djangocon-2011-notes Documentation

Release 0.0.1

Kenneth Love

September 29, 2011

CONTENTS

Notes, duh!

CHAPTER

ONE

DJANGOCON 2011

1.1 From Designer to Django'er in Six Weeks

1.1.1 Presented by Tracy Osborn

1. **Start out on the right foot:**
    - If you can, spend 100% of your time on your project.
    - Have a large savings account, low debt.
    - Make sure you're having fun.
    - Success can be just launching the app.
    - Be focused and optimistic before you start.

   Cofounders are awesome, but no cofounder is better than the wrong cofounder.
2. **Launch as fast as possible:**
    - What can you take out?
    - Launch tiny bits to keep motivated.
    - Work on the hard stuff first.
    - Spent as little time as possible on learning, as much on building.
    - Launch with bad code (really, it's okay).
3. Have a plan for monetization
4. **Don't be forever alone:**
    - Talk to friends.
    - Don't be prideful; accept help.
    - NDAs suck.
    - Surround yourself with good people & resources.

5: Take shortcuts:

- Django is plug & play
- South, django-registration, django-profiles, sorl-thumbnail, django-debug-toolbar, djangopackages
- dotCloud

- virtualenv, pip

5.5 Design shortcuts:

- themeforest
- 99designs (quick logos only)
- "The Non-designer's Design Book"
- Create a coming-soon page.

Q&A

1. **Django/Python rough edges?**
   - Review logic first (when learning).
   - Had lots of people to ask questions to. Removed most (all?) speed bumps.
2. **Why choose Django and stay with it?**
   - Chose Python, really, not just Django.
   - Django provides for lazy developing :)
3. **How did you find customers?**
   - Cold emailing for the vendors.
   - Blogs and promotion help a lot.
4. **How to rein in a designer that likes to design hard-to-build items?**
   - Make him build it!
   - Pluggables never exactly match.
5. **How far did your roadmap go?**
   - Not far at all.

1.2 Confessions of Joe Developer

1.2.1 Presented by Daniel Greenfeld

1. **I'm stupid:**
   - **Can't figure things out**
     - **If I get stuck for more than 30 minutes:**
       * Find libraries that do it for me.
       * Ask on Twitter for answers.
       * SO is good, but watch out for trolls.
       * IRC can be good, again, trolls.
   - **Can't remember things**
     - Documentation makes me look good.
     - Docstrings are awesome.

        - Learn you some REStructuredText
        - Write down even the slide bullets.
        - Sphinx makes me and my code look good.
    - Ask the dumbest questions
2. **I'm lazy:**
    - **Don't want to do anything twice**
        - If I write the same code twice, I stick it in a method.
        - Then stick the function into a utils module.
        - Then I put it up on Github so I don't lose it.
    - **Don't want to debug code that worked before**
        - Manually testing code by watching it run is hard...
        - ... and boring ...
        - ... and hence is error prone.
        - Meaning you have to do more work.
        - **Are you testing enough?**
            * coverage.py is great
            * django-coverage runs coverage.py for Django
            * But you only want to test your own apps. Refer to code on Github for how.
            * Look at how he does INSTALLED_APPS
    - Don't want to upload ZIP files per documentation change
3. Don't be smart and lazy
4. **Technical Debt**
    - **Postponed activities:**
        - **Documentation**
            * unshared knowledge
        - Tests
        - attending to TODO statements
        - Code too confusing to be modified easily

A positive trait good tech leads often look for is the ability to ask questions.

Links

- http://bit.ly/audreyr-sphinx
- https://github.com/pydanny/django-party-pack/

Q&A

1. **Do you try to understand the 3rd party code or isolate it?**
    - Pip & Virtualenv to isolate it.

