings.py. See *Database Access Settings* and be sure everything works before you proceed.

Create and Initialize the Database

Now, as usual with Django projects, you'll need to create and initialize your database. If you haven't changed the default database settings, and if you've followed the *PostGIS template setup* instructions, then the database creation command would simply be:

```
$ sudo -u postgres createdb -U openblock --template template_postgis openblock_myblock
```

If you have a different postgresql setup, for example you're using a different user than `openblock`, just change the -U option accordingly.

Now initialize your database tables:

```
$ export DJANGO_SETTINGS_MODULE=myblock.settings
$ django-admin.py syncdb --migrate
```

(The –migrate option is important; it loads some initial data that openblock depends on including stored procedures, and some default Schemas that you can try out, modify, and delete as needed.)

This will also prompt you to create an administrative user, which is a good idea.

Starting the Test Server

Run django's test server using your project's settings and visit http://127.0.0.1:8000/ in your Web browser to see the site in action (with no data):

```
$ export DJANGO_SETTINGS_MODULE=myblock.settings
$ django-admin.py runserver
...
Development server is running at http://127.0.0.1:8000/
```

You can now visit http://127.0.0.1:8000/ in your Web browser to see the site in action (with no data). You can log in to view the administrative site at http://127.0.0.1:8000/admin/ .

Loading Data: Things You Will Need

To get anything useful out of your site, at minimum you will need the following:

1. Geographic data for your area. See Loading Geographic Data.
2. Sources of news data to feed in.
   (a) Configure the system with schemas for them. See Creating a Custom NewsItem Schema and ebpub docs for *NewsItems and Schemas*.
   (b) Write scraper scripts to retrieve news from your news sources and load it into the database. See the Data Scraper Tutorial, ebdata and http://developer.openblockproject.org/wiki/ScraperScripts .
3. Optionally, customize the look and feel of the site. See the ebpub docs for *Site views/templates*.

Gathering all this data and feeding it into the database can be a bit of work at this point. The `obdemo/bin/bootstrap_demo.sh` script in the openblock source code does all this for the demo site with Boston data, by calling other scripts; together, they should serve as a decent example of how to do things in detail.

If you want to load the demo data into your project, you can use the steps listed in *Loading Demo Data*. **Note**: use the settings module for your project instead of *obdemo.settings*.

Additional Resources

For more documentation (in progress), see also:

- http://developer.openblockproject.org/wiki/Data
- http://developer.openblockproject.org/wiki/Ideal%20Feed%20Formats

5.1.7 Configuring OpenBlock

You should have a look at `ebpub/ebpub/settings_default.py`. It contains many comments about the purpose and possible values of the various settings expected by OpenBlock.

A few items are worth special mention.

Sensitive Settings

These are settings you *must* customize, and avoid putting in a public place eg. a public version control system.

- `PASSWORD_CREATE_SALT` - this is used when users create a new account.

- PASSWORD_RESET_SALT - this is used when users reset their passwords.
- STAFF_COOKIE_NAME and STAFF_COOKIE_VALUE - this is used for allowing staff members to see some parts of the site that other users cannot, such as types of news items that you're still working on.

Choosing Your Map Base Layer

If you don't like the look of OpenBlock's default maps, you have many options for your *base layer* - the tiled images that give your map its street lines, geographic features, place names, etc.

Default: OpenStreetMap tiles hosted by OpenGeo

This is the default setting in ebpub/ebpub/settings_default.py. It is a fairly clean design inspired by everyblock.com, and was derived from OpenStreetMap data. It is free for use for any purpose, but note that there have been some reliability issues with this server in the past.

Other Publicly Available Layers

It's easy to use any base layer supported by olwidget. More specifically:

Google Maps If your intended usage on your website meets Google's Terms of Service, or if you have a Premier account, you may be able to use Google Maps for your base layer.

In your settings.py, set MAP_BASELAYER_TYPE to any of 'google.streets', 'google.physical', 'google.satellite', or 'google.hybrid'. Then be sure to get an API key from Google and put it in your settings file as GOOGLE_API_KEY.

Open Street Map Set MAP_BASELAYER_TYPE to either 'osm.mapnik' or 'osm.osmarender'.

Microsoft VirtualEarth / Bing Maps Set MAP_BASELAYER_TYPE to any of 've.road', 've.shaded', 've.aerial', or 've.hybrid'.

Yahoo Maps Set MAP_BASELAYER_TYPE = 'yahoo' and be sure to set YAHOO_APP_ID to your Yahoo app id.

Other public WMS servers Set MAP_BASELAYER_TYPE to either 'wms.map' (not very useful for OpenBlock) or 'wms.nasa'.

CloudMade Cloudmade hosts a *lot* of community-designed map base layers. You can even design your own online using their tools.

Get an API key from them and put it in your settings as CLOUDMADE_API_KEY. Then set MAP_BASELAYER_TYPE = 'cloudmade.<num>' (where <num> is the number for a cloudmade style). For example, 'cloudmade.998'.

To find interesting cloudmade style numbers, browse at http://maps.cloudmade.com/editor ; the style number is at bottom right of each style.

Blank (no base layer) Try MAP_BASELAYER_TYPE = 'wms.blank'

Custom or Other Base Layers

Do you have your own tile server running, or have a URL to something else not in the above list? Great! You can use that with a few extra settings. This option takes a little more work; you will have to know which OpenLayers layer subclass is appropriate, and what parameters to pass to it.

In fact, this is how our default OpenGeo / OpenStreetMap layer is configured, so let's use that as an example:

```
MAP_BASELAYER_TYPE = 'custom.opengeo_osm'

MAP_CUSTOM_BASE_LAYERS = {
   'opengeo_osm':  # to use this, set MAP_BASELAYER_TYPE='custom.opengeo_osm'
       {"class": "WMS",  # The OpenLayers.Layer subclass to use.
        "args": [  # These are passed as arguments to the constructor.
           "OpenStreetMap (OpenGeo)",
           "http://maps.opengeo.org/geowebcache/service/wms",
           {"layers": "openstreetmap",
            "format": "image/png",
            "bgcolor": "#A1BDC4",
            },
           {"wrapDateLine": True
            },
           ],
        }
}
```

Multiple databases?

Note that while Django supports using multiple databases for different model data, OpenBlock does not. This is because we use South to automate *database migrations*, and as of this writing South does not work properly with a multi-database configuration.

Configuring Cities / Towns: METRO_LIST

If you look at `obdemo/obdemo/settings.py.in`, or at the `settings.py` that is generated when you start a custom app, you will notice it contains a list named `METRO_LIST`.

This list will (almost) always contain only one item, a dictionary with configuration about your local region.

Most of the items in this dictionary are fairly self explanatory. Here's an example for Boston:

```
  METRO_LIST = [
   {
       # Extent of the metro, as a longitude/latitude bounding box.
       'extent': (-71.191153, 42.227865, -70.986487, 42.396978),

       # Whether this area should be displayed to the public.
       'is_public': True,

       # Set this to True if the region has multiple cities.
       'multiple_cities': False,

       # The major city in the region.
       'city_name': 'Boston',

       # The SHORT_NAME in the settings file.
       'short_name': SHORT_NAME,
       # The name of the metro or region, as opposed to the city (e.g., "Miami-Dade" instead of "Mia
       'metro_name': 'Boston',

       # USPS abbreviation for the state.
       'state': 'MA',

       # Full name of state.
       'state_name': 'Massachusetts',

       # Time zone, as required by Django's TIME_ZONE setting.
       'time_zone': 'America/New_York',

       # Only needed if multiple_cities = True.
       'city_location_type': 'city',

   },
]
```

More information on a few of these follows.

short_name

This is how OpenBlock knows which dictionary in `METRO_LIST` to use. It must exactly match the value of `settings.SHORT_NAME`.

extent

This is a list of (leftmost longitude, lower latitude, rightmost longitude, upper latitude).

One way to find these coordinates would be to use Google Maps to zoom to your region, then point at the lower left corner of your area, right-click, and select “Drop LatLng Marker”. You will see a marker that displays the latitude,longitude of that point on the map. Then do the same in the upper right corner.

This defines a bounding box - the range of latitudes and longitudes that are relevant to your area. It is used in many views as the default bounding box when searching for relevant NewsItems. It is also used by some data-loading scripts to filter out data that’s not relevant to your area.

multiple_cities

Set `multiple_cities` to `True` if you want one OpenBlock site to serve multiple cities or towns in the same region.

For example, you might be setting it up for a county. In this example you could use the county name for `city_name` and `metro_name`. Or you might be somewhere like the San Francisco Bay Area and wanting to include San Francisco, Oakland, Berkeley, and so on. So `city_name` might be ‘San Francisco’ and `metro_name` might be something like ‘Bay Area’.

If `multiple_cities` is True, you must also set `city_location_type`, see below.

This option affects numerous URLs on the site; users will be able to browse first by city, then by street, then by block, and so on. If it’s `False`, the city browsing page will be left out of the site structure.

city_location_type

You only need this if `multiple_cities` is True. In that case you will need to create a *LocationType* for cities, and `city_location_type` should be set to that `LocationType`‘s slug.

You will then want to create a `Location` for each city in your region. See *Loading Location Data* for more.

When would you put more than one dictionary in METRO_LIST?

The only dictionary in `METRO_LIST` that has any effect is the one whose `short_name` matches `settings.SHORT_NAME`.

The purpose of having more than one metro dictionary in `METRO_LIST` would be to run multiple OpenBlock sites for multiple metro areas with some shared configuration. *You are probably not doing this.*

The idea is that you could have one settings file containing the master `METRO_LIST`, and then for each site you'd have its own settings file that imports `METRO_LIST` (and any other shared stuff you like) from the master settings file. Each site-specific settings file would also set `settings.SHORT_NAME` to match the `’short_name’` key of one of the dictionaries.

Most people will probably not be doing that. This feature serves the needs of everyblock.com, which runs separate sites for many cities across the USA.

Email

OpenBlock uses email for two things: account confirmation, and alerts to which users can subscribe in order to get notified when news happens in their neighborhood or other area of interest.

OpenBlock is configured like `any other Django application`. In your `settings.py`, you'll want to set these:

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST='localhost'
EMAIL_PORT='25'
# If your email host needs authentication, set these.
#EMAIL_HOST_USER=''
#EMAIL_HOST_PASSWORD=''
#EMAIL_USE_TLS=False  # For secure SMTP connections.
# This is used as "From:" in emails sent to users.
GENERIC_EMAIL_SENDER = 'admin@example.com'
```

Don't have an SMTP Server?

You may be able to use an appropriate account on Gmail or another public mail service. See for example this blog post).

Email on AWS EC2

If you are installing on amazon's EC2 servers, note that you must use a different server to send mail, as Amazon limits the amount of mail you can send, and most ISPs will block it as likely spam anyway. So use another service such as Gmail as per the previous paragraph, or you might try Amazon's own email service: https://aws.amazon.com/ses/

OpenBlock REST API

MAX_KEYS_PER_USER – how many API keys each OpenBlock user can register. Default 1.

API_THROTTLE_TIMEFRAME, API_THROTTLE_AT – Together these control how many API requests a user or API key can make in certain period of time. If the user makes more than API_THROTTLE_AT requests within a period of API_THROTTLE_TIMEFRAME seconds, then all further requests will be denied until another API_THROTTLE_TIMEFRAME seconds have passed.

API_THROTTLE_EXPIRATION – How long to keep track of last access times per user. This is just for housekeeping, in practice it doesn't affect your users.

Enable caching too!

In order to enable throttling, you MUST also configure CACHES['default'] to something other than a DummyCache, as per the DJango caching documentation.

Django-Static

OpenBlock currently uses Django-Static to manage static media such as Javascript and CSS files. The advantage over Django's built-in "StaticFiles" app is that Django-Static automatically handles timestamping media URLs and minify-ing scripts. With eg. a suitable *Apache config*, you can safely set far-future expiration dates and never have stale scripts.

The relevant settings are DJANGO_STATIC, DJANGO_STATIC_MEDIA_ROOTS, DJANGO_STATIC_NAME_PREFIX, DJANGO_STATIC_SAVE_PREFIX. All have sensible defaults in ebpub/settings_default.py. If you need to override them, see the README.

Note there are some exceptions: we don't use django-static for either JQuery or OpenLayers because you might want to use hosted versions of those, and django-static probably isn't the best way to minify large frameworks anyway.

Django Background Tasks

For long-running jobs, we currently use django-background-task. This is currently used only by some data loading pages in the admin UI. The relevant settings are MAX_RUN_TIME and MAX_ATTEMPTS. See the README for more information.

Miscellaneous Settings

AUTH_PROFILE_MODULE – A module and class name to use for user profile data. Default is "preferences.Profile", you can override this if you want to do something custom, but may require diving in to the code to understand what assumptions we make about profiles.

DEFAULT_DAYS – How many days of news to show on many views.

DEFAULT_LOCTYPE_SLUG – Which LocationType to show on the /locations page. Once you've *created some LocationTypes*, this should be set to the slug of your preferred LocationType.

DEFAULT_MAP_CENTER_LAT, DEFAULT_MAP_CENTER_LON, DEFAULT_MAP_ZOOM – Where to center city-wide maps by default, eg. on the home page.

EBPUB_CACHE_GEOCODER – True by default; this caches geocoding results in the database, which makes geocoding faster, but debugging harder, and can add a bit to the size of database.

EB_DOMAIN – The domain used for the root of some generated URLs, eg. in feeds, widgets, and generated emails.

EB_MEDIA_ROOT – Directory that's the root of ebpub's static media files. By default this is calculated from the location of the installed ebpub package.

HTTP_CACHE – Cache directory used by scrapers when fetching data from remote sites. By default this goes in a subdirectory of '/tmp'.

NEIGHBORNEWS_USE_CAPTCHA – Whether to put a ReCaptcha form on the forms for adding user-contributed news. Only relevant if ebpub.neighbornews is in settings.INSTALLED_APPS. This can be True, False, or a function that takes a "request" argument and returns True or False. You'll also need to acquire API keys from recaptcha.org and set them as RECAPTCHA_PUBLIC_KEY and RECAPTCHA_PRIVATE_KEY.

JQUERY_URL – URL where our version of JQuery lives. Default is a hosted version.

OPENLAYERS_URL – URL where our version of OpenLayers lives. Default is currently OpenLayers 2.11, hosted locally.

OPENLAYERS_IMG_PATH – URL where OpenLayers images are found.

SCRAPER_LOGFILE_NAME – Where scrapers should log their output.

SCRAPER_LOG_DO_EMAIL_ERRORS – Whether scrapers should log their output.

SHORT_NAME – The short name for your city, in lowercase, eg. "chicago". This is used mainly for determining the default metro (see *Configuring Cities / Towns: METRO_LIST*), which is used through the OpenBlock code.

5.1.8 Loading Geographic Data

Overview

OpenBlock needs several kinds of geographic data for your city or region. This document explains what you need and how to load it.

You will need the following:

- Boundaries for local areas of interest to your users, such as neighborhoods, towns, ZIP codes, political districts, etc.

  It is recommended to set up ZIP codes first; if you are using a *multiple_cities* configuration, you should also load your city boundaries first.

- A database of city streets and blocks. This is used for geocoding, for address searches, and for browsing news by block.

  Note that this data is *not* used to generate background tile images for your maps. Those are provided by a separate service such as Google Maps or a WMS server. See *Choosing Your Map Base Layer* for more on setting up your map base layer.

Admin UI or command line?

It is now possible to do all your geographic data loading via the OpenBlock's web admin UI at http://localhost:8000/admin, or via command-line scripts. It is purely a matter of preference.

Multiple Cities?

If the area you're interested in isn't a single city, be sure to read *Configuring Cities / Towns: METRO_LIST*, especially the *multiple_cities* section.

USA Only?

OpenBlock was originally written with the assumption that it will be installed in the USA, for a major metropolitan area. It may be possible to work around those assumptions, but using OpenBlock outside the USA is not officially supported at this time. We encourage experimentation and asking questions on the mailing list; we know of several people currently trying it in other countries.

Background Tasks

Several of the things you can do in the admin UI can potentially take a long time, so they are launched as background tasks. If you want to use these admin UI features, be sure to read this section.

You'll need to do this once at server start time:

