eadred Documentation

Release 0.3

Will Kahn-Greene

April 16, 2014

Contents

eadred is a Django-app for generating sample data.

Version 0.3

Code http://github.com/willkg/django-eadred

License BSD 3-clause; see LICENSE file

Issues https://github.com/willkg/django-eadred/issues

Documentation http://django-eadred.rtfd.org/

CHAPTER 1

Table of Contents

1.1 About eadred

eadred is a Django-app for generating sample data.

Why? Here are the use cases we're solving:

Use Case 1: Contributors

> Mildred wants to contribute to your Django project, but your Django project is non-trivial and out of the box it's not very usable because it needs data.
>
> However, you're using eadred, so in your setup documentation, you have a one-liner that generates all the sample data Mildred needs to start hacking immediately.

Use Case 2: Bootstrapping

> Willhelm wants to set up an instance of your Django project. It requires certain non-trivial initial data to be in place before it works.
>
> However, you're using eadred, so in your setup documentation, you have a one-liner that generates all the initial data needed.

Use Case 3: Large amounts of random data

> Phylida is a hacker on your Django project and trying to fix bugs with a section of code that requires large amounts of data—say it's one of those things that graphs data sets or something.
>
> You're using eadred, so it's a one-liner to generate a large set of initial data.

"Wait, use cases 1 and 2 are easily solved with Django and fixtures!"

I think fixtures are good for specific use cases where your models aren't changing and you have some contributor who likes entering in data to build the initial fixtures. Having said that, I don't use fixtures.

eadred allows you to programmatically generate the data using model makers, factories, fixtures, random seeds—whatever your needs are.

Additionally, eadred provides library functions to make generating data easier.

1.2 What's new in eadred

1.2.1 Version 0.3: April 16th, 2014

Changes

- Fixed documentation.
- Fixed test infrastructure and added tox support.
- Support for Python 3.3 and later. (Thank you Trey Hunner!)

1.2.2 Version 0.2: February 16th, 2013

API-breaking changes:

None

Changes

- Added some helper functions for generating data.

1.2.3 Version 0.1: September 28th, 2012

API-breaking changes:

None

Changes

- Initial writing.

1.3 Installation

There are a few ways to install eadred.

1.3.1 From PyPI

Do:

```
$ pip install eadred
```

1.3.2 From git

Do:

```
$ git clone git://github.com/willkg/django-eadred.git
```

For other ways to clone, see https://github.com/willkg/django-eadred.git.

1.3.3 Add to installed apps

After installing eadred, add it to the `INSTALLED_APPS` your project's `settings.py`:

```
INSTALLED_APPS += ('eadred',)
```

1.4 Generating sample data

1.4.1 Creating a generate_sampledata function

Before you can generate data, you need to write a `generate_sampledata` function for every Django app that you want to generate sampledata.

For example, say you had a Django project named *testproject* with a directory structure like this:

```
testproject/
|- __init__.py
|- settings.py
|
|- testapp/
   |- __init__.py
```

In the `settings.py` file you have `INSTALLED_APPS` set like this:

```
INSTALLED_APPS = ['testproject.testapp']
```

That way `testproject.testapp` is a valid Django app in your Django project.

In the `testapp/` directory, create a file named `sampledata.py`.

In the `sampledata.py` file, create a function named `generate_sampledata`.

Here's an example:

```
def generate_sampledata(options):
    pass
```

In that function, you do whatever you want to do to generate data.

1.4.2 Options

The *options* argument holds the options that the OptionParser that Django uses to parse the command line arguments. Thus it has the standard options that are passed to every Django command:

- *settings*
- *pythonpath*
- *verbosity*
- *traceback*

Most of those probably aren't useful, but *verbosity* might be.

Additionally, you can use the *with* keyword argument to pass additional parameters which will get pulled out and added to the *options* dict.

For example:

```
def generate_sampledata(options):
    print options.get('foo')
```

If you do this:

```
$ ./manage.py generatedata --with=foo=bar
```

it'll print:

```
bar
```

1.4.3 Running ./manage.py generatedata

General use

To generate sample data, do this:

```
$ ./manage.py generatedata
```

That will go through all your `INSTALLED_APPS` looking for modules named `sampledata` and executing the `generate_sampledata` method in each.