1.3 Testing: The Developer Strikes Back

1.3.1 Presented by Sandy

1. **Testing is hard to do right.**
    - Delegate test responsibilities correctly.
    - Won't (always) get it right on the first try.
2. Unit test organization strategies for Django projects

    What is "Unit Testing"? A method by which individual units of source code are tested.
    - **Each app needs it's own test module.**
        - **Each module should have (many) submodules.**
            * Each contains classes that test units of code.
    - **Organize your test suite however you want**
        - Just be consistent.
3. **Dealing with increasingly complex test scenarios**
    - Separate code and service/infrastructure testing
    - Don't keep all the tests in one `tests.py`
    - Use sub-module and root `tests.py` files sparingly.
    - Give Mock (http://python-mok.sourceforge.net/) a try.
4. **Beyond the business logic**
    - Testing 3rd party APIs or cache
    - **These should be tested separately**
        - Better speed of tests if you're not constantly making outbound calls.
    - **Dealing with cache:**
        - Dev env doesn't usually have long-term or complete sets of cached data.
        - Cache should be agnostic to model changes.
        - Do object structure comparison when pulling from cache.
        - Testing how your app deals with 'old' data, not the infrastructure.
5. **Writing tests can improve coding habits**
    - Smaller, modular code is testable, 200-line functions aren't.
    - **Write more tests**
        - Discover easily-testable coding patterns.

        - Functions should perform a **single function**.

6. Tests are alive
7. **How do I enforce testing on my team?**
    - Most people don't like 2AM fires.
    - Iterate faster with confidence.
    - Git pre-commit hooks, stop checkins without tests.
    - Coverage.py can make it a game.
    - Public shaming?
8. **How much testing is enough testing?**
    - No simple answer for that.
9. **Junction between Unit and Integration**
    - Difficult areas to test, because behavior is dependant upon environment.
10. Testing a virgin codebase
    - You will refactor code as you write tests
11. Successful strategies
    - Require unit tests for all code going forward
    - Reserve time to clear out test debt
        - Good for new team members
        - Everyone takes a turn
    - Establish a good foundation to build on
    - Every bug is two bugs:
        - One in the code that broke
        - One is the test that would/will catch the bug
12. Test data
    - Fixtures, mocks, ObjectCreator
    - Use SQLite
13. Should I get test data from cache?
    - No. Adds unnecessary complexity to unit tests.
    - If tests only succeed with certain caches, probably some code that can be refactored.
14. Graceful code degredation
    - Devs need to think outside their dev instance
    - Service unavailable shouldn't mean your site is unavailable
    - If failure is catastrophic, you should know **how** and **why** it happened.
15. Forcing awareness
    - Code should not be written in such a way that it won't work (at least in some fashion) if certain services or infrastructers are missing/different.

16. Test infrastructe
    - Staging should be identical to production.
    - Run Solr or RabbitMQ on staging, but not dev.
    - Don't overdo it on logging.
17. Testing tools
    - Nose
    - Coverage
    - Mock

    Code not tested is broken by design

1.4 Itch Scratching

1.4.1 Presented by Brad Fitzpatrick

I wanna be a mother-f****** computer programmer and you betta aks somebody

Software never fuckin' dies.

Realization: not all your ideas are dumb.

Easy != Quick Time-consuming != Hard

http://contributing.appspot.com

1.5 Django Package Thunderdome: Is your package worthy?

1.5.1 Presented by Audrey Roy & Daniel Greenfeld

1. Review criteria
   (a) Purpose
       - Name should tell you what it does.
       - Real need and be useful.
   (b) Scope
       - Small, narrow focus is better.
   (c) Documentation
       - No docs means the package is pre-alpha
       - Doc strings != documentation
       - If there are dependencies, they should be in the docs
       - Installation should be bulletproof
   (d) Testing
       - Tests improve reliability

- Tests make it easy to advance python/django versions
- Tests make it easier for community submissions

(e) Activity

- When was the last commit?
- How frequent are the commits?
- Are there periodic version updates?

(f) Community

- Active leaders
- How many contributors?
- Proper attribution of authors?

(g) Modularity

- "Pluggability"
- Installation should be minimally invasive
- Do not confuse modularity with over-engineering

(h) Availability on PyPI

- Actually on PyPI
- **Latest release on PyPI**
  - Should not have to go to repo for working version
- Proper version numbers

(i) VCS/Hosting

- Great: Github/Bitbucket
- OK: Launchpad/SourceForge
- Outdated: Google project hosting
- Poor: Bespoke

(j) License

- You need a license
- Companies prefer BSD or MIT
- http://opensource.org/licenses/category

2. **API creation**

(a) **django-piston**

- Lost points for docs in a wiki.
- Hundreds of forks

(b) django-tastypie (2nd place)

(c) django-rest-framework (1st place)

(d) django-xmlrpc * No tests

3. **Fundamentals**

(a) django-debug-toolbar (1st place)

(b) django-coverage

(c) **django-extensions (2nd place)**

- Scope is just too big

(d) **Pinax**

- Scope is just too big
- History is unclear on PyPI

4. **Registration**

(a) **django-registration**

- Popular, un-official mirror with templates.
- If the original had templates, the fork wouldn't be needed.

