Writing reusable applications

James Bennett

DjangoCon, Mountain View, CA,
September 6, 2008

The extended remix!

The fourfold path

- Do one thing, and do it well.
- Don’t be afraid of multiple apps.
- Write for flexibility.
- Build to distribute.

1

“

Do one thing, and do it well.

”

-- The UNIX philosophy

Application == encapsulation

Keep a tight focus

- Ask yourself: “What does this application do?”
- Answer should be one or two **short** sentences

Good focus

- “Handle storage of users and authentication of their identities.”
- “Allow content to be tagged, del.icio.us style, with querying by tags.”
- “Handle entries in a weblog.”

Bad focus

- "Handle entries in a weblog, and users who post them, and their authentication, and tagging and categorization, and some flat pages for static content, and…"
- The coding equivalent of a run-on sentence

Warning signs

- A lot of very good Django applications are very small: just a few files
- If your app is getting big enough to need lots of things split up into lots of modules, it may be time to step back and re-evaluate

Warning signs

- Even a lot of "simple" Django sites commonly have a dozen or more applications in `INSTALLED_APPS`
- If you've got a complex/feature-packed site and a short application list, it may be time to think hard about how tightly-focused those apps are

Case study: user registration

Django snippets: Sign up

http://www.djangosnippets.org/accounts/register/

djangosnippets

Log in or register | Snippets | About

By author | By language | By tag | Highest-rated | Most bookmarked

Sign up

Username:

Email address:

Password:

Password (type again to catch any typos):

I have read and agree to the Terms of Service:

Register

Fill out the form to the left (all fields are required), and your account will be created; you'll be sent an email with instructions on how to finish your registration.

Snippets
By author
By language
By tag
Highest-rated

About
Powered by Django.
Learn more about this site.
Read the FAQ.

Features

- User fills out form, inactive account created
- User gets email with link, clicks to activate
- And that's it

User registration

- Different sites want different information
- Different types of users
- Different signup workflow
- Etc., etc.

Some "simple" things aren't so simple.

Approach features skeptically

Should I add this feature?

- What does the application do?
- Does this feature have anything to do with that?
- No? Guess I shouldn't add it, then.

Top feature request in django-registration: User profiles

“

What does that have to do with user registration?

”

-- Me


No, You Can't Have a Pony
Not Yours

The solution?

django-profiles

- Add profile
- Edit profile
- View profile
- And that's it.

2

Don’t be afraid of multiple apps

The monolith mindset

- The “application” is the whole site
- Re-use is often an afterthought
- Tend to develop plugins that hook into the “main” application
- Or make heavy use of middleware-like concepts

The Django mindset

- Application == some bit of functionality
- Site == several applications
- Tend to spin off new applications liberally

Django encourages this

- Instead of one “application”, a list: `INSTALLED_APPS`
- Applications live on the Python path, not inside any specific “apps” or “plugins” directory
- Abstractions like the `Site` model make you think about this as you develop

Should this be its own application?

- Is it completely unrelated to the app's focus?
- Is it orthogonal to whatever else I'm doing?
- Will I need similar functionality on other sites?
- Yes? Then I should break it out into a separate application.

Unrelated features

- Feature creep is tempting: “but wouldn’t it be cool if…”
- But it’s the road to Hell
- See also: Part 1 of this talk