```
$ export DJANGO_SETTINGS_MODULE=myblock.settings  # change as needed
$ django-admin.py process_tasks
```

Like the *runserver* command, this won't immediately exit. It will sit quietly until there are background jobs to process for installing geographic data.

NB. If you are installed on EC2, then `django-admin.py process_tasks` is already run via a cron job.

Why not Celery?

We use django-background-task for our background jobs. Celery is common in the Django world for handling asynchronous tasks, and is a more mature, robust, and featureful solution. However, our mandate was to make OpenBlock as easy as possible to install, and we could not justify burdening our users with yet another service to install, configure, and maintain.

US ZIP Codes

Load ZIP Codes via Admin UI

If you have a list of the ZIP codes you'd like to install, just be sure the *background task daemon* is running. Then you can surf to `http://<your domain>/admin/db/location/` and click the link "Import ZIP Shapefiles". You can pick your state, paste your list of ZIPs, and wait for the import to finish.

You can do this several times if your area crosses state lines.

When this is done, skip down to *Verifying ZIP Codes* below.

(TODO: screen shot?)

Load ZIP Codes via Command Line

Alternately, you can use command-line scripts to install ZIP codes.

The US Census Bureau has shapefiles for all USA zip codes. Go to http://www2.census.gov/cgi-bin/shapefiles2009/national-files, select your state from the drop-down, and submit. Toward the bottom of the file list, you should see one labeled "5-Digit ZIP Code Tabulation Area (2002)".

Download that file. It should have a name that looks like `tl_2009_36_zcta5.zip` where 36 is a state ID (in this case, 36 is for New York).

Unzip the file. It should contain a number of files like this:

```
$ unzip tl_2009_36_zcta5.zip
inflating: tl_2009_36_zcta5.dbf
inflating: tl_2009_36_zcta5.prj
inflating: tl_2009_36_zcta5.shp
inflating: tl_2009_36_zcta5.shp.xml
inflating: tl_2009_36_zcta5.shx
```

The ZIP code file you downloaded is for an entire state. You're probably not setting up OpenBlock for an entire state, so you'll need to filter out those ZIP codes that are irrelevant to your area of interest. The zip import script allows you to do that, if you have configured your *metro extent*.

```
$ import_zips_tiger -v -b /path/to/where/you/unzipped/the/files/
```

The `-b` option tells it to filter out zip codes outside your metro extent, and `-v` tells it to give verbose output.

It will tell you which ZIP codes were skipped, and at the end, print a count of how many were created.

Verifying ZIP Codes

To verify that your ZIP codes loaded, point your browser at the home page. There should be a link to view "61 ZIP codes", or however many you loaded. Follow that to see a list of them all, and click on one to see a page about that ZIP code.

If you want to have a look "under the hood", you can use the django admin UI to do so. Browse to http://localhost:8000/admin , and enter your admin username / password when prompted.

Navigate to "Db" -> "Location Types". You should see that there is a Location Type called "ZIP Code" in the system now.

Navigate back to "Db", then go to "Db" -> "Locations". You should see a number of ZIP codes in the list. If you click on one, you should see an edit form that contains a map, showing you the borders of this ZIP code.

(TODO: screen shot?)

Streets / Blocks

Finding Blocks Data

In the US, the Census Bureau's TIGER data website is a good source of data. From http://www2.census.gov/cgi-bin/shapefiles2009/national-files, you will need several files. First select the State you're interested in. Download the file labeled "Place (Current)".

Next, select the County you're interested in. From the county's page, download the files labeled "All Lines", "Topological Faces (Polygons With All Geocodes)", and "Feature Names Relationship File".

Loading Blocks using the Admin UI

It's easy to use the admin UI to load US Census shapefiles. First, be sure the *background task daemon* is running.

Then you can surf to `http://<your domain>/admin/streets/blocks/` and click the link "Import Block Shapefiles". Type in the city name that these blocks are in, upload the four zip files you downloaded above, click "Import" and wait for it to finish.

(It is likely to take several minutes - more or less, depending on your hardware; this is the most computationally intensive thing that OpenBlock ever does.)

Streets, Intersections, and BlockIntersections will be done automatically.

You can repeat this process if your area spans multiple shapefiles. (It tends to get slower as the number of intersections grows.)

When done, skip down to *Verifying Blocks*.

(TODO: screen shot?)

Loading Blocks using the Command Line

You don't have to use the admin UI if you're happy at the command line. It takes several steps.

Loading Blocks from Census TIGER files First, unzip all four files you downloaded in *Finding Blocks Data*.

The block importer can filter out blocks outside the city named by the `--city` option. It can also filter out blocks outside your *metro extent* by passing the `--filter-bounds` option.

You can run it like this (assuming all the unzipped shapefiles are in the current directory):

```
$ import_blocks_tiger --city=BOSTON --filter-bounds \
  tl_2009_25025_edges.shp tl_2009_25025_featnames.dbf \
  tl_2009_25025_faces.dbf tl_2009_25_place.shp
```

The order of file arguments is important. First give the edges.shp filename, then the featnames.dbf file, then the faces.dbf file, then the place.shp file.

The filenames would be different from the example shown for a different city/county, of course.

Be patient; it typically takes at least several minutes to run.

It can also filter out blocks outside of one or more locations by passing the `--filter-location` option with a LocationType slug and Location slug; for example:

```
$ import_blocks_tiger --filter-location="cities:cambridge" \
  --filter-location="cities:newton" ...
```

If you run it with the `--help` option, it will tell you how about all options:

```
$ import_blocks_tiger  --help
Usage: import_blocks_tiger edges.shp featnames.dbf faces.dbf place.shp

Options:
 -h, --help            show this help message and exit
 -v, --verbose
 -c CITY, --city=CITY  A city name to filter against
 -f, --fix-cities      Whether to override "city" attribute of blocks and
                       streets by finding an intersecting Location of a city-
                       ish type. Only makes sense if you have configured
                       multiple_cities=True in the METRO_LIST of your
                       settings.py, and after you have created some
                       appropriate Locations.
 -b, --filter-bounds   Whether to skip blocks outside the metro extent from
                       your settings.py. Default True.
 -e ENCODING, --encoding=ENCODING
                       Encoding to use when reading the shapefile
```

Loading Blocks from ESRI files If you have access to proprietary ESRI blocks data, you can instead use the script `ebpub/streets/blockimport/esri/importers/blocks.py`.

Populating Streets and Intersections After all your blocks have loaded, you *must* run another script to derive streets and intersections from the blocks data. This typically takes several minutes for a large city.

The following commands must be run *once*, in exactly this order:

```
$ populate_streets -v -v -v -v streets
$ populate_streets -v -v -v -v block_intersections
$ populate_streets -v -v -v -v intersections
```

The `-v` argument controls verbosity; give it fewer times for less output.

Verifying Blocks

Try starting up django and browsing or searching some blocks:

```
$ django-admin.py runserver
```

Now browse http://localhost:8000/streets/ and have a look around. You should see a comprehensive list of streets on that page, and each should link to a list of blocks. On the list of blocks, each block should link to a detail page that includes a map of a several-block radius.

You should also be able to search. In the search bar at top right, type in some addresses or intersections that you know should exist, and verify that they're found.

No Blocks?

If you don't get any blocks, it's possible that the shapefiles you downloaded don't correspond to the geographic area you configured in settings.py. Double-check that you downloaded the right file, and that your *metro extent* covers the same area.

Other Locations: Neighborhoods, Etc.

What kinds of locations?

Aside from ZIP codes, what kinds of geographic regions are you interested in?

OpenBlock can handle any number of types of locations. You can use the admin UI to create as many `LocationTypes` as you want, by visiting http://localhost:8000/admin/db/locationtype/ and click "Add". Fill out the fields as desired. You'll want to enable both 'is_browsable' and 'is_significant'.

(Note also that the shapefile import scripts described below can create LocationTypes for you automatically, so you may not need to do anything in the admin UI.)

You're limited only by the data you have available. Some suggestions: try looking for neighborhoods/districts/wards, police precincts, school districts, political districts...

Drawing Locations by Hand

If you don't have shapefiles available, it's always possible to hand-draw locations in the admin UI. This is a great option for relatively simple shapes where you don't need to be very precise with the edges. This might also be appropriate for areas whose boundaries are informal. For example, often local residents will have a general sense of where neighborhoods begin and end, but there may not be "official" boundaries published anywhere.

Just browse to */admin/db/locations*, click "Add location", drag and zoom the map as desired, select a location type, and start clicking away on the map.

When happy with your polygon, double-click on the last point to stop drawing.

To modify it, click the “Modify features” icon in the map toolbar and then you can click and drag individual points, or click a point and hit the Delete key to remove a point.

There are Undo and Redo buttons, although the history will be forgotten once you click the Save button on the form.

(TODO: screen shots?)

For precise complex shapes, it’s just not practical to draw a 500-point polygon in our admin UI.

Finding Location Data

The trouble with loading local place data is that, at least in the USA, there is no central agency responsible for all of it, and no standards for how local governments should publish their geospatial data. This means it’s scattered all over the web, and we can’t just tell you where to find it.

Try googling for the name of your area plus “shapefiles”.

Loading Location Data

Once you have one or more Location Types defined, you can start populating them, either via the command line or the admin UI.

Admin UI: Importing Locations Browse to /admin/db/locations, click “Upload Shapefile”, and upload the zipped file you downloaded. Submit the form.

On the next screen, you can choose a Location Type, then choose from the “layers” available in this shapefile (often there is only one).

Then you get to choose which field contains the name of each location. The form will show you an example value from each field, so it’s usually pretty obvious which field is the one to choose. (If none of them make any sense, it’s possible that this shapefile isn’t usable by OpenBlock.)

Submit the form and you’re done.

Command Line: Importing Locations From Shapefiles There is a script `import_locations` that can import any kind of location from a shapefile. If a LocationType with the given slug doesn’t exist, it will be created when you run the script.

If you run it with the `--help` option, it will tell you how to use it:

```
$ import_locations  --help

Usage: import_locations [options] type_slug /path/to/shapefile

Options:
 -h, --help            show this help message and exit
 -n NAME_FIELD, --name-field=NAME_FIELD
                       field that contains location's name
 -i LAYER_ID, --layer-index=LAYER_ID
                       index of layer in shapefile
 -s SOURCE, --source=SOURCE
                       source metadata of the shapefile
 -v, --verbose         be verbose
 -b, --filter-bounds   exclude locations not within the lon/lat bounds of
                       your metro's extent (from your settings.py) (default
                       false)
 --type-name=TYPE_NAME
                       specifies the location type name
 --type-name-plural=TYPE_NAME_PLURAL
                       specifies the location type plural name
```

All of these are optional. The defaults often work fine, although `--filter-bounds` is usually a good idea, to exclude areas that don't overlap with your metro extent.

Command Line: Neighborhoods From Shapefiles There is also a variant of the location importer just for neighborhoods. Historically, "neighborhoods" have been a bit special to OpenBlock - there are some URLs hard-coded to expect that there would be a LocationType with slug="neighborhoods".

Again, if you run this script with the `--help` option, it will tell you how to use it:

```
$ import_neighborhoods  --help
Usage: import_neighborhoods [options] /path/to/shapefile

Options:
  -h, --help            show this help message and exit
  -n NAME_FIELD, --name-field=NAME_FIELD
                        field that contains location's name
  -i LAYER_ID, --layer-index=LAYER_ID
                        index of layer in shapefile
  -s SOURCE, --source=SOURCE
                        source metadata of the shapefile
  -v, --verbose         be verbose
  -b, --filter-bounds   exclude locations not within the lon/lat bounds of
                        your metro's extent (from your settings.py) (default
                        false)
```

Again, all of the options are really optional. The defaults often work fine, although `--filter-bounds` is usually a good idea, to exclude areas that don't overlap with your metro extent.

Can I load KML, GeoJSON, OpenStreetMap XML, or other kinds of files?

No, at this time the only files we can directly import are shapefiles. Try using tools like ogr2ogr to convert your data into shapefiles.

Places

TODO: document what Places are, how they differ from Locations, and why you'd care.

Alternate Names / Misspellings

Often users will want to search your site for an address or location, but they may spell it wrong - or it may have multiple names.

OpenBlock provides a simple way that you can support these searches.

You can use the admin UI at `/admin/streets/streetmisspelling/` to enter alternate street names. Click the "Add street misspelling" button, then type in the incorrect (alternate) and correct version of the street name.

Likewise, you can use the `/admin/db/locationsynonym/` page to add alternate names for Locations, and the `/admin/db/placesynonym` page to add alternate names for Places.

5.1.9 Indices and tables

- search

5.2 Using OpenBlock

This section is all about what you can **do** with OpenBlock after you've taken care of Installation and Setup.

5.2.1 Creating a Custom NewsItem Schema

In OpenBlock lingo, a *Schema* is a type of *NewsItem*. (It's a good idea to read *Brief Overview: Concepts and Terminology* if you haven't.)

OpenBlock's ebpub package provides several models for defining Schemas. This section provides a brief example of creating a Schema, defining its custom fields, and creating a NewsItem with the Schema.

It is assumed for this section that you have installed either the demo or have created a custom application.

For background and additional detail, see also *SchemaFields and Attributes* in the ebpub documentation, the code in ebpub.db.models and the video "Behind the scenes of EveryBlock.com"

Cleaning Up After Experiments

Typically it will take you a little while to settle on a schema you like, and you may accumulate some junk newsitems while testing. It's often useful to clean these up and start fresh. There is a script at `ebpub/db/bin/delete_newsitems.py` that can delete all NewsItems of a given Schema, plus all their Attributes and Lookups.

Experimenting with Existing Schemas

When running *syncdb –migrate* with `'ebpub'` in `settings.INSTALLED_APPS`, a few default schemas will be loaded for you: a basic 'Local News' type, and an 'Open311 Service Requests' type.

If you also have obdemo in `settings.INSTALLED_APPS`, you will get the schemas that are used by our Boston demo, including: Boston events, restaurant inspections, building permits, and Boston police reports.

You may wish to look at these in the admin UI to see how they're configured. Some of them might be appropriate for your purposes out of the box or with slight modifications.

You can of course use the admin UI to delete any that you don't want to use.

Note that modifications and deletions are permanent. We don't have an "undo" feature, sorry.

Make a Schema From Scratch

For this example, we will model a "Crime Report". Beyond the basic NewsItem information, like title, date, location etc, we will want to record some custom information with each report:

- Responding officer's name
- Type of crime (in english)
- Police code for crime

Steps are shown using the django shell, but this could also be performed in a script, or similar steps in the administrative interface. This code can also be found in **misc/examples/crime_report_schema.py**. This section assumes your application is *myblock*; substitute your own or *obdemo* for the demo application. Start in the root of your virtual env:

```
$ source bin/activate
$ django-admin.py shell --settings=myblock.settings
Python 2.6.1 (r261:67515, Feb 11 2010, 00:51:29)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Creating the Schema

The first step is to create an *ebpub.db.models.Schema* to represent the *Crime Report* type:

```
>>> from ebpub.db.models import Schema
>>> crime_report = Schema()
```

This object will contain metadata about all Crime Reports, like what its title is and how to pluralize it:

```
>>> crime_report.indefinite_article = 'a'
>>> crime_report.name = "Crime Report"
>>> crime_report.plural_name = "Crime Reports"
```

The *slug* is the unique identifier for this Schema that will be used in URLs on the site. It should be brief and contain URL safe characters:

```
>>> crime_report.slug = 'crimereport'
```

The *min_date* field can be used to limit how far back the user can navigate when viewing crime reports. For now, we'll just assume that everything is in the future:

```
>>> from datetime import datetime
>>> crime_report.min_date = datetime.utcnow()
```

The *last_updated* field tracks when we last checked for new crime reports. We'll also just stub this out to the current time for now:

```
>>> crime_report.last_updated = datetime.utcnow()
```

The *has_newsitem_detail* field controls whether this item has a page hosted on this site, or whether it has its own external url. We'll host these ourselves:

```
>>> crime_report.has_newsitem_detail = True
```

The *is_public* field controls whether or not NewsItems of this type are visible to anybody other than administrators on the site. Normally you should wait until the type is set up and loaded with news before "turning it on". We'll just make it available immediately:

```
>>> crime_report.is_public = True
```

The *is_event* field controls whether or not NewsItems of this type are announcements of future events, rather than news that happened in the past. For more details on how to do this, see *Event-like News Types* This doesn't apply to crime reports, so we'll leave it set to False:

```
>>> crime_report.is_event = False
```

There are a few additional fields you can explore (see the code in ebpub.db.models.Schema), but this will be good enough to start with. So let's save it and move on:

```
>>> crime_report.save()
```

At this point you will be able to see the type listed on your site's front page, and reach an (empty) listing using your slug by visiting http://localhost:8000/crimereport assuming you are running the web server.