(b) Pinax

(c) **django-userena (2nd place)**

- Inaccurate authors file

(d) django-social-auth (1st place)

5. **Profiles**

(a) django-profiles

(b) **django-easy-profiles**

- Not really on PyPI
- Not really modular or tested

(c) django-userena (2nd place)

(d) idios (1st place)

6. **Blogs**

(a) biblion

(b) django-mingus

(c) django-basic-apps

(d) django-blog-zinnia (1st place)

6. **Tagging**

(a) django-taggit (1st place)

(b) django-tagging

(c) django-tagging-ng

7. **Database Migrations**

(a) **South (1st place)**

- no license?

(b) **nashvegas (2nd place (no shit?))**

- No tests

8. **Honorable Mentions**
    (a) celery & django-celery
    (b) django-haystack with pysolr or whoosh
    (c) django-fixture-generator, django-sorting, django-pagination, others but they went too fast.

Note: http://bit.ly/django-thunderdome-2011

---

9. **Beyond**
    - run code through PEP8
    - aim for 100% test coverage
    - elegant, clean, explicit ways of doing things
10. Get more users * djangopackages.com (!)

1.6 Real World Django Deployment with Chef

1.6.1 Presented by Noah Kantrowitz

Getting started with chef

- Resources
  - type
  - name
  - parameters
  - takes actions
  - can send notifications
- Recipes
  - A collection of resources
  - Recipes are evaluated for resources in the order they appear.
  - Can include other recipes
  - Dynamic configuration through search
  - Can be extended through Ruby scripting
- **Roles**
  - describe nodes
  - holds a list of recipes
- Cookbooks are collections of recipes
- Environments are pegged cookbook versions

Python-specific tools

- python::package, python::source – install python
- python::pip, python::virtualenv – make it dance
- gunicorn::default, gunicorn_config
- supervisor::default, supervisor_service

Note: https://github.com/coderanger/djangocon2011

Note: http://pydanny-event-notes.readthedocs.org/en/latest/DjangoCon2011/deploy-via-chef.html

---

1.7 Building Web APIs in Django with Tastypie

1.7.1 Presented by Issac Kelly

Slides: http://bit.ly/nwfYvI

Demo: http://bit.ly/oKcCWM

Code: http://bit.ly/mTs5Yw

Why Tastypie?

- Makes sense
- Well-tested
- HATEOAS
- Good features and support, fits well into existing ORM thinking
- Totally extensible
- Works well with Javascript
- Cool stuff is coming out of the community

Why not Tastypie?

- You know better than your users (functional API instead of REST API)
- RESTish is enough

Serialization Methods

- JSON
- XML
- YAML
- Binary plist

Authentication Classes

- NO-OP
- API key
- HTTP Basic Auth & Digest

Authorization Classes

- ReadOnly
- Default Authorized
- Django Authorization

It's Python, class-based, extensible.

1.8 Stop Tilting at Windmills: Spotting Bottlenecks

1.8.1 Presented by Yann Malet & Brandon Konkle

Get numbers before you start trying to debug. You can't fix a problem you don't have.

Performance Testing

- Time to load a page
- Number of queries per page
- Time taken by the queries
- Number of cache hits/misses
- Time spent waiting on 3rd parties

Note: Django Debug Logging: https://github.com/lincolnloop/django-debug-logging

---

Load Testing

- Taking your entire app infrastructure into account
- Understanding how your performance changes under stress and over time
- Creating realisting test plans that imitate real life traffic

Database

- Handling numerous queries in parallel
- Types of queries running at the same time

Caching

- Cache key expiration
- Cache size and key eviction

Note: JMeter

Command-line tools

- htop
- dstat
- pg_top, mtop
- memcache-top

1.9 Best Practices For Frontend Django Developers

1.9.1 Presented by Christine Cheung

Polished front-end is as important as back-end.