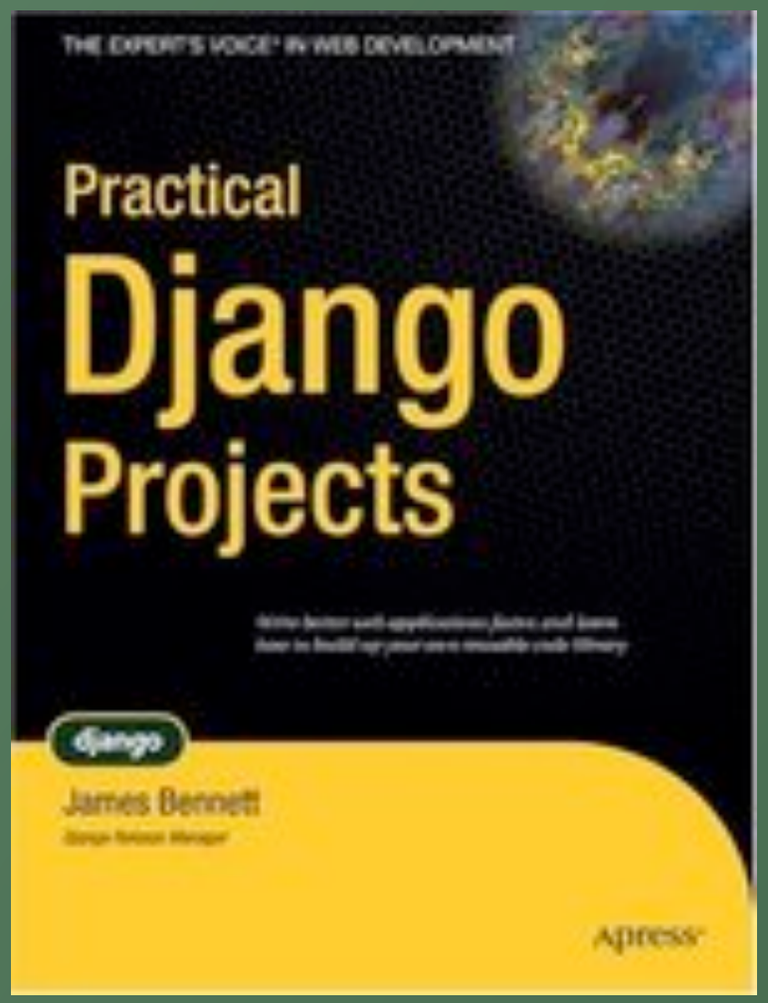
I've learned this the hard way

djangosnippets.org

- One application
- Includes bookmarking features
- Includes tagging features
- Includes rating features

Should be about four applications

So I wrote a book telling people not to do what I did



Page 210, in case you were wondering.

Orthogonality

- Means you can change one thing without affecting others
- Almost always indicates the need for a separate application
- Example: changing user profile workflow doesn't affect user signup workflow. Make them two different applications.

Reuse

- Lots of cool features actually aren't specific to one site
- See: bookmarking, tagging, rating...
- Why bring all this crap about code snippets along just to get the extra stuff?

Case study: blogging

I wanted a blog

- Entries and links
- Tagging
- Comments with moderation
- Contact form
- “About” page
- Etc., etc.

I ended up with

- A blog app (entries and links)
- A third-party tagging app
- `contrib.comments` + moderation app
- A contact-form app
- `contrib.flatpages`
- Etc., etc.

Advantages

- Don’t keep rewriting features
- Drop things into other sites easily

Need a contact form?

```
urlpatterns += (‘’,
    (r’^contact/’, include(‘contact_form.urls’)),
)
```

And you're done

But what about…

Site-specific needs

- Site A wants a contact form that just collects a message.
- Site B's marketing department wants a bunch of info.
- Site C wants to use Akismet to filter automated spam.

3

Write for flexibility

Common sense

- Sane defaults
- Easy overrides
- Don’t set anything in stone

Form processing

- Supply a form class
- But let people specify their own if they want

```
class SomeForm(forms.Form):
    ...

def process_form(request, form_class=SomeForm):
    if request.method == 'POST':
        form = form_class(request.POST)
        ...
    else:
        form = form_class()
    ...
```

Templates

- Specify a default template
- But let people specify their own if they want

```
def process_form(request, form_class=SomeForm,
                 template_name='do_form.html'):
    ...
    return render_to_response(template_name,
                              ...
```

Form processing

- You want to redirect after successful submission
- Supply a default URL
- But let people specify their own if they want

```
def process_form(request, form_class=SomeForm,
                 template_name='do_form.html',
                 success_url='/foo/'):
    ...
    return HttpResponseRedirect(success_url)
```

URL best practices

- Provide a URLConf in the application
- Use named URL patterns
- Use reverse lookups: `reverse()`, `permalink`, `{% url %}`

Working with models

- Whenever possible, avoid hard-coding a model class
- Use `get_model()` and take an app label/model name string instead
- Don't rely on `objects`; use the default manager

```
from django.db.models import get_model

def get_object(model_str, pk):
    model = get_model(*model_str.split('.'))
    return model._default_manager.get(pk=pk)

user_12 = get_object('auth.user', 12)
```

Working with models

- Don't hard-code fields or table names; introspect the model to get those
- Accept lookup arguments you can pass straight through to the database API

Learn to love managers

- Managers are easy to reuse.
- Managers are easy to subclass and customize.
- Managers let you encapsulate patterns of behavior behind a nice API.

