
Django Patterns

Release 1.0

Corey Oordt

Sep 27, 2017

Contents

1	What are pluggable apps?	1
2	My Development Assumptions and Principles	3
2.1	Key Design Assumptions	3
2.2	Key Design Principles	4
3	Application Construction	7
3.1	Version Reporting	7
3.2	Views as a package	10
4	Configuration Patterns	13
4.1	Autodiscovery	13
4.2	Configurable Applications	14
5	Decorator App Patterns	19
5.1	Introduction to decorator applications	19
5.2	Adding fieldsets to a model's admin	21
5.3	Change the admin widget of a field	21
5.4	Lazy Field Insertion	23
5.5	Lazy Manager Insertion	25
6	Model Patterns	27
6.1	Abstract Model Mixins	27
6.2	Automatically Filling in a User in the Admin	29
6.3	Configurable Options for Common Situations	31
6.4	Configurable Foreign Keys	32
6.5	Flexible storage of uploaded files	32
7	Template Patterns	35
7.1	Easily overridable templates	35
7.2	Extend one template	35
7.3	Import your template blocks	36
8	URL patterns	37
8.1	URLs that live under any prefix	37

CHAPTER 1

What are pluggable apps?

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

Confusion between the “web app” the user sees and the Django apps that power it.

Pluggable apps are:

- Focused: focused use cases and include nothing that isn’t required. “Write programs that do one thing and do it well.” — Doug McIlroy (inventor of UNIX pipes)”
- Self-contained: Everything someone needs to get the app working is declared or included.
- Easily adaptable: A focused application can inevitably find new uses, if it doesn’t take too much for granted or make too many assumptions.
- Easily installed: Pluggable applications are installed and not modified. Applications are wired together and configured in the project. The only “Apps” in your project codebase are apps that are so specific to the project that they can’t be used elsewhere.

My Development Assumptions and Principles

by Corey Oordt

Since I started developing in Django in 2006, I've been lucky enough to meet and work with many talented people with a variety of experiences. Gradually, typically through failure or dire need, we developed a methodology and approach to development of projects.

The approach was internalized; each of us *knew* how it worked so it was never directly expressed. With a recent move to a new job, I was struck by the differences and needed to express the ideas in terms of **assumptions** and **principles**.

Assumptions are the preconceptions you and your team hold when approaching a project. These assumptions aren't necessarily bad. As long as you are aware of them, and regularly check to make sure they are valid, they will be helpful.

Principles are the guides to behavior. They are not hard and fast rules that you must never break, merely guides to success that should be understood and deviated from with full knowledge of why it makes sense in the situation.

Key Design Assumptions

No two projects are alike.

Each project will share some things with others, but not all projects will share the same things, or in the same way. You will need to listen to the needs of the end users and the people running the project.

Most projects will fail

It should fail quickly, with as little effort as possible to determine its imminent doom. This is not a pessimism, but innovation. The easier it is to try something to see if it works, the more you will try and the more that will work.

People have their own way of doing things.

I'm an opinionated bastard. As an opinionated bastard, it really torques me when others make me do things in ways I disagree with. When there are several ways to get the same result, let others get there by any of those means.

Another way to look at this is to manage *outcomes*, not practices or methods.

Two or more project's requirements may conflict.

The conflicts only matter if the projects must share something or are co-dependent.

Things change.

Life moves very fast on the internet. Projects must adapt quickly or fail miserably.

Key Design Principles

Design the user's experience first

Many developers create solutions that are functional, but not usable. When you start with the user's experience, you at least get more usable solutions, if not better ones.

Break down the tools to their simplest bits.

It is easier to find new uses for simple tools than complex tools. I am constantly surprised how people have found new ways to use code that was meant for something else.

Similar ideas are:

- Separate functionality whenever possible.
- Be flexible in implementation.
- Couple loosely.

Code is like a making a baby: only do it if you are willing to support it for the rest of your life.

I've learned the hard way that code you wrote years ago can come back to haunt you. I've received calls from people I wrote code for years before asking for help due to an unforeseen bug. (To me the unforeseen bug was that they were still using the software.) Unless you are willing to be a jerk, you got to give them a hand.