It may scale, but bloated markup and JS will slow performance.

The implementation of the design is what the user notices.

Start Organized

Structure the hierarchy of static and template files.

Templatetags and Filters

Template system is meant to express presentation, not logic.

CSS & Javascript

Avoid plugin overkill. No more than 3-4.

Namespace your Javascript.

- Prevents conflicts
- Easier to read & maintain

Don't use $(document).ready()

Do Not:

- `document.write`
- Inline Javascript
- Backbone with Tastypie

Tools For Rapid Development

- HTML5 Boilerplate

Note: I disagree, but not important.

- Modernizr
- **Compass Framework**
  - CSS authoring framework & set of utilities
  - SASS
  - Image spriting
  - `django-compass`

Data Handling

- Leverage the power of both back and frontends
- Share the work between them
- Class-based views for quick prototyping

Define your datatypes

- Write out the API, evaluate its design.
- Know when to make a view vs API
- Contex Processors (API keys, etc)

Testing and Performance

- CSSLint
- JSLint
- Google Closure Compiler (turns simple functions into closures)
- Use a build script to minify and gzip files.

Note: I'd recommend combining files too.

- Have a `TEMPLATE_DEBUG`-triggered switcher for flat/compiled static files
- Async/lazy-load JS

Note: Or media: images, video, etc

1.10 Designers Make It Go To Eleven

1.10.1 Presented by Idan Gazit

Compromise is the soul of design.

There's no way to get everything that you want.

> Design is a discipline you can learn. And you need to practice to level up.

You drive a TV with a remote. You drive an iPhone with your whole body. Medium provides a constraint to your design.

The most important constaints are audience constraints:

```
* Who
* What
* When
* Where
* Why
```

Constraints inform choices. Based on science.

User Experience Delight & obviousness

Interaction Design input/output often confused with UX paving cowpaths

User Interface expose functionality cues & clues affordance discoverability conventions interactability

Information Architect info hierarchy sort criteria findability

Visual Design visual hierarchy establishing connections mood & narrative beauty = performance

> Things that are beautiful (seem to) work better

> I need your help to make [my project] pretty. Designers will run away, screaming

All the same reasons for working in OO, build up a resume, etc.

As OO goes, Django is pretty designer-friendly. It appreciates non-code contributions: docs & tests.

Note: http://bit.ly/suck-threshold

Project templates would be awesome for new users. Pinax and the like are too heavy, though.

1.11 Advanced Django Form Usage

```
if any(self.errors):
```

1.12 Teaching Django to Comrades

1.12.1 Who is your audience?

- New to web programming
- New to Python

1.12.2 Where do you start?

1. The tutorial.
2. Something else that they can build, from green pastures, with code reviews.
3. THEN get them working on your code.

1.12.3 What are some of the pain points?

- Reverse relations
- Migrations
- **Forms**
  - What do I use?
  - Custom widgets?
  - Custom data?
- Where do random bits of code go?
- Existing methods or write my own?
- URLs

Note: If it's hard, and you're new, you're probably doing it wrong.

Note: Don't fight the system until you're assured victory.

Deployment: don't start here.

Consultants, tutorials, and convention videos are great starting resources.

1.12.4 Get Help

- IRC
- django-users mailing list
- StackOverflow
- User groups

1.12.5 Give Help

- All of those places.
- Fork projects.
- Report, verify, and fix bugs in 3rd party apps.

1.13 Y'all Wanna Scrape with Us? Content Ain't a Thing : Web Scraping With Our Favorite Python Libraries

1.13.1 Presented by Katharine Jarmul

I got 99 problems but content ain't one

- Everyone needs good content.
- Good content exists all over the web.
- Scrape it 'til you make it.

LXML: Diving in

`lxml.etree` vs. `lxml.html`

- `etree`: best for properly formatted xml/xhtml
- `etree`: powerful and fast for SOAP or other xml-formatted content
- `html`: best for web sites & irregular content

`lxml.html`: hidden gems

`cssselect` utilizes css element syntax to find and highlight html elements.

`iterlinks` creates a generator of all **linky** elements on the page. Remember: ads have lots of links.

`sourceline` can identify the location of your element on the page. Exists in both `lxml.html` and `lxml.etree`.