Generating data in specified apps

Say you had a bunch of apps in your Django project that have sampledata modules, but you only want to generated data in one of them. You can specify the apps you want to generate data in on the command line:

```
# All apps
$ ./manage.py generatedata

# Only app1
$ ./manage.py generatedata app1

# Only app1 and app2
$ ./manage.py generatedata app1 app2
```

Passing arguments

You can also pass arguments to the `generate_sampledata` functions using the *with* keyword argument.

Examples:

```
# Passes {'fixtures': True} to options
$ ./manage.py generatedata --with=fixtures

# Passes {'type': 'random'} to options
$ ./manage.py generatedata --with=type=random

# Passes {'type': 'random', 'seed': '1024'} to options.
$ ./manage.py generatedata --with=type=random --with=seed=1024
```

You can have as many as you like—each will get parsed out as a separate key or key/val parameter.

1.4.4 Recipes

Example 1: Load a fixture

This `generate_sampledata` function loads a fixture (ab)using the Django loaddata command:

```
from django.core.management.commands import loaddata

def generate_sampledata(options):
    cmd = loaddata.Command()
    cmd.execute('mydata.json')
```

Run it like this:

```
$ ./manage.py generatedata
```

Example 2: Generate data with model makers

This example has a rough model maker for the Record model. Also, it allows the user to specify how many records he/she wants to create using the *count* parameter:

```
import datetime
from someproject.someapp.models import Record

def record(**kwargs):
    rec = Record(**kwargs)
    rec.save()
    return rec

def generate_sampledata(options):
    count = options.get('count', '10')
    count = int(count)

    now = datetime.datetime.now()

    # Creates count number of records, each on a new day.
    for i in range(count):
        record(created=now - datetime.timedelta(days=i),
               message='Lorem ipsum %d' % i)
```

Run it like this:

```
$ ./manage.py generatedata
$ ./manage.py generatedata --with=count=20
```

1.5 API

eadred comes with some helper functions for generating random data. If they work for you, yay! If not, you can look at the code and write your own.

eadred.helpers.**make_unique**(*gen*)

Wraps a generator to uniquify strings by appending counter

Parameters **gen** – the generator to wrap

Example:

```
from eadred.helpers import name_generator, make_unique

gen = make_unique(name_generator())
for i in range(50):
    mymodel = SomeModel(name=gen.next())
    mymodel.save()
```

Example 2:

```
>>> gen = make_unique(name_generator(['alice', 'jane', 'harry']))
>>> gen.next()
u'alice0'
>>> gen.next()
u'harry1'
>>> gen.next()
u'jane2'
```

eadred.helpers.**name_generator**(*names=None*)

Creates a generator for generating names.

Parameters **names** – list or tuple of names you want to use; defaults to ENGLISH_MONARCHS

Returns generator for names

Example:

```
from eadred.helpers import name_generator

gen = name_generator()
for i in range(50):
    mymodel = SomeModel(name=gen.next())
    mymodel.save()
```

Example 2:

```
>>> gen = name_generator()
>>> gen.next()
u'James II'
>>> gen.next()
u'Stephen of Blois'
>>> gen.next()
u'James I'
```

Note: This gives full names for a "name" field. It's probably not useful for broken down name fields like "firstname", "lastname", etc.

---

eadred.helpers.**email_generator**(*names=None*, *domains=None*, *unique=False*)

Creates a generator for generating email addresses.

Parameters

- **names** – list of names to use; defaults to ENGLISH_MONARCHS lowercased, ascii-fied, and stripped of whitespace
- **domains** – list of domains to use; defaults to DOMAINS
- **unique** – True if you want the username part of the email addresses to be unique

Returns generator

Example:

```
from eadred.helpers import email_generator

gen = email_generator()
for i in range(50):
    mymodel = SomeModel(email=gen.next())
    mymodel.save()
```

Example 2:

```
>>> gen = email_generator()
>>> gen.next()
'eadwig@example.net'
>>> gen.next()
'henrybeauclerc@mail1.example.org'
>>> gen.next()
'williamrufus@example.com'
```

eadred.helpers.**sentence_generator**(*sentences=None*)

Creates a generator for generating sentences.

Parameters **sentences** – list or tuple of sentences you want to use; defaults to LOREM

Returns generator

Example:

```
from eadred.helpers import sentence_generator

gen = sentence_generator()
for i in range(50):
    mymodel = SomeModel(summary=gen.next())
    mymodel.save()
```

eadred.helpers.**paragraph_generator**(*sentences=None*)

Creates a generator for generating paragraphs.