This leads to two practices: use external code libraries whenever possible to reduce the amount of code for which you are directly responsible, and write your code in a way that you won't be too horrified to see it in three years.

A similar practice is when making a utility to enhance functionality, don't assume that the implementer will "own the code".

External dependencies should be declared and few

I see dependencies like farts on an elevator: the fewer the better (and please confess).

If you want people to do something, make it incredibly easy to do.

And don't forget its sibling: *If you want people to do something a specific way, make it easier to do it that way than any other.*

Even small barriers will limit how often “best practices” are followed. People put off things if:

- They have to do prep work to accomplish the task
- They aren't sure how to accomplish the task
- They don't understand why the task is important

Version Reporting

Contributors: Corey Oordt, Stefan Foulis

- *What problem does this pattern solve?*
- *When to use it*
- *Why should I use it?*
- *Implementation*
- *How to use it*
 - *Inside your `setup.py` file*
 - *Inside your Sphinx documentation's `conf.py`*

What problem does this pattern solve?

It provides a flexible method of recording and reporting your application's version.

When to use it

You should use it with any application or project that has specific releases.

Why should I use it?

1. It is easy to see which version is currently installed somewhere.

2. It is easy to import the version into other places, like documentation or packaging.
3. It is easy for others to test the version of your code to better handle backwards-compatibility.

Implementation

PEP 386 defines the standard way to specify versions within the Python community. The most common scenario is the `Major.Minor.Micro` with a possible `alpha/beta/release candidate` suffix.

Examples:

```
1.0
0.6.1
2.1.1b1
0.3rc2
```

When recording your version number you should:

- Put it within the code, so it's accessible after the package is installed
- Easily retrieve all the individual parts of the version
- Record the individual version parts as integers (where appropriate) for easy comparison
- Have a properly formatted string version available

Putting the version information in your application's `__init__.py` is a great, out-of-the-way place.

Here is an example that conforms to PEP 386:

coolapp/__init__.py

```
1 __version_info__ = {
2     'major': 0,
3     'minor': 1,
4     'micro': 0,
5     'releaselevel': 'alpha',
6     'serial': 1
7 }
8
9 def get_version(short=False):
10     assert __version_info__['releaselevel'] in ('alpha', 'beta', 'final')
11     vers = ["%(major)i.%(minor)i" % __version_info__, ]
12     if __version_info__['micro']:
13         vers.append(":%(micro)i" % __version_info__)
14     if __version_info__['releaselevel'] != 'final' and not short:
15         vers.append(':%s%i' % (__version_info__['releaselevel'][0], __version_info__[
16 ↪ 'serial']))
17     return ''.join(vers)
18
19 __version__ = get_version()
```

This sets up a `__version_info__` dictionary to hold the version fields, a `get_version()` function to format the `__version_info__` into a string, and `__version__`, which is the formatted string version. It is similar to Django's method:

django/__init__.py

```
1 VERSION = (1, 4, 0, 'alpha', 0)
2
3 def get_version():
```

```

4 version = '%s.%s' % (VERSION[0], VERSION[1])
5 if VERSION[2]:
6     version = '%s.%s' % (version, VERSION[2])
7 if VERSION[3:] == ('alpha', 0):
8     version = '%s pre-alpha' % version
9 else:
10     if VERSION[3] != 'final':
11         version = '%s %s %s' % (version, VERSION[3], VERSION[4])
12 from django.utils.version import get_svn_revision
13 svn_rev = get_svn_revision()
14 if svn_rev != u'SVN-unknown':
15     version = "%s %s" % (version, svn_rev)
16 return version

```

How to use it

Inside your setup.py file

The `setup.py` file needs a version for your application and you can import it directly from your application, as seen in this example taken from `django-app-skeleton`'s `setup.py` file:

django-app-skeleton/skel/setup.py

```

1 # ... other stuff above
2
3 setup(
4     name = "$$$$APP_NAME$$$",
5     version = __import__('$$$PKG_NAME$$$').get_version().replace(' ', '-'),
6     url = '$$$$URL$$$ ',
7     author = '$$$$AUTHOR$$$ ',
8     author_email = '$$$$AUTHOR_EMAIL$$$ ',
9     description = DESC,
10    long_description = get_readme(),
11    packages = find_packages(),
12    include_package_data = True,
13    install_requires = read_file('requirements.txt'),
14    classifiers = [
15        'License :: OSI Approved :: Apache Software License',
16        'Framework :: Django',
17    ],
18 )