`find`, `findall` can locate html elements within another node or a page. Exists in both `lxml.html` and `lxml.etree`.

descendents/children/siblings/ancesorts all elements have `iterchildren`, `itersiblings`, `iterancestors` and `iterdescendents`.

forms can find all (normal) forms on a page. beware of CAPTCHAs and the like.

`text`, `text_content`, and `iter_text` ways to get content without tags.

If you have to parse in realtime, LXML is sometimes too much.

`re` html == strings == parseable.

`feedparser` standard XML has rules, feedparser knows them.

htmlparser good base class for your own HTML parser. good for "I have an idea about how I want to handle `embed` tags".

Content is 1/2 of the equation.

I'm tired of ugly pages with badass content.

Note: Text = Content = Boss

1.14 Making Interactive Maps for the Web

1.14.1 Presented by Zain Memon

DB PostgreSQL + PostGIS

ORM GeoDjango

Front-end TileStache

Slippy Map PolyMaps (but would probably use Leaflet or GMaps in the future)

What makes a good map

- **Attractive**
  - Use a different base layer
  - cloudlayer.com
  - GMaps styled map wizard
- Readable
- Interactive
- **Fast**
  - Use a CDN
  - Don't pass through too much data
  - Don't use too many polygons (draw shapes on the server and send flat images to the map, if possible)

They have a bunch of points, for each point they define arbitrary clusters.

Note: http://polymaps.org/ex/cluster.html

1.15 Lightning Talks

1.15.1 1. Redis and Protocol Buffers

Presented by Joshua Ginsberg

Protocol buffers from Google.

- Non-human-readable :(

- 1.25M records takes up about 250MB of memory.
- Protocol versioning is built-in

1.15.2 2. Managing Servers

Presented by Nate Aune

How to make deployment a non-event?

Note: http://bit.ly/php-to-django

1.15.3 3. The CMS that doesn't do anything

Philo

Template designers can customize how model instances are embedded per template

Template designers are in charge of how content is interpreted

http://philocms.org

1.15.4 4. Making small, positive changes

Presented by Steve Holden

- We can make a positive difference in many ways.
- Doing so is good for us and good for our community.
- Tell people what to do.
  - They need some encouragement.

Note: Alex Gaynor is the Django community's standard equivalent of a kitten.

1.15.5 5. Backbone.js with Django and Tastypie

Two things needed to make it work:

- change backbone sync method
- override url and parse from the sources

1.16 Class-Based Views

1.16.1 The Old Way

You probably use one of these two ways for making a view that renders a form and then validates it on submit:

```
@login_required
def awesome_view(request):
    if request.method == "POST":
        form = AwesomeForm(request.POST)

        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('awesome'))
        else:
            form = AwesomeForm()

    return render("awesome.html", {'form': form})
```

Or:

```
@login_required
def awesome_view(request):
    form = AwesomeForm(request.POST or None)

    if form.is_valid():
        form.save()
        return HttpResponseRedirect(reverse('awesome')

    return render("awesome.html", {'form': form})
```

These both work and they're fine but you often have to repeat yourself and you have to deal with `if` conditions when debugging. Not much fun, not very clean.

1.16.2 The New Way

So, here's a class-based way of doing it:

```
Class MyAwesomeView(TemplateView):
    template_name = "awesome.html"

    def get(self, request):
        form = AwesomeForm()
        return self.render_to_response({'form': form})

    def post(self, request):
        form = AwesomeForm(request.POST)
        if form.is_valid():
            form.save()
            return HttpResponseRedirect(reverse('awesome'))

        return self.render_to_response({'form': form})
```

You have a `template_name` variable at the top that specifies the template for the view.

Below that is the `get()` method that handles all `GET` requests. Typically it just takes `self` and `request` but if your route requires more arguments, they'll need to be specified here, too.

And, surprise, surprise, below that is `post()` that takes care of your `POST` requests. Same arguments as `get()` since it has to be hit by the same route.

1.16.3 What about @login_required?

True, we don't have `@login_required` on our methods and it won't work if you just add it there. What you can do, though, is wrap the `dispatch()` method and that will wrap all methods from that view with whatever decorators you provide. Example below:

```
@method_decorator(login_required)
def dispatch(self, *args, **kwargs):
    return super(MyAwesomeView, self).dispatch(*args, **kwargs)
```

Add this into your class and you should be safe from all those nasty unauthenticated users.