Making Maps Prettier

If you want your different NewsItem types to stand out from each other on maps, you have two options.

You can set your schema's map_color to a hex color code (eg. #FF0000), and markers for that news type will be filled with that color.

Or, you can set your schema's map_icon_url to the URL of an image to use for its markers. Should be roughly 35x35 pixels. (If you are hosting your own map icons, it's fine to use a relative URL here.)

OpenBlock does not currently ship with any map icons. One source of good free (Creative Commons 3.0 BY-SA) map icons is http://mapicons.nicolasmollet.com/ .

Adding Custom Fields

As mentioned earlier, we will add the following custom fields:

- Responding officer's name
- Type of crime (in english)
- Police code for crime

We will create an ebpub.db.models.SchemaField to describe each custom field. Let's start with the reporting officer:

```
>>> from ebpub.db.models import SchemaField
>>> officer = SchemaField()
>>> officer.schema = crime_report
>>> officer.pretty_name = "Reporting Officer's Name"
>>> officer.pretty_name_plural = "Reporting Officer's Names"
```

The values of *all* the custom fields for a particular NewsItem will be stored in a single ebpub.db.models.Attribute object. The Attribute object has a fixed set of fields which can be used for custom attributes. The fields are named according to their type, and numbered:

Names	Possible Numbers	Type
varcharNN	01 - 05	models.CharField (length 255)
dateNN	01 - 05	models.DateField
timeNN	01 - 02	models.TimeField
datetimeNN	01 - 04	models.DateTimeField
boolNN	01 - 05	models.NullBooleanField
intNN	01 - 07	models.IntegerField
textNN	01	models.TextField

Each SchemaField will map onto one of the fields of the Attribute class. We'll map the reporting officer onto the first varchar field *varchar01* by setting the real_name attribute:

```
>>> officer.real_name = 'varchar01'
```

When working with a crime report NewsItem, we'll want to have an alias for this attribute in the code, so we don't always have to remember what 'varchar01' means for crime reports. This is set using the `name` field of the SchemaField. We'll call it *officer*, and move on:

```
>>> officer.name = 'officer'
```

That's the important stuff. There are a bunch of mandatory display-related fields; we'll just gloss over these for now:

```
>>> officer.display = True
>>> officer.display_order = 10
>>> officer.is_searchable = True
>>> officer.is_lookup = False
>>> officer.is_filter = False
>>> officer.is_charted = False
```

Now we can save this SchemaField:

```
>>> officer.save()
```

The name of the crime works the same way, but we'll need to store it in a different field. We'll use the second varchar field, *varchar02*:

```
>>> crime_name = SchemaField()
>>> crime_name.schema = crime_report
>>> crime_name.real_name = "varchar02"
>>> crime_name.pretty_name = "Crime Type"
>>> crime_name.pretty_plural_name = "Crime Types"
>>> crime_name.name = "crime_type"
>>> crime_name.display = True
>>> crime_name.display_order = 10
>>> crime_name.is_searchable = True
>>> crime_name.is_lookup = False
>>> crime_name.is_filter = False
>>> crime_name.is_charted = False
>>> crime_name.save()
```

For the crime code, we'll use an integer field:

```
>>> crime_code = SchemaField()
>>> crime_code.schema = crime_report
>>> crime_code.real_name = "int01"
>>> crime_code.pretty_name = "Crime Code"
>>> crime_code.pretty_plural_name = "Crime Codes"
>>> crime_code.name = "crime_code"
>>> crime_code.display = True
>>> crime_code.display_order = 10
>>> crime_code.is_searchable = True
>>> crime_code.is_lookup = False
>>> crime_code.is_filter = False
>>> crime_code.is_charted = False
>>> crime_code.save()
```

Phew, okay we just designed a NewsItem type!

Creating a NewsItem with the Schema

Now we can finally start churning out amazing crime reports. We start by making a basic news item with our schema and filling out the basic fields:

```
>>> from ebpub.db.models import NewsItem
>>> report = NewsItem()
>>> report.schema = crime_report
>>> report.title = "Hooligans causing disturbance downtown"
>>> report.location_name = "123 Fakey St."
>>> report.item_date = datetime.utcnow()
>>> report.pub_date = datetime.utcnow()
>>> report.description = "Blah Blah Blah"
>>> report.save()
```

Great, now (any only now) we can set the extra fields, which are weirdly immediately set when accessing the special `attributes` dictionary on the NewsItem. (There is some python magic going on, see the code in `ebpub.db.models`.) We use the names that we assigned when we were designing the schema:

```
>>> report.attributes['officer'] = "John Smith"
>>> report.attributes['crime_type'] = "Disturbing The Peace"
>>> report.attributes['crime_code'] = 187
```

If you visit the crime reports page at http://localhost:8000/crimereport it should list your new item. You can click its link to view the custom details you added.

Hooray!

Lookups: normalized enums

For attributes that have only a few possible values, you can add another layer of indirection called a Lookup to confuse you... err, normalize the data somewhat. See *Lookups* for more.

5.2.2 Data Scraper Tutorial

Currently, anybody using OpenBlock will have to write their own scraper scripts to import news data.

You have several options for how to write scraper scripts. We'll look at each in turn:

1. *Use an Existing Scraper* (separate page) if there is one that meets your needs.
2. *Use the OpenBlock REST API* to push data in from any script in any language that can make HTTP connections.
3. *Expediently hack a Python script* that creates instances of ebpub.db.NewsItem, in any way you like.
4. For "*list/detail*" sites, – sites that display a list of records (eg. an RSS feed, or an HTML index page), with optional separate linked pages providing more detail about each record – you can write a Python script that builds on the infrastructure in `ebdata.retrieval.scrapers.newsitem_list_detail`.
5. For "*unstructured*" sites - websites not intended for machine consumption, eg. big piles of HTML and/or binary files such as PDF or Excel - you can write a Python script that builds on `ebdata.blobs`.

Let's look at each option in turn. But first, we need a Schema.

Setting up Schemas

First of all, we're going to need a Schema that describes our NewsItems.

This is fully documented at Creating a Custom NewsItem Schema. Our examples will use schemas that are bootstrapped by installing obdemo.

Using the REST API

This is an especially good solution if you mainly have experience with, or access to programmers experienced with, languages other than Python.

You can use any programming language or tool that's able to make HTTP connections to your OpenBlock site, and work with JSON data. That's just about any modern language.

The general approach will be the same regardless of language:

- Fetch data from the source you're interested
- Parse the data
- For each news item you parsed:
  - Massage the item data into *the GeoJSON format* required by our API
  - Send a *POST request* to push the news item into OpenBlock.

TODO: write an example, maybe in something other than Python?

"Expedient Hack" scraping

If you only have a couple hours for a proof of concept, can write a little Python, and aren't yet deeply familiar with OpenBlock, this is a good way to start.

You can always refactor it into something more robust later.

The process is conceptually simple. The script should download some data from the web, create one or more NewsItems whose fields are populated with that data, and save the NewsItems. The grunt work is in extracting and massaging the data you need.

Here's an example. This script uses feedparser to fetch an RSS feed from boston.com and creates a NewsItem for each entry:

```
#!/usr/bin/env python

"""A quick-hack news scraper script for Boston; consumes RSS feeds.
"""

import datetime
import feedparser
import logging

from django.contrib.gis.geos import Point
from ebpub.db.models import NewsItem, Schema
from ebpub.utils.logging import log_exception

# Note there's an undocumented assumption in ebdata that we want to
# unescape html before putting it in the db.
from ebdata.retrieval.utils import convert_entities

logger = logging.getLogger()

def main():
    logger.info("Starting add_news")
    url = 'http://search.boston.com/search/api?q=*&sort=-articleprintpublicationdate&subject=massachu

    schema = Schema.objects.get(slug='local-news')
    for entry in feedparser.parse(url):
        try:
            # Check to see if we already have this one.
            item = NewsItem.objects.get(schema__id=schema.id, url=entry.link)
            logger.debug("Already have %r (id %d)" % (item.title, item.id))
        except NewsItem.DoesNotExist:
            # Nope, we need to create a new one.
            item = NewsItem()

        item.schema = schema
        item.title = convert_entities(entry.title)
        item.description = convert_entities(entry.description)
        item.url = entry.link
        item.item_date = datetime.datetime(*entry.updated_parsed[:6])
        item.pub_date = datetime.datetime(*entry.updated_parsed[:6])

        item.location_name = entry.get('x-calconnect-street') or entry.get('georss_featurename') or u
        point = entry.get('georss_point') or entry.get('point')
        if not point:
             # Don't bother saving. There's no point if there's no point ;)
             continue
        x,y = point.split(' ')
        item.location = Point((float(y), float(x)))

        # If our Schema had some SchemaFields, we'd save them now like so:
        # item.attributes = {'foo': 'bar', ...}

        item.save()

    logger.info("Finished add_news")

if __name__ == '__main__':
    import sys
    args = sys.argv
    loglevel = logging.INFO
    if '-q' in args:
        loglevel = logging.WARN
    logger.setLevel(loglevel)
    main()
```

This script actually runs. A longer version is at `obdemo/scrapers/add_news.py`.

So, what's left out? Among other things:

- We don't really do much error handling.
- This scraper doesn't demonstrate address parsing or geocoding, since this feed happens to provide location names and geographic points already.
- We get all our information directly from the feed and don't follow any links to other documents. Sometimes you need to do that.
- This schema doesn't require any custom attributes, so we don't show that. It's trivial though, just assign a dictionary to item.attributes.

Also notice the `-q` or `--quiet` command-line option that silences all non-error output. This is an OpenBlock scraper convention intended to allow running scrapers under *Cron Configuration* without sending yourself tons of useless email messages.

Using NewsItemListDetailScraper for List/Detail pages

A "list-detail site" is a site that displays a list of records (eg. an RSS feed, or an HTML index page), which might be paginated. Each record might have its own page – a "detail" page – or the list/feed might include all the information you need.

Here's an example that doesn't use detail pages. This is a slightly simplified version of the `ebdata/scrapers/us/ma/boston/police_reports/retrieval.py` script. It uses a Schema that's loaded when bootstrapping the `obdemo` package.

Since this feed doesn't provide locations, we'll use ebdata's code for address extraction and ebpub's geocoder:

```python
from ebdata.nlp.addresses import parse_addresses
from ebdata.retrieval.scrapers.list_detail import RssListDetailScraper
from ebdata.retrieval.scrapers.newsitem_list_detail import NewsItemListDetailScraper
from ebdata.textmining.treeutils import text_from_html
from ebpub.db.models import NewsItem
from ebpub.geocoder import SmartGeocoder
from ebpub.geocoder.base import GeocodingException
from ebpub.utils.logging import log_exception
import logging
import datetime


class BPDNewsFeedScraper(RssListDetailScraper, NewsItemListDetailScraper):

    schema_slugs = ('police-reports',)
    has_detail = False

    def list_pages(self):
        # This gets called to iterate over pages containing lists of items.
        # We just have the one page.
        url = 'http://www.bpdnews.com/feed/'
        yield self.fetch_data(url)

    def existing_record(self, record):
        # This gets called to see if we already have a matching NewsItem.
        url = record['link']
        qs = NewsItem.objects.filter(schema__id=self.schema.id, url=url)
        try:
            return qs[0]
        except IndexError:
            return None

    def save(self, old_record, list_record, detail_record):
        # This gets called once all parsing and cleanup is done.
        # It looks a lot like our 'expedient hack' code above.

        # We can ignore detail_record since has_detail is False.

        date = datetime.date(*list_record['updated_parsed'][:3])
        description = list_record['summary']

        # This feed doesn't provide geographic data; we'll try to
        # extract addresses from the text, and stop on the first
        # one that successfully geocodes.
        # First we'll need some suitable text; throw away HTML tags.
        full_description = list_record['content'][0]['value']
        full_description = text_from_html(full_description)
        addrs = parse_addresses(full_description)
        if not addrs:
            self.logger.info("no addresses found")
            return

        location = None
        location_name = u''
        block = None
        # Ready to geocode. If we had one location_name to try,
        # this could be done automatically in create_or_update(), but
        # we have multiple possible location_names.
        for addr, unused in addrs:
            addr = addr.strip()
            try:
                location = SmartGeocoder().geocode(addr)
            except GeocodingException:
                log_exception(level=logging.DEBUG)
                continue
            location_name = location['address']
            block = location['block']
            location = location['point']
            break
        if location is None:
            self.logger.info("no addresses geocoded in %r" % list_record['title'])
            return

        kwargs = dict(item_date=date,
                      location=location,
                      location_name=location_name,
                      description=description,
                      title=list_record['title'],
                      url=list_record['link'],
                      )
        attributes = None
        self.create_or_update(old_record, attributes, **kwargs)


if __name__ == "__main__":
import sys
from ebpub.utils.script_utils import add_verbosity_options, setup_logging_from_opts
from optparse import OptionParser
if argv is None:
    argv = sys.argv[1:]
optparser = OptionParser()
add_verbosity_options(optparser)
scraper = BPDNewsFeedScraper()
opts, args = optparser.parse_args(argv)
setup_logging_from_opts(opts, scraper.logger)
# During testing, do this instead:
# scraper.display_data()
scraper.update()
```

That's not too complex; three methods plus some command-line option handling and you're done. Most of the work was in save(), doing address parsing and geocoding.

But you do have to understand how (and when) to implement those three methods. It's highly recommended that you read the source code for `ebdata.retrieval.scrapers.list_detail` and `ebdata.retrieval.scrapers.newsitem_list_detail`.