Parameters **sentences** – list or tuple of sentences you want to use; defaults to LOREM

Returns generator

Example:

```
from eadred.helpers import paragraph_generator

gen = paragraph_generator()
for i in range(50):
    mymodel = SomeModel(description=gen.next())
    mymodel.save()
```

1.6 Hacking HOWTO

This covers setting up a development environment for developing on eadred.

1.6.1 Setting up a development environment

Run:

```
$ virtualenv ./venv/
$ . ./venv/bin/activate
$ pip install -r requirements/dev.txt
```

This installs all required dependencies for eadred and eadred itself for development.

Note: You don't have to put your virtual environment in `./venv/`. Feel free to put it anywhere.

1.6.2 Coding conventions

- PEP-8: http://www.python.org/dev/peps/pep-0008/
- PEP-257: http://www.python.org/dev/peps/pep-0257/
- Use pyflakes. Srsly.

pep8 covers Python code conventions. pep257 covers Python docstring conventions.

Minor caveats:

- We use Sphinx, so do function definitions like they do: http://packages.python.org/an_example_pypi_project/sphinx.html#function definitions.
- Don't kill yourself over 80-character lines, but it is important.
- If you're flummoxed by the conventions, just send me the patch and as long as it functionally works, I can do a cleanup pass in a later commit.

1.6.3 Git conventions

I encourage good commit messages in a form that works well with git's various commands. Something like http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html. except that I don't care about verb tense or capitalization and if the commit message is tied to a bug report, the bug report number should be the first thing in the first line. Here's the tbaggery example with some adjustments:

```
475. short summary (50 chars or less)

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as
the subject of an email and the rest of the text as the body.  The
blank line separating the summary from the body is critical
(unless you omit the body entirely); tools like rebase can get
confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded
  by a single space, with blank lines in between, but conventions
  vary here

- Use a hanging indent
```

Why? Here's the reasons:

- 50 characters or less works well with the various git commands that show only the summary line and also on github.
- Having the bug number as the first thing makes it easy to see which commits covered which bugs without parsing the commit message. We do that a lot ("When did the fix for bug xyz land?").
- Wrapping the subsequent paragraphs allows them to show up nicely in git output as well as on github.

Why not the other things? Here's the reasons:

- Capitalization or non-capitalization for a phrase doesn't affect the output of git commands or my ability to quickly parse a summary.
- Ditto for verb tense.
- I'm all for ditching convention baggage for things that don't matter.

1.6.4 Code documentation conventions

Documentation in the code is really helpful. Please add comments where you think it's necessary.

We like to use docstrings for classes, methods and functions. They should be in reStructuredText format. Something along these lines, though most of our docstrings aren't as formal or complete:

```
def foo(arg1, arg2):
    """Foo does something interesting

    :arg arg1: Controls whether or not to bar
    :arg arg2: Name of the baz to use

    :raises ValueError: If arg2 is not a valid baz.

    :returns: A bat.
    """
```

The purpose in-code documentation is three-fold:

1. to clarify complex code so it's easier to discern what it's doing
2. to make it clear why the code is doing what it's doing

3. to document any issues the code might have

1.6.5 Building the documentation

The documentation in *docs/* is built with Sphinx. To build HTML version of the documentation, do:

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

1.6.6 Running the tests

To run the tests, do:

```
$ ./run_tests.py
```

or run it with the python interpreter of your choice:

```
$ /path/to/python run_tests.py
```

Also, you can run the tests in the various environments we support:

```
$ tox
```

1.6.7 Writing tests

Tests are located in `eadred/tests/`.

We use nose for test utilities and running tests.

1.6.8 Release process

1. Checkout master tip.
2. Update version numbers in `eadred/_version.py`.
   (a) Set `__version__` to something like `0.4`.
   (b) Set `__releasedate__` to something like `20120731`.
3. Update `CONTRIBUTORS`, `CHANGELOG`, `MANIFEST.in`.
4. Verify correctness.
   (a) Run tests.
   (b) Build docs.
   (c) Verify all that works.
5. Tag the release:

   ```
   $ git tag -a v0.4
   ```

6. Push everything:

   ```
   $ git push --tags origin master
   ```

7. Update PyPI:

```
$ python setup.py sdist upload
```

8. Blog post, twitter, etc.

CHAPTER 2

Indices and tables

- *genindex*

e