1.16.4 URLs

How do you add these views to your URLs? Glad you asked:

```
from project.views import MyAwesomeView

[...]
url(r'^awesome/$', MyAwesomeView.as_view(), name="awesome")
[...]
```

If you really hate having to import each view, at the bottom of your `views.py` file, you can do something like: `awesome_view = MyAwesomeView.as_view()` and then set your URLs view as `app.awesome_view`. I don't think this is needed, but to each his/her own.

Thanks and I hope that helps everyone get a small grip on class-based views. There are many other class-based views available, such as `FormView` for handling form rendering and validation (so this entire method wouldn't be needed) but there are very little docs for them. **Let's write them!**

CHAPTER

TWO

PDX PYTHON

2.1 PDX Python Meetup

2.1.1 September, 2011

Delicious data with mmstats

Problem:

- You have an app.
- It has state.
- What is my app doing?
- How do you inspect for that state?
- Simple in-memory stats get hard to expose in multi-process environments.

Solution #1: Logging

Pros

- Universally supported
- Easily enhanced
- Persistent

Cons

- Libraries suck
- Operational burden (rotating, shipping, routing, etc)
- Records events more than inspects state
- Difficult to predict where needed
- Performance impact if too verbose

Solution #2: Graphite

Pros

- Fast, sexy, "enterprise"
- Python

Cons

- Do you want a graph or a graph?
- Have fun installing it.
- Still not great for introspection.

Solution #3: socketconsole

Pros:

- Pure Python.
- Very useful for deadlocks, blocking code, threaded app.
- Simple to integrate:

```
import socketconsole
socketconsole.launch()
```

Cons:

- CPython only.
- Doesn't work with gevent or eventlet monkeypatching.
- Doesn't work with greenthreads.
- Limited functionality.
- All the fun of Python threads.

Solution #4: REPL Backdoors

Pros:

- Pure Python!
- Changing code at runtime is for winners.
- Inspect all the things!

Cons:

- With great power comes great responsibility.
- Requires threads or event loop.
- Still can't reach all state.

Solution #5: GDB

Pros:

- Well, you wanted introspection.
- Has some Python helpers: `pygdb`, `gdb-heap`

Cons:

- Seriously? Definitily only a a last resort.

Solution #6: JMX

Pros:

- Universally supported.
- Powerful, extensible, 2-way.
- Helpful tools.

Cons:

- Where "universal" means "runs on the JVM".
- Not as easy to monitor as you'd think.

Note: Python needs this.

`mmstats` Goals

- Simple API to expose state.
- Separate publishing from reading, aggregating, etc.
- Language, platform, framework agnostic.
- Minimal & predictable performance impact.
- Optional persistence (e.g. post-mortems)
- 1-way (for now?)

What is `mmstats`?

- `mmap` alows sharing memory between processes.
- Language independent data structure:
  - Series of fields (structs)
  - Fields have `label`, `type`, and `values`.
- Exposed in Python app as a model class.

Performance Implementation

Single writer, multi-reader

- No locks.
- No syscalls (write, read, send, recv, etc).
- All in userspace for readers & writers.
- Reading has no impact on writers.
- Fixed field sizes.

Consistency without Locks

Q: How are reads consistent without locks? A: Double buffering.

`django-slow-log` or How we found a view doing 100k queries

Things it does:

- Query count.
- Hostname of the machine.
- Time delta of request started til response started.
- Memory delta.
- Load delta.
- Etc.

How it does it:

- Database backend.
- Uses celery.
- A couple system calls for the deltas.

Who should use it:

- Any Django site that does anything interesting (sites that do a decent amount of traffic, several hits per minute)

CHAPTER

THREE

OTHER

3.1 `django-social-auth` Setup

3.1.1 1. Install the app

`pip install django-social-auth` and wait a few minutes. This is a good time to get your API keys set up with Twitter, Facebook, whatever.