For a more complex example that does use detail pages and custom attributes, see `ebdata/scrapers/general/seeclickfix/seeclickfix_retrieval.py`.

What does this framework buy you? The advantage of using ebdata.retrieval.scrapers.newsitem_list_detail for such sites is that you get code and a framework for dealing with a lot of common cases:

- There's an `RssListDetailScraper` mix-in base class that handles both RSS and Atom feeds for the list page, with some support for pagination. (That saves us having to implement parse_list()).
- It supports all the advanced features of ebpub's NewsItems and Schemas, eg. arbitrary Attributes, Lookups, and the like (although this example doesn't use them).
- The `create_newsitem()` method can automatically geocode addresses if you have a single good address but no geographic location provided.
- The `display_data()` method allows you to test your feed without saving any data (or even without having a Schema created yet). Call this instead of update() during testing.
- The `safe_location()` method (not shown) can verify that a location name (address) matches a provided latitude/longitude.
- The `last_updated_time()` method (not shown) keeps track of the last time you ran the scraper (very useful if your source data provides a way to limit the list to items newer than a date/time).
- There are hooks for cleaning up the data, see the various `clean*` methods.

Disadvantage:

- It's fairly complex.
- You probably still have to do a fair amount of the error-handling, parsing (for things other than RSS or Atom feeds), and so forth.
- It requires you to understand the base classes (`NewsItemListDetailScraper` and `ListDetailScraper`), because it has a lot of "inversion of control" – meaning, you use it by subclassing one or more of the base classes, and overriding various methods and attributes that get called by the base class as needed. Until you fully understand those base classes, this can be confusing.

For a more complete example that uses detail pages and some of those other features, see `ebdata/scrapers/general/seeclickfix/seeclickfix_retrieval.py`.

Blobs

For "unstructured" sites, with a lot of raw HTML or binary files (Excel, PDF, etc.), you may want to build something based on ebdata.blobs.

We haven't done one of these yet.

Some examples you can peruse from the `everyblock` part of the the openblock-extras code (note that we lack Schemas for any of these):

```
everyblock/cities/sf/zoning/new_retrieval.py
everyblock/cities/boston/city_press_releases/retrieval.py
everyblock/cities/seattle/city_press_releases/retrieval.py
everyblock/cities/miami/city_press_releases/retrieval.py
everyblock/cities/charlotte/city_council/retrieval.py
everyblock/cities/charlotte/county_proceedings/retrieval.py
everyblock/cities/chicago/city_press_releases/retrieval.py
everyblock/cities/dc/news_articles/retrieval.py
everyblock/cities/nyc/news_articles/retrieval.py
everyblock/cities/philly/city_press_releases/retrieval.py
everyblock/cities/philly/city_council/retrieval.py
```

Running Your Scrapers

Once you have scrapers written, you'll need to run them periodically. Read Running Scrapers for more.

5.2.3 Running Scrapers

Once you've written your scraper scripts, you'll need to re-run them frequently to keep the news on your site up to date.

You can do this any way you like; *cron* would work fine.

There is also a *daemon* that comes with OpenBlock which is tailor-made for this purpose, although we're deprecating it as of the 1.1 release since there's no real reason for OpenBlock to reinvent this particular wheel.

Cron Configuration

Here's an example config file for running scrapers via cron.

important!

You must set the `DJANGO_SETTINGS_MODULE` environment variable, and use the python interpreter that lives in your *virtualenv*. It's also crucial that the user who runs each cron job have permission to run those scripts, and permission to write to any log files, etc. that the scrapers write to. I recommend using the same (non-root) user account you used for installing openblock.

```
# Put this in
SHELL=/bin/bash

# Edit these as necessary
DJANGO_SETTINGS_MODULE=obdemo.settings
SCRAPERS=/path/to/ebdata/scrapers
BINDIR=/path/to/virtualenv/bin
PYTHON=/path/to/virtualenv/bin/python
USER=openblock
# Where do errors get emailed?
MAILTO=somebody@example.com

# Format:
# m     h dom mon dow user   command

# Retrieve flickr photos every 20 minutes.
0,20,40 *  *   *  *   $USER  $PYTHON $SCRAPERS/general/flickr/flickr_retrieval.py -q

# Meetup can be slow due to hitting rate limits.
# Several times a day should be OK.
0 7,18,22 * * * $PYTHON $SCRAPERS/general/meetup/meetup_retrieval.py -q

# Aggregates every 6 min.
*/6      0  0   0  0   $USER  $BINDIR/update_aggregates --quiet
```

A more extensive example is in the `obdemo` source code; look for `sample_crontab`.

As noted in Data Scraper Tutorial, it's a very good idea if scripts have a command-line option to discard all non-error output, since cron likes to email you with all output. When using cron, silence is golden.

Updaterdaemon Configuration

Deprecated!

Since cron and similar tools work just fine, we're declaring Updaterdaemon deprecated; that is, we no longer recommend using it.

The daemon script is named runner.py and it lives in ebdata, more specifically at ebdata/retrieval/updaterdaemon/runner.py. To configure it, you need to write a (small) Python script that contains a list of TASKS.

There is an example config file at ebdata/retrieval/updaterdaemon/config.py, and the one we use for obdemo is at obdemo/sample_scraper_config.py.

What goes in the config file? Let's put together a (small) example based on the one for obdemo.

First, we need a function that imports and runs one of our scrapers, just once. Let's use the one from obdemo that creates "Events". Our function can look like:

```
def do_events():
    from obdemo.scrapers.add_events import main
    return main()
```

(Note that this function could do anything we want to run periodically; updaterdaemon actually doesn't know anything about scrapers per se. One other thing you probably want to do regularly is send out openblock's *E-mail alerts*.)

Next, we need a way to know when, or how often, that function should run. We'll use another function for that; let's call it a "time callback". The time callback takes one argument - a Python datetime - and returns True if we should run our scraper now, and False otherwise. Here's one that runs every ten minutes:

```
def every_ten_minutes(datetime):
    if datetime.minute % 10 == 0:
        return True
    return False
```

(Note that runner.py only wakes up and checks the time once per minute, so we don't need to be very careful here about the time check - we won't accidentally run this many times in one minute.)

(Note also that the example config file in ebdata/retrieval/updaterdaemon/config.py already contains factories to generate a number of useful time callbacks, such as multiple_daily, daily, and weekly. We could just import and call one of those. Read the source to see how they work.)

Finally, we need to wrap all this up in a list (or tuple) calles TASKS. This is what the runner.py script looks for in the config file. TASKS is a list of tuples, each in the form (time_callback, function_to_run, {keyword args}, {environ}).

We've already got the first two of those. What about the last two? keyword args is a dictionary of extra arguments to pass to our function. Ours doesn't actually need any, so we'll use an empty dictionary, like {}.

environ is a dictionary of environment variables to set before running our function. Generally this will need to set DJANGO_SETTINGS_MODULE. For the demo, we set it to obdemo.settings by default, unless there is already an environment variable by that name. This looks like:

```
env = {'DJANGO_SETTINGS_MODULE': os.environ.get('DJANGO_SETTINGS_MODULE', 'obdemo.settings')}
```

Putting it all together, we get this complete config file:

```
from ebdata.retrieval.updaterdaemon.config import multiple_hourly

def do_events():
    from obdemo.scrapers.add_events import main
    return main()

def every_ten_minutes(datetime):
    if datetime.minute % 10 == 0:
        return True
    return False

env = {'DJANGO_SETTINGS_MODULE': os.environ.get('DJANGO_SETTINGS_MODULE', 'obdemo.settings')}

TASKS = (
    (every_ten_minutes, do_events, {}, env),
)
```

Testing the daemon

Give it a try:

```
$ python ebdata/ebdata/retrieval/updaterdaemon/runner.py --config=/path/to/config.py  start
```

If it works, nothing obvious should happen :) It's running in the background. You shouldn't expect anything to happen until the next multiple of 10 minutes. When it's time, check the log file to see if anything's happening:

```
$ tail -f /tmp/updaterdaemon.log
```

(Hit Ctrl-C to get out of that.)

If there's nothing in the main log, check the error log:

```
$ less /tmp/updaterdaemon.err
```

To stop the daemon, do this:

```
$ python ebdata/ebdata/retrieval/updaterdaemon/runner.py stop
```

Installing the init script

UpdaterDaemon also comes with a script suitable for putting in `/etc/init.d`, so it will be restarted whenever the system is rebooted. To install this script, copy it from `ebdata/retrieval/updaterdaemon/initscript` into something like `/etc/init.d/openblock-updaterdaemon`. It is known to work on Ubuntu; let us know if you have trouble with it on other linux systems.

After copying, edit the script, setting a few crucial environment variables:

`HERE` should point to the virtualenv where you installed OpenBlock.

`CONFIG` should point to a config file as described in the previous sections.

`SU_USER` should be the name of the user account to use for running the daemon.

You might also want to set `LOGFILE` and `ERRLOGFILE` to control where the logs go.

Now try running the script as root:

```
$ sudo /etc/init.d/openblock-updaterdaemon start
```

Check the log files to make sure it's working.

5.2.4 Sending Alerts

OpenBlock users can subscribe to email alerts from any location they're interested in.

In order to support this feature, there is a script that needs to be called regularly. You can do this any way you like; cron would work fine.

Here's an example crontab file that sends the daily alerts once a day, and the weekly alerts once a week. Adjust the environment variables as needed:

```
DJANGO_SETTINGS_MODULE=obdemo.settings
VIRTUAL_ENV=/path/to/my/environment
@daily $VIRTUAL_ENV/bin/send_alerts  --frequency daily
@weekly $VIRTUAL_ENV/bin/send_alerts --frequency weekly
```

Note that OpenBlock does not remember which alerts have already been sent, so you *should not* send eg. daily alerts more than once a day, or your users will get duplicate alert messages.

Disabling Alerts

Disabling Email Alerts entirely is easy. In your `settings.py`, just remove `"ebpub.alerts"` from `settings.INSTALLED_APPS`. (You can copy the setting from `ebpub/default_settings.py` and modify it.) This will remove the alerts sign-up links from all OpenBlock pages.

5.2.5 OpenBlock REST API

Introduction

Purpose

Support simple widgets, mashups, frontends based on OpenBlock content and filtering capability. We would appreciate any feedback you have on how to improve the usefulness and usability of this API.

Caveat

This is a preliminary work-in-progress API and may be changed substantially in future versions.

See Also

The OpenBlock API uses several standards for formats and protocols. Please see the (externally maintained) documentation focused on the particular formats for more details. These include GeoJSON, Atom, and JSONP. Some helpful links:

Format	URL
GeoJSON	http://geojson.org/
Atom	http://www.atomenabled.org
GeoRSS	http://www.georss.org/Main_Page
JSONP	http://en.wikipedia.org/wiki/JSON#JSONP
rfc 3339 (date)	http://www.ietf.org/rfc/rfc3339.txt (also the w3c "note-datetime" is essentially the same format: http://www.w3.org/TR/NOTE-datetime)

Examples / Quickstart

Grab some ‘articles’ about Roxbury

```
curl "http://bos.openblock.org/api/dev1/items.json?type=articles&locationid=neighborhoods/roxbury&li
```

API Overview

URL prefix

All calls to the OpenBlock API referenced in this document are prefixed by:

```
/api/dev1/
```

Authentication and API Keys

Authentication for those methods which require it may currently be accomplished in two ways:

- HTTP Basic Auth headers (any normal user account registered with OpenBlock will work)
- sending your API key in the `X-Openblock-Key` HTTP header

An API key may be obtained by logging in and visiting your preferences page, if the OpenBlock site makes that available to you; or the site administrators can create keys via the admin UI at `admin/key/apikey/`.

Support for Cross Domain Access

To enable widgets and mashups in the browser from domains other than the host OpenBlock instance, the API supports the JSONP convention.

Unless otherwise noted, all portions of the API using the http GET method support JSONP by providing the “jsonp” query parameter. The “jsonp” parameter may only contain letters, numbers, and underscores; other characters will be removed.

GET methods supporting *Atom* output may also provide the “jsonp” parameter. In this case the output is JSONP-X.

Rate Limits, AKA Throttling

The API may be configured to limit the number of API requests a user can perform during a certain time period. This limit applies across all resources provided by the API.

By default, the limit is 150 requests per hour.

If the user is not authenticated and provides no API key, the user’s IP address will be used.

If this limit is reached, the server will return a `503 SERVICE UNAVAILABLE` response, with a `Retry-After` header indicating the number of seconds after which the user can try again.

The site administrator controls throttling by changing several values in settings.py:

```
API_THROTTLE_AT=150  # max requests per timeframe.
API_THROTTLE_TIMEFRAME = 60 * 60  # Default 1 hour.
# How long to retain the times the user has accessed the API. Default 1 week.
API_THROTTLE_EXPIRATION = 60 * 60 * 24 * 7
```

```
# NOTE in order to enable throttling, you MUST also configure
# CACHES['default'] to something other than a DummyCache. Example:
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
    }
}
```

Read API Endpoints

GET [api prefix]

Purpose Test the availability of this version of the API. This request does not implement JSONP.

Response

Status	Meaning
200	This version of the API is available.
404	This version of the API is not available.
503	You have exceeded the *rate limit.*

GET items.json

Purpose Retrieve details of a certain set of news items as *NewsItem JSON Format*.

Parameters See section *Item Search Parameters*

Response

Status	Meaning
200	The request was valid, the response contains news items that match the criteria.
400	The request was invalid due to invalid criteria
503	You have exceeded the *rate limit.*

A successful response returns a GeoJSON FeatureCollection containing a list of *NewsItem JSON Format* features. Each resulting Feature in the collection represents a "NewsItem" that matches the specified search criteria ordered by item date.

Example result:

```
{"type": "FeatureCollection",
 "features": [
    {"type": "Feature",
     "properties": {
        "title": "An Article About Roxbury",
        "url": "...",
        "type": "articles",
        "description": "Test Roxbury",
        ...
     },
     "geometry": {
       "type": "Point",
       "coordinates": [-71.086787000000001, 42.314782999999998]
     }
    },
  ...
]}
```

GET items.atom

Purpose Retrieve details of a certain set of news items in ATOM format.

Parameters See section *Item Search Parameters*

Response

Status	Meaning
200	The request was valid, the response contains news items that match the criteria.
400	The request was invalid due to invalid criteria
503	You have exceeded the *rate limit.*

A successful response returns an Atom Feed. Each resulting Atom Entry in the feed represents a “NewsItem” that matches the specified search criteria ordered by item date.

Format is specified in the section *News Item Formats*

Example result

```
FIXME example
```

GET items/<id>.json

Purpose Get a single NewsItem as *NewsItem JSON Format*.

Parameters None.

Response

Status	Meaning
200	Found. The body will be the NewsItem represented as *NewsItem JSON Format*.
404	The NewsItem does not exist.
503	You have exceeded the *rate limit.*

GET geocode

Purpose Geocode a street address or location name to geographic location.

Parameters

Parameter	Description
q	address or location name to geocode

Response

Status	Meaning
200	The request was valid and locations matching the query were found
404	No locations matching the query were found.
400	Invalid input: missing or empty ‘q’ parameter.
503	You have exceeded the *rate limit.*

A successful response contains a GeoJSON FeatureCollection with Features corresponding to the query given. The list will contain multiple results if the match was ambiguous.