Advanced techniques

- Encourage subclassing and use of subclasses
- Provide a standard interface people can implement in place of your default implementation
- Use a registry (like the admin)

The API your application exposes is just as important as the design of the sites you'll use it in.

In fact, it's **more**
important.

Good API design

- "Pass in a value for this argument to change the behavior"
- "Change the value of this setting"
- "Subclass this and override these methods to customize"
- "Implement something with this interface, and register it with the handler"

Bad API design

- "API? Let me see if we have one of those..." (AKA: "we don't")
- "It's open source; fork it to do what you want" (AKA: "we hate you")
- `def application(environ, start_response)` (AKA: "we have a web service")

No, really. Your gateway interface is **not** your API.

4

Build to distribute

So you did the tutorial

```
➜from mysite.polls.models import
  Poll
➜mysite.polls.views.vote
➜include(‘mysite.polls.urls’)
➜mysite.mysite.bork.bork.bork
```

Project coupling kills re-use

Why (some) projects suck

- You have to replicate that directory structure every time you re-use
- Or you have to do gymnastics with your Python path
- And you get back into the monolithic mindset

A good "project"

- A settings module
- A root URLConf module
- And that's it.

Advantages

- No assumptions about where things live
- No tricky bits
- Reminds you that it's just another Python module

It doesn't even have to be one module

ljworld.com

- `worldonline.settings.ljworld`
- `worldonline.urls.ljworld`
- And a whole bunch of reused apps in sensible locations

What reusable apps look like

- Single module directly on Python path (`registration`, `tagging`, etc.)
- Related modules under a package (`ellington.events`, `ellington.podcasts`, etc.)
- No project cruft whatsoever

And now it's easy

- You can build a package with `distutils` or `setuptools`
- Put it on the Cheese Shop
- People can download and install

General best practices

- Be up-front about dependencies
- Write for Python 2.3 when possible
- Pick a release or pick trunk, and document that
- But if you pick trunk, update frequently

Templates are hard

- Providing templates is a big “out of the box” win
- But templates are hard to make portable (block structure/inheritance, tag libraries, etc.)

I usually don't do default templates

Either way

- Document template names
- Document template contexts

Be obsessive about documentation

- It's Python: give stuff docstrings
- If you do, Django will generate documentation for you
- And users will love you forever

“

If the implementation is hard to explain, it’s a bad idea. If the implementation is easy to explain, it may be a good idea.

”

-- The Zen of Python

Documentation-driven development

- Write the docstring before you write the code
- Rewrite the docstring before you write the code
- And write doctests while you're at it

Advantages

- You'll never be lacking documentation
- It'll be up-to-date
- It's a lot easier to throw away a docstring than to throw away a bunch of code

Django will help you

- Docstrings for views, template tags, etc. can use reStructureText formatting
- Plus extra directives for handy cross-references to other components you're using

Recap:

- Do one thing, and do it well.
- Don’t be afraid of multiple apps.
- Write for flexibility.
- Build to distribute.

In the beginning...

- There was Django.
- And Ellington.
- And a couple other open-source apps.

...PyCon 2007...

- A few people presented/announced things they'd developed
- Sort of a watershed moment

...DjangoCon 2008

- Search for "django" on Google code hosting: **848** projects
- djangosites.org lists **1,636** sites
- And those are just the ones we know about so far...

“

This is Django’s killer feature.

”

-- Me

Good examples

- django-tagging (Jonathan Buchanan, http://code.google.com/p/django-tagging/)
- django-atompub (James Tauber, http://code.google.com/p/django-atompub/)
- Search for “django” on code hosting sites

More information

- django-hotclub (http://groups.google.com/group/django-hotclub/)
- Jannis Leidel's django-packages (http://code.google.com/p/django-reusableapps/)
- Django Pluggables: http://djangoplugables.com/

Questions?

Photo credits

- "Purple Sparkly Pony Ride" by ninjapoodles, http://www.flickr.com/photos/ninjapoodles/285048576/
- “Stonehenge #2” by severecci, http://www.flickr.com/photos/severecci/129553243/
- “sookiepose” by 416style, http://www.flickr.com/photos/sookie/41561946/
- "The Happy Couple" by galapogos, http://www.flickr.com/photos/galapogos/343592116/