3.1.2 2. `settings.py`, where the work is really done

1. Once it's installed, add it to your `INSTALLED_APPS`.
2. Add `'social_auth.context_processors.social_auth_by_type_backends'` to your `TEMPLATE_CONTEXT_PROCESSORS`. Yes, this means you probably have to go look them up in Django's documentation so that you don't override the defaults (why isn't this in there by default?) but that's OK.
3. Then comes a big block of settings. I usually put these at the end:

```
# SOCIAL AUTH SETTINGS
TWITTER_CONSUMER_KEY = 'your-key-here'
TWITTER_CONSUMER_SECRET = 'your-secret-here'
FACEBOOK_APP_ID = 'your-id-here'
FACEBOOK_API_SECRET = 'your-secret-here'
SOCIAL_AUTH_ENABLED_BACKENDS = ('facebook', 'twitter')
SOCIAL_AUTH_COMPLETE_URL_NAME = 'socialauth_complete'
SOCIAL_AUTH_ASSOCIATE_URL_NAME = 'associate_complete'
SOCIAL_AUTH_DEFAULT_USERNAME = lambda u: slugify(u) # you'll need to import slugify from 'django
SOCIAL_AUTH_EXTRA_DATA = False
SOCIAL_AUTH_CHANGE_SIGNAL_ONLY = True

AUTHENTICATION_BACKENDS = (
    'social_auth.backends.twitter.TwitterBackend',
    'social_auth.backends.facebook.FacebookBackend',
    'django.contrib.auth.backends.ModelBackend',
)

LOGIN_URL = '/login/'
LOGIN_REDIRECT_URL = '/'
LOGIN_ERROR_URL = '/login-error/'
```

4. Next, edit `urls.py` and add in `url(r'', include('social_auth.urls'))`.

5. You'll want to set up some HTML links for people to login through:

```
<li><a href="{% url socialauth_begin 'twitter' %}">Login with Twitter</a></li>
<li><a href="{% url socialauth_begin 'facebook' %}">Login with Facebook</a></li>
```

3.1.3 3. Explanations of a few bits

In the 3rd step above, you can configure a few things differently if you want.

1. `SOCIAL_AUTH_ENABLED_BACKENDS` needs to match the backends that you add to `AUTHENTICATION_BACKENDS`. This lets you only load the backends you need.
2. Sadly, `SOCIAL_AUTH_COMPLETE_URL_NAME` and `SOCIAL_AUTH_ASSOCIATE_URL_NAME` seem to be magic settings with the last release. Just set them to this and don't worry about it unless you want to.
3. `SOCIAL_AUTH_DEFAULT_USERNAME` lets you set up a default username. You'll get one from the social auth service but it may not be compatible with Django's username requirements, hence the use of `slugify()`. You could also use `string.alphabet` and some randomness to make a string for them.
4. Lastly, `SOCIAL_AUTH_CHANGE_SIGNAL_ONLY` stops the 3rd party apps from overriding anything the user has now entered in their `User` instance. For example, Twitter returns a user's full name as Django's `first_name`. This means if you let them set first & last name, Twitter will override it each time they login. The `SOCIAL_AUTH_EXTRA_DATA` setting tells the backends we don't want any extra data they send through. If, though, you **do** want the data, obviously turn this off.

3.1.4 4. Extra stuff

Not something I use often, but you can set up signals for after a user logs in/signs up:

```
from social_auth.signals import pre_update
from social_auth.backends.twitter import TwitterBackend

def twitter_user_update(sender, user, response, details, **kwargs):
    profile, create = Profile.objects.get_or_create(user=user)
    profile.image = response['profile_image_url']

    tokens = response['access_token'].split('&')
    profile.oauth_token_secret = tokens[0].split('=')[-1]
    profile.oauth_token = tokens[1].split('=')[-1]

    if user.username != response['screen_name']:
        redirect = Redirect.objects.get(new_path=profile.get_absolute_url())
    else:
        redirect = False

    profile.user.username = response['screen_name']
    if create:
        profile.point = generate_point(lat=0, lng=0)
    profile.save()
    user.username = response['screen_name']
    user.save()

    if redirect:
        redirect.new_path = profile.get_absolute_url()
```

```
            redirect.save()

        return True

    pre_update.connect(twitter_user_update, sender=TwitterBackend)
```

This should be pretty self-explanatory if you're used to using signals in Django. This one, each time a user logs in, overrides their username and saves their tokens so we can post to Twitter as them.

If you have any questions, hit me up on Twitter as @kennethlove