Example response:

```
"type": "FeatureCollection",
"features": [
 {
  "geometry": {
   "type": "Point",
   "coordinates": [
    -71.086787000000001,
    42.314782999999998
   ]
  },
  "type": "Feature",
  "properties": {
   "city": "BOSTON",
   "type": "neighborhoods",
   "name": "Roxbury",
   "query": "Roxbury"
  }
 }]}
```

A 404 response will return the same structure but with an empty list of “features”.

GET items/types.json

Purpose Retrieve metadata describing the types of news items available in the system and their attributes.

Response The output maps an identifier (“slug”) to a mapping of key-value pairs describing one news item type.

Each type consists of a few strings suitable for labels in a UI (‘name’, ‘plural_name’, ‘indefinite_article’), plus a ‘last_updated’ date when news items of this type were last loaded.

Each news item type may also have its own extended metadata which is described in the ‘attributes’ mapping. Each attribute has a ‘pretty_name’ and a ‘type’ (one of ‘text’, ‘bool’, ‘int’, ‘date’, ‘time’, ‘datetime’).

Example:

```
[{'elvis-sightings': {
   'indefinite_article': 'an',
   'name': 'Elvis Sighting',
   'plural_name': 'Elvis Sightings',
   'slug': 'elvis-sightings',
   'last_updated': '2011-02-22',
   'attributes': {
     'verified': {
       'pretty_name': 'Verified Really Elvis',
       'type': 'bool'
    }
  }
}]
```

GET locations.json

Purpose Retrieve all predefined locations on the server as a list.

Parameters

Parameter	Description
type	(optional) return only locations of the specified type, eg “neighborhoods” see See *GET locations/types.json* for types.

Response A list of JSON objects describing each location. Each has the following keys:

- name - human-readable name of the location.
- slug - name suitable for use in URLs.
- url - link to a view of this location as GeoJSON (see *GET locations/<locationid>.json*.
- description - may be blank.
- city - name of the city.
- type - a Location Type slug. See *GET locations/types.json*.

Example:

```
[
 {
  "city": "YOUR CITY",
  "description": "",
  "url": "/api/dev1/locations/zipcodes/02108.json",
  "type": "zipcodes",
  "slug": "02108",
  "name": "02108"
 },
 {
  "city": "YOUR CITY",
  "description": "",
  "url": "/api/dev1/locations/neighborhoods/allstonbrighton.json",
  "type": "neighborhoods",
  "slug": "allstonbrighton",
  "name": "Allston/Brighton"
 }
]
```

GET locations/<locationid>.json

Purpose Retrieve detailed geometry information about a particular predefined location. Available URLs can be discovered by querying the locations.json endpoint, see *GET locations.json*

Response A GeoJSON Feature object representing one named location.

Example:

```
{ "type": "Feature",
 "geometry": {
   "type": "Polygon",
   "coordinates": [
     [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0], ...
     ]
   },
 "properties": {
   "type": "zipcode",
   "city": "boston",
   "name": "02115",
   "slug": "02115",
   "description": "lorem ipsum blah blah",
   "centroid": "POINT (101.0 0.5)",
   "area": 3633354.76,
   "source": "http://example.com/zip_codes_or_something",
   "population": null,
   }
 },
```

GET locations/types.json

Purpose Retrieve a list of location types, eg “towns”, “zipcodes”, etc. which can be used to filter locations.

Response A JSON object describing the location types available.

Example:

```
{
 "towns": {"name": "Town",
           "plural_name": "Towns",
           "scope:" "boston"},
 "zipcodes": { ... }
}
```

GET places/types.json

Purpose Retrieve a list of place types, eg “points of interest”, “police stations”, etc. which can be used to access data about places in the system.

Response A JSON object describing the place types available.

Example:

```
{
    "poi": {
        "name": "Point of Interest",
        "plural_name": "Points of Interest",
        "geojson_url": "/api/dev1/places/poi.json"
    },
    "police": {
        "name": "Police Station",
        "plural_name": "Police Stations",
        "geojson_url": "/api/dev1/places/police.json"
    }
}
```

GET places/<placetype>.json

Purpose Retrieve a list of places of the specified type, eg “points of interest”, “police stations”, etc.

Response A GeoJSON feature collection object describing the places of the type specified.

Example:

```
{
 "type": "FeatureCollection",
 "features": [
  {
   "geometry": {
    "type": "Point",
    "coordinates": [
     -71.052149999999997,
     42.332369999999997
    ]
   },
   "type": "Feature",
   "properties": {
    "type": "poi",
    "name": "Fake Monument",
    "address": ""
   }
  },
  {
   "geometry": {
    "type": "Point",
    "coordinates": [
     -71.052149999999997,
     42.332369999999997
    ]
   },
   "type": "Feature",
   "properties": {
    "type": "poi",
    "name": "Fake Yards",
    "address": ""
   }
  }
 ]
}
```

Item Search Parameters

Search parameters specified select all items that match all criteria simultaneously, eg specifying type=”crimereport”&locationid=”neighborhoods/roxbury” selects all items that are of type “crimereport” AND in the Roxbury neighborhood and no other items.

Spatial Filtering

Spatial filters allow the selection of items based on geographic areas. At most one spatial filter may be applied per API request.

Predefined Area Selects items in some predefined area on the server, eg a neighborhood, zipcode etc. To discover predefined areas see the endpoint “GET locations.json”

Parameter	Description
locationid	server provided identifier for predefined location. eg: “neighborhoods/roxbury”

Bounding Circle Selects items within some distance of a given point.

Parameter	Description
center	<lon>,<lat> comma separated list of 2 floating point values representing the longitude and latitude of the center of the circle. eg: center=-71.191153,42.227865
radius	positive floating point maximum distance in meters from the specified center point

Other Filters

News Item Type Restricts results to a single type of news item, eg only crime reports. The full set of types available can be retrieved by querying the schema types list api endpoint or by inspection of the values of the 'type' field of news items returned from the api. See 'GET newsitems/types.json'

Parameter	Description
type	schemaid of the type to retrict results to, eg crimereport

Date Range Restricts results to items within a time range

Parameter	Description
startdate	limits items to only those whose pub_date is newer than the given date. date format is YYYY-MM-DD or rfc3339 for date/time
enddate	limits items to only those whose pub_date is older than the given date. date format is YYYY-MM-DD or rfc3339 for date/time

Result Limit and Offset

Parameter	Description
limit	maximum number of items to return. default is 25, max 200
offset	skip this number of items before returning results. default is 0

Write API Endpoints

POST items/

Purpose Create a new NewsItem. *Authentication required*.

Parameters The body of the POST must be a *NewsItem JSON Format* representation of a single NewsItem.

Note that you must include either the `geometry`, or `properties['location_name']`, or both:

- If `geometry` is omitted, the location_name will be used for geocoding to generate a geometry.
- If `location_name` is omitted, the geometry will be used for reverse-geocoding to generate a block name.
- If both are omitted, or geocoding/reverse-geocoding fails, it is an error.

Response

Status	Meaning
201	Created the NewsItem successfully. The 'Location' header will be a URI to the JSON representation of this NewsItem.

400	Invalid input. Response will be a JSON object with an 'errors' key containing validation hints.
	For example, if the required 'url' field is not provided and the 'item_date' is in the wrong format, the response would be:
	`{`
	`  "errors": {`
	`    "url": [`
	`      "This field is required."`
	`    ],`
	`    "item_date": [`
	`      "Enter a valid date."`
	`    ]`
	`  }`
	`}`

401	Permission denied. See *Authentication*.
503	You have exceeded the *rate limit.*

News Item Formats

NewsItem JSON Format

A NewsItem is represented by a GeoJSON Feature containing:

- a "geometry" attribute representing its specific location, generally a Point.
- a "type" attribute, which is always "Feature".
- a "properties" attribute containing details of the news item according to its schema.

See the GeoJSON specification for additional information on GeoJSON: http://geojson.org/geojson-spec.html

Example:

```
{
  "geometry": {
   "type": "Point",
   "coordinates": [
    -71.055719999999994, 42.359819999999999
   ]
  },
  "type": "Feature",
  "properties": {
    "title": "Looked kind of like Elvis",
    "type": "elvis-sightings",
    "description": "Witnesses reported someone who looked just like Elvis except eight feet tall and
    "url": "http://example.com/elvis123",
    "item_date": "2010-12-10",
    "pub_date": "2010-12-10T16:55:01-06:00",
    "verified": false,
    "location_name": "123 Main St, Springfield, MA",
  }
}
```

Common Properties The following `properties` are common to all Schema and will always be present:

Name	Type	Meaning
title	text	Headline or other title from the source.
type	text	Name (slug) of the item's type; this must correspond to one of the values returned by *GET items/types.json*
description	text	Summary of the news item.
url	text	Original URL where the news was found.
pub_date	rfc3339 date/time	Date/time this Item was added to the OpenBlock site. (Set automatically in *POST items/*.)
item_date	rfc3339 date	Date this news occurred, or was published on the original source site.
location_name	text	Human-readable name of the location.

Extended Properties: Schema Attributes Additional properties may be returned according to the NewsItem's type, aka *schema*.

In order to know what attributes are defined for each schema, or to know what to include in *POST items/*, you can do a request to *GET items/types.json*.

NewsItem Schema attributes are output in the corresponding JSON value type if one exists, otherwise a formatted string is used.

Field Type	JSON Representation
string	string
number	number
boolean	boolean
datetime	rfc3339 formatted datetime string, eg: "1999-12-29T12:11:45Z"
date	rfc3339 formatted date string, eg: "1999-12-29"
time	rfc3339 formatted time string, eg: "12:11:45Z"

NewsItem Atom Format

Generally follows the *Atom* specification. Location information is specified with *GeoRSS-Simple*.

Extended schema attributes are specified in the "http://openblock.org/ns/0" namespace.

Example:

```
<?xml version="1.0" encoding="utf8"?>
<feed xmlns="http://www.w3.org/2005/Atom"
      xmlns:openblock="http://openblock.org/ns/0"
      xmlns:georss="http://www.georss.org/georss">
   <title>openblock news item atom feed</title>
   <link href="/api/dev1/items.json" rel="alternate"></link>
   <link href="/api/dev1/items.atom" rel="self"></link>
   <id>/api/dev1/items.atom</id>
   <updated>2010-12-10T16:55:01-06:00</updated>
   <entry>
      <title>Looked kind of like Elvis</title>
      <link href="http://example.com/elvis123" rel="alternate"></link>
      <updated>2010-12-10T16:55:01-06:00</updated>
      <id>...</id>
      <summary type="html">Witnesses reported someone who looked just
         like Elvis except eight feet tall and with long red hair and green skin.
      </summary>
      <georss:point>42.3598199999999991 -71.0557199999999938</georss:point>
      <georss:featureName>4 S. Market St.</georss:featureName>
      <openblock:type>elvis-sightings</openblock:type>
      <openblock:attributes>
         <openblock:attribute type="bool" name="verified">False</openblock:attribute>
      </openblock:attributes>
   </entry>
</feed>
```

5.2.6 Widgets

Introduction

OpenBlock's widgets allow you to integrate OpenBlock content in external sites and fully customize output using Django templates.

You can also do some advanced configuration to *"pin"* some news items so they don't expire from the widget even if no longer current.

Widgets

Adding a new Widget

A new widget can be created by visiting the OpenBlock admin site and selecting the *Add* link next to *Widgets* in the *Widgets* section.

Configuring a Widget

The configuration of a widget combines criteria for selecting items to display with an output template for controlling the display of those items.

Currently items are always ordered by date.

Field	Meaning
name	Human readable name of the widget.
slug	A unique identifier used to refer to the widget.
description	Notes on the use or meaning of the widget; not for public display.
template	Which output template to use to show the items (see below).
max items	Maximum number of items to show in the widget.
location	Restrict the output to a particular predefined location.
types	Restrict the output to only the selected Schema types. If none are specified, any type is allowed.
item link template	If specified, links to item detail pages will use this template to determine where to link to.

Embedding Content

The administrative page for each widget provides two ways you can embed the widget on an external site.

Javascript Based Inclusion To use javascript based inclusion, cut and paste the “Embed Code” for the widget into any web page. The *div* will be replaced with the contents of the widget after the page loads. The div and javascript may be placed anywhere and do not need to be kept together.

Server Side Inclusion A request to the “Server Side Include URL” produces the output of the widget directly. This URL is suitable for use with any CMS or web server that can stitch pages together from content residing at different URLs or via sub-requests.

Templates

Each widget must have a ‘template’ which is used to generate the output that is included on the page. These templates are normal “Django Templates.” You can read the official Django template docs at:

```
http://docs.djangoproject.com/en/1.3/topics/templates/
```

There are also a variety of other tutorials and sources of information about Django templates available by casual googling.

When the template is rendered, OpenBlock will supply a context consisting of the items that should be displayed in some manner in the widget along with some information about the widget’s configuration.

Creating A Template

A new template can be created by visiting the OpenBlock admin site and selecting the *Add* link next to *Templates* in the *Widgets* section.

Item Info

The variable *items* contains a list of items that should be displayed by the widget. This list is generally looped over using the *for* template tag, eg:

```
{% for item in items %}
   <!-- output something about the item -->
   {{ item.title }}
{% endfor %}
```

Each item in the list contains a basic set of fields, and may include several extension fields that are particular to the type (schema) of the item.

Basic Fields

Field	Meaning
item.id	Openblock's unique identifier for this item.
item.title	The headline or title of the item.
item.internal_url	If the item is hosted by OpenBlock, this is a link to the OpenBlock page about the item. This value can be overridden via the "Item Link Template" setting on a widget.
item.external_url	If the item is hosted by an outside site, this is a link to the item.
item.pub_date	'Publication' date/time (the time when the content was added to OpenBlock). Must be formatted using a Django date filter, eg `{{item.pub_date|date:"Y m d h i"}}`. See http://docs.djangoproject.com/en/1.3/ref/templates/builtins/#std:templatefilter-date
item.item_date	Date (without time) associated with item. Meaning varies by item type. Must be formatted using a Django date filter, eg `{{item.item_date|date:"Y m d"}}`. See http://docs.djangoproject.com/en/1.3/ref/templates/builtins/#std:templatefilter-date
item.description	Description, body text, or text content of the item.
item.location.name	Text of location, address, place etc. Depending on item type and method of determining location, this may not be present or of varying meaning.
item.location.lat	Latitude of primary Point location of item.
item.location.lon	Longitude of primary Point location of item.
item.schema.name	The name of the type of item, eg "Restaurant Inspection".
item.schema.slug	The unique identifier of the item's type.
item.intersecting	A mapping of Location Type slugs to Locations that overlap this item. (TODO: example of how to use this)

Extension Fields Depending on the item's type (schema), a number of extension fields may be present. For example, a Restaurant Inspection might have a list of 'violations'; a Police Report might contain a field for a Crime Code.

Extended attributes can be accessed in two ways: By name via the `attributes_by_name` variable, or as an ordered list via the `attributes` variable. The attributes list is ordered according to the Display Order configured in the Schema's administrative user interface.

If you are using `attributes_by_name`, you access each attribute according to its unique identifier as configured in the Schema, eg:

```
{{ item.attributes_by_name.crime_code.value }}
```

If you are accessing the attributes as a list, you might say:

```
{% for attribute in item.attributes %}
  {{ attribute.value }}
{% endfor %}
```

No matter how it is accessed, each attribute houses the data and metadata about the attribute. The following fields are available:

Field	Meaning
attribute.name	Unique identifier of the attribute. This is the same as the name used in attributes_by_name, eg "crime_code".
attribute.title	Human readable title of the attribute, eg "Crime Code".
attribute.is_list	True if the attribute's value is a list of values, eg a list of codes or violations.
attribute.value	The value of the attribute. This may be a list in some cases, which can be tested via the is_list field.

Widget Info

The context variable `widget` provides information about the widget. The `widget` variable has the following fields:

Field	Meaning
widget.name	the human readable name of the widget
widget.slug	a unique identifier for the widget

Item Link Templates

An item link template can be specified to override the url used to link to detail pages for items listed in a widget by adjusting the 'item.internal_url' value available to the widget template.

For example, if your site has a different public url or url scheme than openblock uses internally, you can use this value to rewrite item links accordingly.

You may reference any of the fields shown above in your url template, but there is only one item, referenced as *item*. URL templates follow the same django template syntax above, but should evaluate to a single url.

Example:

```
http://mypublicsite.com/xzy/openblock/items/{{item.id}}/
```

This will link items to mypublicsite and fill in the identifier for the item being linked to depending on the item.

Note unless you have a specific reason not to, use the urlencode filter on any value that may contain unsafe characters for urls.

Example:

```
http://mypublicsite.com/track_click_and_redirect?realurl={{item.external_url|urlencode}}
```

Here, we link to a theoretical redirector on mypublicsite to capture a click through to an externally hosted (3rd party) detail page.

You are free to use django's full template syntax as long as the result contains a single url. Here for example, we perform some logic to determine whether to link internally, or use the redirector based on the item's schema:

```
{% if item.schema.slug == "restaurant-inspections" %}
    http://mypublicsite.com/xzy/openblock/inspections/{{item.id}}/
{% else %}
    http://mypublicsite.com/track_click_and_redirect?realurl={{item.external_url|urlencode}}
{% endif %}
```

Pinned Items, aka "Sticky Widgets"

Normally, a widget will show only the NewsItems that currently best match the specified type(s) and location. It's possible to configure widgets to "pin" certain NewsItems so they stay visible – or "stick" – either permanently, or until an expiration date you specify.

To do so, go to the admin UI and navigate to the widget you want to change. At top right, click the "configure sticky items" button.

The "Pinned Items" form shows a list of currently visible NewsItems on the left column. To pin one, drag it into an empty slot on the right column.

It will stay pinned in that position - either forever, or until the optional Expiration Date (and optionally a time).

You can re-order the pinned items by dragging and dropping up and down.

To manually remove a pinned item, just click the `x` button next to it.

When done with your changes, click the Save button.

5.2.7 Deployment

Apache

Most people use apache and mod_wsgi for deploying Django apps. If you're deploying obdemo, there's a suitable wsgi script at `obdemo/wsgi/obdemo.wsgi`; if you generated a custom app, there's a wsgi script at `src/<projectname>/<projectname>/wsgi/<projectname>.wsgi`.

For more info, see http://docs.djangoproject.com/en/1.3/howto/deployment/modwsgi/

Threading

Be warned that GeoDjango in general - and thus OpenBlock - is not safe to deploy multi-threaded. With mod_wsgi, this typically means setting the `threads=1` option in the `WSGIDaemonProcess` directive. See http://docs.djangoproject.com/en/1.3/ref/contrib/gis/deployment/ for more info.

Note on Virtual Hosting and Paths

Currently (2011/04/20), OpenBlock's views and templates (in the ebpub package) contain a lot of hard-coded URLs that only work if the site is deployed at the root of your domain.

In other words, you can deploy OpenBlock at http://example.com/ or http://openblock.example.com/ but you can't successfully deploy it at http://openexample.com/openblock.

Example Apache Config

Adjust the paths according to your installation.

```
<VirtualHost *:80>

ServerName ....compute-1.amazonaws.com

# Static media handling.
# You'll want the "expires" module enabled.

Alias /media/ /home/openblock/openblock/src/django/django/contrib/admin/media/
Alias /styles/ /home/openblock/openblock/src/openblock/ebpub/ebpub/media/styles/
Alias /scripts/ /home/openblock/openblock/src/openblock/ebpub/ebpub/media/scripts/
Alias /images/ /home/openblock/openblock/src/openblock/ebpub/ebpub/media/images/
Alias /cache-forever/ /home/openblock/openblock/src/openblock/ebpub/ebpub/media/cache-forever/
Alias /olwidget/  /home/openblock/openblock/src/django-olwidget/

# Only needed if you're running obdemo.
Alias /map_icons/ /home/openblock/openblock/src/openblock/obdemo/obdemo/media/map_icons/

<Directory /home/openblock/openblock/src/openblock/ebpub/ebpub/media/ >
  # I'm assuming everything here safely has a version-specific URL
  # whether via django-static or eg. the OpenLayers directory.
  ExpiresActive on
  ExpiresDefault "now plus 10 years"
</Directory>

WSGIScriptAlias / /home/openblock/openblock/src/openblock/obdemo/obdemo/wsgi/obdemo.wsgi
WSGIDaemonProcess obdemo_org user=openblock group=www-data processes=10 threads=1
WSGIProcessGroup obdemo_org
```

```
CustomLog /var/log/apache2/openblock-access.log combined
ErrorLog /var/log/apache2/openblock-error.log
</VirtualHost>
```

5.2.8 Maintaining an OpenBlock Site

Upgrades

Check the Release Notes

Database Migrations

When upgrading your copy of the OpenBlock code, there may sometimes be updates to Model code which require corresponding changes to your existing database.

You can do this with one command (it's prudent to make a database backup first):

```
django-admin.py syncdb --migrate
```

To see what migrations exist and which ones you've already run, you can do:

```
django-admin.py migrate --list
```

Under the hood, this uses South to automate these database migrations. If you really want to see what the migrations do, or need to write your own, you'll want to read the South documentation to understand how they work. Then look in the `migrations/` subdirectories located under the various app directories, notably `ebpub/ebpub/db/migrations/` to see what the existing migration scripts look like.

5.2.9 Indices and tables

- search

5.3 Packages

OpenBlock consists of a number of packages, summarized below:

5.3.1 Main Code Packages

obdemo

The obdemo package contains code, templates, and configuration specific to http://demo.openblockproject.org.

They are intended to serve as a useful example of how to set up a site based on the OpenBlock code.

By default, the site is set up to use Boston as the default location for the maps. You can change that by tweaking settings.py, but then you're on your own for finding local data to load.

How The Demo Works

obdemo uses the following parts of the OpenBlock codebase:

- ebpub does the heavy lifting, providing all the view and model code. We also use the base templates, scripts, and css from here, although we override a few templates.
- ebdata is used to feed news data into the system.
- obadmin obadmin provides the administrative interface, the "oblock" setup command that we use for installation and bootstrapping. It also provides a custom test runner (called as usual by `manage.py test`).
- obdemo itself is a thin wrapper around the other packages and provides some data fixtures, migrations, and scripts to bootstrap the data used for the Boston demo.

For the maps, we use a free base layer based on Open Street Map and hosted by OpenGeo. Consequently, we don't need ebgeo or Mapnik.

We don't currently use any of the other openblock-extras packages (ebblog, ebwiki, everyblock, or ebinternal).

Deployment

See Deployment

ebpub

Publishing system for block-specific news, as used by EveryBlock.com.

Before you dive in, it's *highly* recommend you spend a little bit of time browsing around EveryBlock.com to get a feel for what this software does.

Also, for a light conceptual background on some of this, particularly the data storage aspect, watch the video "Behind the scenes of EveryBlock.com" here: http://blip.tv/file/1957362

Settings

ebpub requires a smorgasbord of eb-specific settings in your settings file. If you follow the Creating a Custom Site Based on OpenBlock or Installing and Setting Up the Demo Site directions, they provide suitable settings files that you can adjust as needed. Otherwise, you might just start with the file ebpub/settings.py and tweak that (or import from it in your own settings file). The application won't work until you set the following:

```
DATABASE_USER
DATABASE_NAME
DATABASE_HOST
DATABASE_PORT
SHORT_NAME
PASSWORD_CREATE_SALT
PASSWORD_RESET_SALT
METRO_LIST
EB_MEDIA_ROOT
EB_MEDIA_URL
EB_DOMAIN
DEFAULT_MAP_CENTER_LON
DEFAULT_MAP_CENTER_LAT
DEFAULT_MAP_ZOOM
```

Models

Broadly speaking, the system requires two different types of data: geographic boundaries (Locations, Streets, Blocks and Intersections) and news (Schemas and NewsItems).

Geographic Models

See also Loading Geographic Data

LocationTypes / Locations A `Location` is a polygon that represents a geographic area, such as a specific neighborhood, ZIP code boundary or political boundary. Each `Location` has an associated `LocationType` (e.g., "neighborhood"). To add a Location to the system, follow these steps:

1. Create a row in the "db_locationtype" table that describes this LocationType. See the LocationType model code in `ebpub/db/models.py` for information on the fields and what they mean.
2. Get the Location's geographic representation (a set of longitude/latitude points that determine the border of the polygon). You might want to draw this on your own using desktop GIS tools or online tools, or you can try to get the data from a company or government agency.
3. With the geographic representation, create a row in the "db_location" table that describes the Location. See the Location model code in `ebpub/db/models.py` for information on the fields and what they mean.

   You can create them in various ways: use the admin UI; use the script `add_location` to create one by specifying its geometry in well-known text (WKT) format; use the script `import_locations` to import them from shapefiles; or use the Django model API; or do a manual SQL INSERT statement.

You'll need to create at least one LocationType with the slug "neighborhoods", because that's hard-coded in various places throughout the application.

Blocks A Block is a segment of a single street between one side street and another side street. Blocks are a fundamental piece of the ebpub system; they're used both in creating a page for each block and in geocoding.

Blocks are stored in a database table called "blocks". To populate this table, follow these steps:

1. Obtain a database of the streets in your city, along with each street's address ranges and individual street segments. If you live in the U.S.A. and your city hasn't had much new development since the year 2000, you might want to use the U.S. Census' TIGER/Line file (http://www.census.gov/geo/www/tiger/).
2. Import the streets data into the "blocks" table. ebpub provides two pre-made import scripts:
   - If you're using TIGER/Line data, you can use the script `import_blocks_tiger` which should be on your $PATH.
   - If you're using data from ESRI, you can use the script `ebpub/streets/blockimport/esri/importers/blocks`
   - If you're using data from another source, take a look at the Block model in `ebpub/streets/models.py` for all of the required fields.

Streets and Intersections The ebpub system maintains a separate table of each street in the city. Once you've populated the blocks, you can automatically populate the streets table by running the importer `ebpub/streets/populate_streets.py`.

The ebpub system also maintains a table of each intersection in the city, where an intersection is defined as the meeting point of two streets. Just like streets, you can automatically populate the intersections table by running the code in `ebpub/streets/populate_streets.py`.

Streets and intersections are both necessary for various bits of the site to work, such as the "browse by street" navigation and the geocoder (which supports the geocoding of intersections).

Once you've got all of the above geographic boundary data imported, you can verify it on the site by going to /streets/ and /locations/.

NewsItems and Schemas

Next, it's time to start adding news. The ebpub system is capable of handling many disparate types of news – e.g., crime, photos and restaurant inspections. Each type of news is referred to as a `Schema`.

To add a new Schema, add a row to the "db_schema" database table or use the Django database API. See the Creating a Custom NewsItem Schema documentation, or see the `Schema` model in `ebpub/db/models.py` for information on all of the fields

NewsItems A `NewsItem` is broadly defined as "something with a date and a location." For example, it could be a building permit, a crime report, or a photo. NewsItems are stored in the "db_newsitem" database table, and they have the following fields:

schema the associated Schema object

title the "headline"

description an optional blurb describing what happened

url an optional URL to another Web site

pub_date the date and time this NewsItem was added to the OpenBlock site

item_date the date (without time) of the object (e.g. the date the news occurred, or failing that, the date it was published on the original source site)

location the location of the object (a GeoDjango GeometryField, usually a Point)

location_name a textual representation of the location, eg. an address or place name

location_object an optional associated Location object

block an optional associated Block object

attributes extensible metadata, described in the section on *SchemaFields and Attributes*.

map_icon_url A URL (can be relative) to an image to use as this news type's icon on maps. Should be roughly 40x40 pixels. Optional.

map_color A color hex code (eg. #FF0000) to use for marking this news type on maps. Only used if map_icon_url is not provided. Optional.

The difference between `pub_date` and `item_date` might be confusing. The distinction is intended for data sets where there's a lag in publishing or where the data is updated infrequently or irregularly. For example, on EveryBlock.com, Chicago crime data is published a week after it is reported, so a crime's item_date is the day of the crime report whereas the pub_date is the day the data was published to EveryBlock.com (generally seven days after the item_date).

Similarly, `location_object` and `location` can be confusing. `location_object` is used rarely; a good use case would be some police blotter reports which don't provide precise location information for a news item other than which precinct it occurs in. In this case, you'd want a LocationType representing precincts, and a Location for each precinct; then, when creating a NewsItem, set its `location_object` to the relevant Location, and don't set `location` or `block` at all. For a live example, see http://nyc.everyblock.com/crime/by-date/2010/8/23/3364632/

NewsItemLocations This model simply maps any number of NewsItems to any number of Locations. The rationale is that locations may overlap, so a NewsItem may be relevant in any number of places. Normally you don't have to worry about this: there are database triggers that update this table whenever a NewsItem's location is set or updated.

SchemaFields and Attributes The NewsItem model in itself is generic – a lowest-common denominator of each NewsItem on the site. If you'd like to extend your NewsItems to include Schema-specific attributes, you can use SchemaFields and Attributes.

A single NewsItem is described by one NewsItem instance, one corresponding Attribute instance which is a dictionary-like object containing metadata, and one Schema that identifies the "type" of NewsItem.

The Schema in turn is described by a number of SchemaFields which describe the meaning of the values of Attribute dictionaries for this type of NewsItem.

Given an appropriate Schema, using this to get/set attributes on NewsItems is trivial - it's just like a dictionary. To assign the whole dictionary:

```
ni = NewsItem.objects.get(...)
ni.attributes = {'some_schemafield_name': 'some value'}
# There is no need to call ni.save() or ni.attributes.save();
# the assignment operation does that behind the scenes.
```

To assign a single value:

```
ni.attributes['some_schemafield_name'] = 'some other value'
# Again there is no need to save() anything explicilty.
```

To get a value:

```
print ni.attributes['some_schemafield_name']
```

Or, from a database perspective: The "db_attribute" table stores arbitrary attributes for each NewsItem, and the "db_schemafield" table is the key for those attributes. A SchemaField says, for example, that the "int01" column in the db_attribute table for the "real estate sales" Schema corresponds to the "sale price".

This can be confusing, so here's an example. Say you have a "real estate sales" Schema, with an id of 5. Say, for each sale, you have the following information:

address

sale date

sale price

property type (single-family home, condo, etc.)

The first two fields should go in NewsItem.location_name and NewsItem.item_date, respectively – there's no reason to put them in the Attribute table, because the NewsItem table has a slot for them.

Sale price is a number (we'll assume it's an integer), so create a SchemaField defining it:

schema_id = 5 The id of our "real estate sales" schema.

name = 'sale_price' The alphanumeric-and-underscores-only name for this field. (Used in URLs.)

real_name = 'int01' The column to use in the db_attribute model. Choices are: int01-07, text01, bool01-05, datetime01-04, date01-05, time01-02, varchar01-05. This value must be unique with respect to the schema_id.

pretty_name = 'sale price' The human-readable name for this attribute.

pretty_name_plural = 'sale prices' The plural human-readable name for this attribute.

display = True Whether to display the value on the site.

is_lookup = False Whether it's a lookup. (Don't worry about this for now; see the Lookups section below.)

is_filter = False Whether it's a filter. (Again, don't worry about this for now.)

is_charted = False Whether it's charted. (Again, don't worry.)

display_order = 1 An integer representing what order it should be displayed in on newsitem_detail pages.

is_searchable = False Whether it's searchable. This only applies to textual fields (varchars and texts). Don't use with Lookups.

Once you've created this SchemaField, the value of "int01" for any db_attribute row with schema_id=5 will be the sale price.

Python code using this Schema is the easy part; you can write things like this:

```
from ebpub.db.models import NewsItem
ni = NewsItem(schema__id=5, title='the title', description='the description', ...)
ni.save()
ni.attributes = {'sale_price': 59, ...}
```

You can then search for items with the same price like so:

```
NewsItem.objects.filter(schema__id=5).by_attribute(sale_price=59)
```

The `by_attribute` method is particular to NewsItems and allows searching for NewsItem by Attribute values.

Lookups Lookups are a normalized way to store attributes that have only a few possible values.

Consider the "property type" data we have for each real estate sale NewsItem in the example above. We could store it as a varchar field (in which case we'd set real_name='varchar01') – but that would cause a lot of duplication and redundancy, because there are only a couple of property types – the set ['single-family', 'condo', 'land', 'multi-family']. To represent this set, we can use a Lookup – a way to normalize the data.

To do this, set `SchemaField.is_lookup=True` and make sure to use an 'int' column for SchemaField.real_name. Then, for each record, get or create a Lookup object (see the model in `ebpub/db/models.py`) that represents the data, and use the Lookup's id in the appropriate db_attribute column. The helper function `Lookup.objects.get_or_create_lookup()` is a convenient shortcut here (see the code/docstring of that function).

Many-to-many Lookups Sometimes a NewsItem has multiple values for a single attribute. For example, a restaurant inspection can have multiple violations. In this case, you can use a many-to-many Lookup. To do this, just set `SchemaField.is_lookup=True` as before, but use a varchar field for the `SchemaField.real_name`. Then, in the db_attribute column, set the value to a string of comma-separated integers of the Lookup IDs.

Charting and filtering lookups Set `SchemaField.is_filter=True` on a lookup SchemaField, and the detail page for the NewsItem (newsitem_detail) will automatically link that field to a page that lists all of the other NewsItems in that Schema with that particular Lookup value.

Set `SchemaField.is_charted=True` on a lookup SchemaField, and the detail page for the Schema (schema_detail) will include a chart of the top 10 lookup values in the last 30 days' worth of data. Similar charts are on the place detail overview page. (This assumes aggregates are populated; see the Aggregates section below.)

Aggregates Several parts of ebpub display aggregate totals of NewsItems for a particular Schema. Because these calculations can be expensive, there's an infrastructure for caching the aggregate numbers regularly in separate tables (db_aggregate*).

To do this, just run `update_aggregates` on the command line.

You'll want to do this on a regular basis, depending on how often you update your data. Some parts of the site (such as charts) will not be visible until you populate the aggregates.

Event-like News Types In order for OpenBlock to treat a news type as being about (potentially) future events, rather than news from the (recent) past, there is a simple convention that you should follow:

1. Set the schema's `is_event=True`.

2. Add a SchemaField with `name='start_time'`. It should be a Time field, i.e. `real_name` should be one of `time01`, `time02`, etc. Leave `is_filter`, `is_lookup`, `is_searchable`, and `is_charted` set to False. The `pretty_name` can be whatever you like of course.

3. Optionally add a SchemaField with `name='end_time'`, if your data source will include this information.

4. When adding NewsItems of this type, the NewsItem's `item_date` field should be set to the date on which the event will (or already did) take place, and the `start_time` attribute should be set to the (local) time it will start, and the `end_time` attribute should be set to the (local) end time if known.

All-day events can be represented by leaving `start_time` empty.

There is no special support for repeating events or other advanced calendar features.

Site views/templates

Once you've gotten some data into your site, you can use the site to browse it in various ways. The system offers two primary axes by which to browse the data:

- By schema – starting with the schema_detail view/template
- By place – starting with the place_detail view/template (where a "place" is defined as either a Block or Location)

Note that default templates are included in ebpub/templates. At the very least, you'll want to override base.html to design your ebpub-powered site. (The design of EveryBlock.com is copyrighted and not free for you to use; but the default templates, css, and images that ship with OpenBlock and ebpub are of course free for your use under the same license terms as the rest of OpenBlock (GPL)).

Custom NewsItem lists When NewsItems are displayed as lists, generally templates should use the newsitem_list_by_schema custom tag. This tag takes a list of NewsItems (in which it is assumed that the NewsItems are ordered by schema) and renders them through separate templates, depending on the schema. These templates should be defined in the ebpub/templates/db/snippets/newsitem_list directory and named [schema_slug].html. If a template doesn't exist for a given schema, the tag will use the template ebpub/templates/db/snippets/newsitem_list.html.

We've included two sample schema-specific newsitem_list templates, news-articles.html and photos.html.

It is also possible to customize the html used in map popups for each schema, by creating a snippet named newsitem_popup_[schema_slug].html in a subdirectory richmaps/ on your template path. If no such template exists, the default is `ebpub/richmaps/templates/richmaps/newsitem_popup.html`.

Custom NewsItem detail pages Similarly to the newsitem_list snippets, you can customize the newsitem_detail page on a per-schema basis. Just create a template named [schema_slug].html in ebpub/templates/db/newsitem_detail. See the template ebpub/templates/db/newsitem_detail.html for the default implementation.

Custom Schema detail pages To customize the schema_detail page for a given schema, create a `templates/db` subfolder in your app, and add a template named `[schema_slug].html` in that directory. See the template `ebpub/templates/db/schema_detail.html` for the default generic implementation.

E-mail alerts

Users can sign up for e-mail alerts via a form on the place_detail pages. To send the e-mail alerts, just run the `send_all()` function in `ebpub/alerts/sending.py`. You probably want to do this regularly by Running Scrapers.

Accounts

This system uses a customized version of Django's User objects and authentication infrastructure. ebpub comes with its own User object and Django middleware that sets request.user to the User if somebody's logged in.

ebdata

Code to help write scripts that import/crawl/parse data from the web into ebpub, as well as extract addresses from (English) text.

Scraper scripts will probably be built on either *ebdata.retrieval* or *ebdata.blobs*, depending on the type of content being scraped.

ebdata.blobs

The blobs package is a Django app responsible for crawling, scraping, extracting, and geocoding news articles from the web.

It is best suited for scraping "unstructured" websites that don't have machine-readable feeds, eg. for scraping raw HTML and/or binary file formats such as PDF or Excel. (For sites that provide RSS or Atom feeds, and/or an API, the *ebdata.retrieval* package may be more suitable.) (For dealing with binary file formats, you'll also want to look into the *ebdata.parsing* package.)

Many examples can be found in the everyblock package.

The blobs app contains two models, `Seed` and `Page`. `Seed` is a news source, like the Chicago Tribune, and a `Page` is a particular html page that was crawled from a Seed.

TODO: This really needs more explanation.

ebdata.nlp

The nlp package contains utilities for detecting locations in text. This package is used by *ebdata.blobs*, but if you want to use it directly, check out the docstrings for the functions in `ebdata.parsing.addresses.`

ebdata.parsing

The parsing package contains helpers for reading different file types.

The `dbf`, `excel`, `mdb`, and `unicodecsv` modules are for reading stuctured data, and generally follow the python csv reader api. See the code for more details on how to use them.

The pdf module is for converting pdf to text, and requires Xpdf. http://www.foolabs.com/xpdf/download.html

ebdata.retrieval

The retrieval package contains a framework for writing scrapers for structured data. Some examples can be found in `ebdata/ebdata/scrapers/`. There are more (unmaintained) examples of how to use this framework in different situations in the everyblock package.

(For scraping data from unstructured sites, eg. sites that lack feeds or machine-consumable API, it may be better to build on the *ebdata.blobs* package.)

The most commonly used scraper base class is the `NewsItemListDetailScraper`. It handles scraping list/detail types of sites, and creating or updating NewsItem objects. "List" could be an RSS or Atom feed, or an HTML index, which links to "detail" pages; these can be any format, such as HTML, XML, or JSON. (In some cases, the feed provides all the necessary information, and there's no need to retrieve any detail pages.)

Generally, to run a scraper, you need to instantiate it, and then call its `update()` method. Sometimes the scraper will take arguments, but it varies on a case-by-case basis; see the scrapers in `ebdata/ebdata/scrapers` for examples. You can also run a scraper by calling its `display_data()` method. This will run the scraper, but won't actually save any of the scraped data. It's very useful for debugging, or when writing a scraper for the first time.

All of the methods and parameters you'll need to use are documented in docstrings of `ebdata.retrieval.scrapers.list_detail.ListDetailScraper` and in `ebdata.retrieval.scrapers.newsitem_list_detail.NewsItemListDetailScraper`. `ListDetailScraper` is a base class that handles scraping, but doesn't actually have any methods for saving data.

The retrieval package also contains `updaterdaemon`, which is a (*deprecated*) cron-like facility for running scrapers. It comes with a unix-style init script, and its configuration and examples are in `ebdata/retrieval/updaterdaemon/config.py`. More documentation at Running Scrapers.

ebdata.scrapers

A collection of ready-to-run scraper scripts, with JSON fixture files for loading the schemas needed by each scraper.

(If you want to write your own scrapers for other data sources, see Data Scraper Tutorial.)

These generally leverage the tools in ebdata.retrieval.

All of them can be run as command-line scripts. Use the `-h` option to see what options, if any, each script takes.

Flickr: ebdata.scrapers.general.flickr Loads Flickr photos that are geotagged at a location within your configured *metro extent*.

You must set both `settings.FLICKR_API_KEY` and `settings.FLICKR_API_SECRET`.

You must also install a library that it depends on:

```
$ $VIRTUAL_ENV/bin/pip install flickrapi
```

(Note that if obdemo is installed, you should already have this library.)

The scraper script is `PATH/TO/ebdata/scrapers/general/flickr/flickr_retrieval.py` and the schema can be loaded by doing `django-admin.py loaddata PATH/TO/ebdata/scrapers/general/flickr/photos_schema.json`.

GeoRSS: ebdata.scrapers.general.georss Loads any RSS or Atom feed. It tries to extract a point location and a location name from any feed according to the following strategy:

- First look for a GeoRSS point.

- If no point is found, look for a location name in standard GeoRSS or xCal elements; if found, geocode that.
- If no location name is found, try to find addresses in the title and/or description, and geocode that.
- If a point was found, but a location name was not, try to reverse-geocode the point.
- If all of the above fail, skip this item.

The scraper script is `PATH/TO/ebdata/scrapers/general/georss/retrieval.py` and a generic "local news" schema can be loaded by doing `django-admin.py loaddata PATH/TO/ebdata/scrapers/general/georss/local_news_schema.json`.

Meetup: ebdata.scrapers.general.meetup Retrieves upcoming Meetups from meetup.com. USA-only. This assumes you have loaded some *US ZIP Codes*, as it will attempt to load meetups for each zip code in turn.

You will need to get an API key, and set it as `settings.MEETUP_API_KEY`.

The scraper script is `PATH/TO/ebdata/scrapers/general/meetup/meetup_retrieval.py` and the schema can be loaded by doing `django-admin.py loaddata PATH/TO/ebdata/scrapers/general/meetup/meetup_schema.json`.

This scraper may take hours to run, since Meetup's API has a rate limit of 200 requests per hour (returning up to 200 meetups each), and a large city may have thousands of meetups every day, and we're trying to load all scheduled meetups for the next few months. The default behavior is to run until the API's rate limit is hit, then wait till the limit is lifted (typically 1 hour), and repeat until all pages for all zip codes have been loaded. If you'd rather do smaller batches, try the `--help` option to see what options you have.

Open311 / GeoReport: ebdata.scrapers.general.open311 A scraper for the Open311 / GeoReport API that is being adopted by a growing number of cities including many served by *SeeClickFix <http://seeclickfix.com>*. (Tip: You can get an open311 endpoint for *any* location served by seeclickfix, not just those listed on that page, by passing `http://seeclickfix.com/<location-name>/open311/v2/` as the API URL.)

It has many command-line options for passing API keys and so forth; run it with the `--help` option.

The scraper script is `PATH/TO/ebdata/scrapers/general/open311/georeportv2.py` and a suitable schema can be loaded by doing `django-admin.py loaddata PATH/TO/ebdata/scrapers/general/open311/open311_service_requests_schema.json`.

SeeClickFix: ebdata.scrapers.general.seeclickfix A scraper for issues reported to SeeClickFix. Note you can also use the Open311 / GeoReport scraper described above, since SeeClickFix supports the GeoReport API as well; we have both scrapers because the SeeClickFix native API has been around longer.

Pass the city and state as command-line arguments.

The scraper script is `PATH/TO/ebdata/scrapers/general/seeclickfix/seeclickfix_retrieval.py` and a suitable schema can be loaded by doing `django-admin.py loaddata PATH/TO/ebdata/scrapers/general/seeclickfix/seeclickfix_schema.json`.

ebdata.scrapers.us Scrapers for specific city data sources in the USA. Currently this includes only scrapers for Boston, MA:

- ebdata/scrapers/us/ma/boston/building_permits/
- ebdata/scrapers/us/ma/boston/businesses/
- ebdata/scrapers/us/ma/boston/events/
- ebdata/scrapers/us/ma/boston/police_reports/

- ebdata/scrapers/us/ma/boston/restaurants/

Many of these are used for http://demo.openblockproject.org. For more information, see the source of each script.

ebdata.templatemaker

The templatemaker package contains utilities for detecting the actual content given a set of html pages that were generated from a template. For instance, templatemaker helps detect and extract the actual article from a page that could also contain navigation links, ads, etc.

This is used internally by *ebdata.blobs*. It is not typically used directly by scraper scripts.

ebdata.textmining

The textmining package contains utilities for preprocessing html to strip out things that templatemaker doesn't care about like comments, scripts, styles, meta information, etc. It is used by *ebdata.templatemaker* but may also be used directly by scraper scripts.

obadmin

Administrative UI, installation and utilities package for OpenBlock.

The oblock command

`oblock` is a command-line program that gets installed with obadmin. It provides commands to help install and manage OpenBlock and its database(s). This is purely optional, but the demo setup script uses it quite a bit.

Try running `oblock help` to see all the available commands. Each "task" can be run as a subcommand, like `oblock sync_all`.

obdemo

The code and configuration used by http://demo.openblockproject.org. This is useful as an example of how to set up your own site based on OpenBlock, and is a great place to start. It primarily uses the ebpub package, and is set up with Boston, MA as the area of interest.

For more information, see obdemo.

ebpub

Publishing system for block-specific news, as used by EveryBlock.com.

This is the core of an OpenBlock site, providing the web interface that users see as well as the underlying data models. You need this.

For more information, see ebpub.

ebdata

Code to help write scripts that crawl/parse/import data into ebpub.

You will probably need to write such scripts to get OpenBlock to do anything useful; that is the main way you feed local news into the system.

For more information, see ebdata.

obadmin

Administrative UI, installation and utilities package for OpenBlock.

For more information, see obadmin.

5.3.2 Other Packages

There are several open-source packages originally released by the EveryBlock.com team in 2009, but not actively used or maintained by the OpenBlock core developers. They have been moved out of OpenBlock itself and into https://github.com/openplans/openblock-extras

5.4 Changes

5.4.1 OpenBlock 1.2.0 (UNRELEASED)

Upgrade Notes

- As usual, install all dependencies, eg if you are upgrading a source checkout:

```
pip install -r ebpub/requirements.txt
pip install -e ebpub
pip install -r ebdata/requirements.txt
pip install -e ebdata
pip install -r obadmin/requirements.txt
pip install -e obadmin
pip install -r obdemo/requirements.txt
pip install -e obdemo
```

- As usual, sync and migrate the database:

```
django-admin.py syncdb
django-admin.py migrate
```

Backward Incompatibilities

- Removed ebdata/retrieval/scrapers/new_newsitem_list_detail.py, which wasn’t used anywhere.

New Features in 1.2

- Optional ReCaptcha on the user-contributed (“Neighbornews”) add/edit forms.
- User-contributed content (“neighbornews”) now has edit and delete forms.

- Sensible defaults on most DateFields and DateTimeFields, can still be overridden.
- Logout now redirects you to whatever page you were viewing.
- Add a “properties” JSON field to the Profile model, for more flexible per-user metadata.
- User admin UI now shows Profiles and API keys inline.
- “Sticky widgets” aka “pinned” NewsItems in widgets: You can use the admin UI to make certain NewsItems stay visible in the widget permanently or until an expiration date that you set.

Bugs fixed

- Fixed double-logging of scrapers to the console.
- /streets/ list doesn’t blow up if you haven’t set DEFAULT_LOCTYPE_SLUG.
- Workaround for getting profile when request.user is a LazyUser instance.
- De-hardcoded more URLs.
- When using a too-old python version, our setup.py scripts now give a more informative error, instead of SyntaxError due to a *with* statement.
- Custom login view now works when going to admin site. Ticket #174
- Logout form was broken by bad template name. Fixed.

Documentation

- Added docs on all the settings in settings_default.py.
- Better documentation about Schemas, Attributes, and how they relate.

Other

- Moved some NewsItemListDetailScraper functionality up into BaseScraper, so it’s more widely usable.

Older Changes

See OpenBlock change history.

5.4.2 OpenBlock change history

For more recent changes, see OpenBlock 1.2.0 (UNRELEASED).

OpenBlock 1.1.0 (October 20, 2011)

Upgrade Notes

- Due to an oversight, both obdemo and the generated custom applications prior to this release didn’t include the TIME_ZONE setting. Please set settings.TIME_ZONE to an appropriate value.
- As usual, install all dependencies, eg if you are upgrading a source checkout:

```
    pip install -r ebpub/requirements.txt
    pip install -e ebpub
    pip install -r ebdata/requirements.txt
    pip install -e ebdata
    pip install -r obadmin/requirements.txt
    pip install -e obadmin
    pip install -r obdemo/requirements.txt
    pip install -e obdemo
```

- As usual, sync and migrate the database:

```
    django-admin.py syncdb
    django-admin.py migrate
```

Backward Incompatibilities

- `updaterdaemon` is now deprecated. It has not been removed, but we no longer recommend or support its use. Use cron or your favorite cron alternative instead for running scrapers regularly. See the "Running Scrapers" docs. This allows us to close a bunch of tickets (#194, #51, #87, #180, #199, #222).
- removed redundant `get_metro_bbox function`.
- remove unused template tags: SHORT_NAME, STATE_ABBREV, EB_SUBDOMAIN.

New Features in 1.1

- Big shareable maps: "Explore these items on a larger map" link on all type-specific news lists. For example, http://demo.openblockproject.org/photos/filter/locations=neighborhoods,financial-district/ links to http://bit.ly/njmZT6 which is shareable via permalink. (There is also undocumented support for embedding these via iframes.)
- Comments on NewsItems. Requires logging in, and the Schema must have allow_comments=True and has_detail=True. Needs docs.
- User-contributed "Neighbor Messages" and "Neighbor Events" news types, in the ebpub.neighbornews package. Needs docs.
- Better support for running in a multi-city area:
  - new get_city_locations() function to get a list of all Locations whose LocationType matches the 'city_location_type' from settings.METRO_LIST.
  - `--fix-cities` option to block import scripts (and admin UI) that allows fixing imported blocks so block.city matches an existing overlapping city-ish Location.
  - clean out intersections and streets on import, so they're regenerated safely. Optionally skip regeneration.
  - some related URL bugfixes.
- Import Places from a CSV file via the admin UI. Needs docs.
- Date and time picker widgets on forms, where relevant. (#186)
- Block import supports filtering by your default metro extent, not just city name. #160
- Support for future events, not just recent news. Several scrapers support this: the ma/boston/events scraper, and the general/meetups/ scraper, and the neighbornews package. See docs in docs/packages/ebdata.rst. (Ticket #246)

- Added a scraper for Meetup.com, in ebdata/scrapers/general/meetups. It's zero-configuration: it just loops over your zip codes and finds all meetups for those. It's at ebdata/scrapers/general/meetup/meetup_retrieval.py and the associated schema can be loaded like so: `django-admin.py loaddata ebdata/scrapers/general/meetup/meetup_schema.json` You'll need to set `MEETUP_API_KEY` in settings.py. (Ticket #208)
- Add a –reset option to `update_aggregates` script, deletes all aggregates and starts over. (ticket #221)
- Add an `ebpub/bin/delete_newsitems.py` script, useful during schema development: wipes all newsitems and attributes and lookups of a given schema.
- Also add –quiet, –verbose, –dry-run, and –help command-line options to `update_aggregates`.
- Email alerts can now be sent via a command-line script. (related to ticket #65). Includes docs for how to set it up with cron.
- Email alert signup can be disabled by removing 'ebpub.alerts' from settings.INSTALLED_APPS. (refs ticket #65).
- obdemo includes flickr and meetup in default news types.
- Flickr scraper (ticket #26). It's at ebdata/scrapers/general/flickr/flickr_retrieval.py and the associated schema can be loaded like so: `django-admin.py loaddata ebdata/scrapers/general/flickr/photos_schema.json` You'll need to set `FLICKR_API_KEY` and `FLICKR_API_SECRET` in settings.py.
- Import locations from shapefiles in the admin UI (ticket #59). With documentation (ticket #234).
- Import blocks from shapefiles in the admin UI. Also populates streets, blockintersections, and intersections. (ticket #215)
- You can now set the default location type via settings.DEFAULT_LOCTYPE_SLUG. (#148)
- Add –verbose and –quiet options to a bunch of command-line scripts and scrapers.
- Don't email scraper errors by default. That's just not nice, and cron already does that.
- All provided scrapers now log to settings.SCRAPER_LOGFILE_NAME.
- Custom apps generated via `paster create -t openblock` now include a wsgi file for use with mod_wsgi, an alternative settings file for use with django-admin process_tasks, a skeleton cron config, executable manage.sh and manage.py files. Also, manage.sh is now better at automatically finding and activating the virtualenv.
- obdemo also includes an example cron config file, a manage.sh file, and the alt. settings file. And no longer has an example updaterdaemon config.
- Our Amazon EC2 AMI will now use cron rather than updaterdaemon. Lots of other fixes in the EC2 scripts too.

Bugs fixed

- Fixed broken map on feeds page, ticket #237.
- Added missing links to the password change form.
- CSRF protection everywhere, ticket #185. (As a side-effect we are now using JQuery 1.5.2.)
- Block import: generated names now sort numerically correctly (eg. "12-100 Main St" rather than "100-12 Main St")
- Block import: Don't try to guess right_from, right_to if not provided; that typically means there really is nothing on that side of the street.

- Boston demo: restaurant inspections scraper fixed to accomodate markup changes.
- De-hardcoded “neighborhoods” from various URLs. (#148)
- Zip code import UI has no default state (to avoid selecting Alabama by mistake).
- Zip code import now sets creation date (#233)
- Removed confusing NewsItem “About” page. (#228)
- Removed map from NewsItem list in admin UI, was too slow. (#219)
- SavedPlace now enforces that it has either a Block or a Location but not both. (#213)
- Items shown on map on schema filter page now use same filters as the items on the page. (#121)
- Support 2010 US Census tiger files (ticket #147). Use them for the Boston demo.
- Georeport / open311 scraper: support unofficial ‘page’ parameter (ticket #245); also, use the ‘address’ field for location_name if provided.
- Seeclickfix scraper: allow city & state params, don’t hardcode to boston; ticket #243.
- place_detail_overview wasn’t actually filtering by place.
- ajax date charts would blow up if no results found.
- Fix ticket #77: Now filtering news by item_date instead of pub_date since that’s the date that’s shown and used for aggregates.
- Fix “show/hide” buttons on place detail page and account page. (tickets #204, #115, 236)
- Fixed bug that caused many “Unknown” locations in location charts. (ticket #192). And removed “unknowns” entirely from the chart.
- Locations weren’t capitalized on some pages. (ticket #202)
- Several bounds-related errors in Location import fixed (thanks to Bret Walker).
- Scrapers that create timezone-aware datetimes no longer blow up.
- GeoReport scraper: scrape a reasonable amount of days, not 60 every darn time. And do pagination (ticket #245)
- Georss scraper: Had the forwards / backwards coordinate test reversed :-
- Georss scraper: Skip items with no location_name.
- Fix some migration ordering bugs.
- parse_date no longer blows up if you feed it a date or datetime instance.
- CSS fixes for ajax date charts on location overview page.

Documentation

- Lots more docs about loading geographic data.
- Document email configuration. (ticket #205)
- Document what you get when doing `paster create -t openblock`.
- More docs about running on Amazon EC2.
- Describe differences from Everyblock
- More help_text added to several Model fields, so admin UI is slightly more self-documenting.

- Many many minor updates and tweaks.

Other

- Upgraded to OpenLayers 2.11. (ticket #250)
- Upgraded to Django 1.3.1.
- Upgraded to JQuery 1.5.2.
- Removed some unused template tags (SHORT_NAME, STATE_ABBREV, EB_SUBDOMAIN).
- Removed old version of map popups code.

OpenBlock 1.0.1 (Sept 7 2011)

This was a minor bugfix (and docs) release, and was mostly identical to 1.0.0.

- The georss scraper now gets coordinates in the right order on the first try, and populates location_name if it falls back to geocoding.
- Fix date formatting on newsitem-detail page. (ticket #201)
- The `import_blocks_tiger` and `import_blocks_esri` scripts had a circular import.
- Fix a broken doctest in bootsrap.py.
- Better handling of unicode in ebdata.textmining.

Documentation

- Added docs for cloning an EC2 instance from our Amazon AMI.
- Document the scrapers that ship out-of-the-box with ebdata.
- Remove nonexistent `--city` option from geodata docs.
- Changed docs theme for easier navigation.

OpenBlock 1.0.0 (August 30, 2011)

Upgrade Notes

- If you originally installed, or already upgraded to, OpenBlock 1.0 Beta, you may skip the rest of this section.
- If you have an existing database that was built with 1.0a1 or earlier, you'll need to run this command to deal with the removal of the "django-apikey" dependency:

```
django-admin.py migrate apikey 0001 --fake
```

- Many data-loading scripts that were scattered all over the source tree are now installed into your environment's `bin` directory, so they should be on your `$PATH`. Documentation has been updated accordingly.
- As usual, you should always run after upgrading:

```
django-admin.py syncdb --migrate
```

If you were unlucky and had last migrated with a git checkout including migrations that later got renamed or removed, you may get errors from migrating. In that case try adding the `--delete-ghost-migrations` option.

- Production webserver configurations will need a line added to get the django-olwidget javascript and CSS to show up. For example, for Apache you'd add a line like (adjust path as needed):

```
Alias /olwidget/ /home/openblock/openblock/src/django-olwidget/
```

- We now require Django 1.3. This probably doesn't have any impact on you. (ticket #155).
- Settings changes:
  - MAP_BASELAYER_TYPE can now be any base layer supported by olwidget, eg. "google.streets". (Some require other settings for eg. API keys; see ebpub/settings_default.py for comments and examples.)
  - You can add custom base layers to your maps by creating the dictionary settings.MAP_CUSTOM_BASE_LAYERS. See ebpub/settings_default.py for an example.

    This replaces the WMS_URL setting from openblock 1.0a1 which is no longer supported.

New Features in 1.0

- ticket #33: Different map icons for different news item types. To use this, you can use the admin UI to configure "map icon url" or "map color" for a Schema.
- ticket #85: Added `streets.PlaceType` model for categorizing `Places`. These also can have individual colors or icon URLs on the /maps/ view. (Original ticket title was "'Landmark' location type")
- ticket #142: JSON push API for news items. See docs/main/api.rst
- ticket #187: REST API standard features: API key provisioning; require keys (or auth) for POST / DELETE; throttling
- Import US Zip Codes as Locations, via the admin UI.
- Work-in-progress: user-submitted content. See code in the ebpub/neighbornews app.
- Work-in-progress: Maps you can share just by copy/pasting a URL. For a sneak preview, browse to /maps/.
- Much better admin UI maps. (ticket #140: Bad admin UI for GeometryFields)
- ticket #72: unify NewsItem.attributes and NewsItem.attribute_values
- ticket #52: Proper validation for Street Misspellings in admin
- ticket #157: fill in normalized name automatically
- ticket #123: Configurable base layer should apply to admin UI maps too

Bug fixes

This is not a complete list; not all bugs fixed in this release were ticketed.

- Fix #172: schema_detail view blows up (TypeError) if there are no NewsItems in the last 30 days, but there is a matching AggregateLocation. (That shouldn't happen, but evidently did with some boston demo schemas; also fixed a related possible off-by-one error that may have been a factor.)
- Schema filter page: don't say 'You might want to try...' if there's nothing to try.
- Fix bug where scrapers that create timezone-aware datetimes blow up

- Fix errors in bounds checking in location importers, thanks to Bret Walker.
- Fix missing import in Place admin
- Fixed several bugs where django-nose (optional) would try to run some things that aren't tests.
- ticket #110: Fix infinite loop if newsitem.schema.has_newsitem_detail is False but newsitem.url is empty; give 404 instead.
- Importers should now not blow up if run more than once.
- ticket #22: Scraper scripts in everyblock/cities/boston mostly don't work OOTB
- ticket #79: Geotagging oddity
- ticket #188: items.json doesn't include location_name
- ticket #200: "obdemo bin scripts are documented, but don't get installed when installing obdemo non-editable"

Documentation

- ticket #80: Documentation for Street Misspellings
- ticket #162: Document pip / easy_install workarounds
- ticket #139: Document adding database user / granting database access
- ticket #198: version number in documentation
- ticket #197: documentation for deploying static media
- Fix outdated paths to example scrapers.
- Fix location of get_or_create_lookup
- Note differences from everyblock

Other

- ticket #181: Prepare packages for distribution on pypi.
- ticket #83: Split out non-core packages into a separate download (`ebblog`, `ebwiki`, `ebgeo`, `ebinternal`, and `everyblock` are now at https://github.com/openplans/openblock-extras)
- ticket #156: Removing lots of clustering code that's totally unused.
- remove a redundant get_metro_bbox function from ebpub.utils.geodjango; use get_default_bounds(), does the same thing.

OpenBlock 1.0 beta (August 18, 2011)

This was largely identical to 1.0.0 described above, modulo a few small bug fixes.

OpenBlock 1.0 alpha 1 (June 9, 2011)

This was the first numbered release; too many changes since the "ebcode" source dump to enumerate.

Highlights

- Out-of-the-box theme, with maps.
- REST API
- Enable the Django admin UI
- Embeddable widgets that you can configure via the admin UI
- Lots more documentation

CHAPTER 6

Indices and tables

- search

Index