```

Inside your Sphinx documentation's conf.py

Sphinx also likes to have the version of your application in the formatted documentation. Since the `conf.py` configuration file is just Python, you can import your version.

coolapp/docs/conf.py

```

1 sys.path.append(os.path.abspath('.'))
2 os.environ['DJANGO_SETTINGS_MODULE'] = 'example.settings'
3
4 import coolapp
5
6 # The version info for the project you're documenting, acts as replacement for

```

```
7 # |version| and |release|, also used in various other places throughout the
8 # built documents.
9 #
10 # The short X.Y version.
11 version = coolapp.get_version(short=True)
12 # The full version, including alpha/beta/rc tags.
13 release = coolapp.get_version()
```

Views as a package

Contributors: Corey Oordt

- *What problem does this pattern solve?*
- *When to use it*
- *Why should I use it?*
- *Implementation*
 - *Python imports briefly*
- *Sources*

What problem does this pattern solve?

Code in `views.py` has become unmanageable.

When to use it

You want to refactor your views into several files.

Why should I use it?

This pattern allows for refactoring the view code into several files without effecting the import process in other files. In other words `from coolapp.views import foo` still works.

Implementation

Note: When refactoring your code into multiple files, look deeper and see if there are better ways to accomplish the tasks, such as using generic views.

Python imports briefly

This pattern takes advantage of the way that Python handles importing items into the local namespace. The statement `from foo import views` will work with code organized as:

Example 1

```
foo
- __init__.py
- views.py
```

as well as:

Example 2

```
foo
- __init__.py
- views
  - __init__.py
```

In the case of Example 2, the contents of `foo/views/__init__.py` is executed. The `__init__.py` file is going to be important in the switch from a [module](#) (one file) to a [package](#) (directory with `__init__.py`).

First rename `views.py` to something like `old_views.py` to prevent name confusion. Second create the `views` directory and add an `__init__.py` file. Then refactor the `old_views.py` into two or more files. See Example 3.

Example 3

```
foo
- __init__.py
- old_views.py
- views
  - __init__.py
  - bar.py
  - baz.py
```

Note: When refactoring your views, you will probably need to change imports from other modules in your app, such as models. The statement `from models import Foo` will no longer work since the `models.py` file is not in the same directory.

Instead, you will need to use a full path import: `from foo.models import Foo`.

Now, to make imports such as `from views import bar_detail_view` work, we need to add a couple of lines to `views/__init__.py`

views/__init__.py

```
1 from bar import *
2 from baz import *
```

These statements import all the contents of `views.bar` and `views.baz` into `views`. You can limit what is imported with `*` defining a list named `__all__` (see [Importing * from a Package](#)) within the module.

`__all__` it is taken to be the list of names that should be imported when `from module import *` is encountered. Django uses this often, such as in `django.conf.urls.defaults`.

Attention: It is up to you to maintain the `__all__` list as you update the file.

Sources

<http://stackoverflow.com/questions/2675722/django-breaking-up-views>

Configuration Patterns

Autodiscovery

Warning: This is just a stub document. It will be fleshed out more.

What problem does this pattern solve?

An app provides a service that requires complex configuration or customization by other apps to properly use it.

When to use it

Why should I use it?

Implementation

For each app, we need to look for an specific module inside that app's package. We can't use `os.path` here – recall that modules may be imported different ways (think zip files) – so we need to get the app's `__path__` and look for the module on that path.

Step 1: find out the app's `__path__` Import errors here will (and should) bubble up, but a missing `__path__` (which is legal, but weird) fails silently – apps that do weird things with `__path__` might need to roll their own index registration.

Step 2: use `imp.find_module` to find the app's `search_indexes.py`. For some reason `imp.find_module` raises `ImportError` if the app can't be found but doesn't actually try to import the module. So skip this app if its `search_indexes.py` doesn't exist

Step 3: import the app's `search_index` file. If this has errors we want them to bubble up.

Django Snippet 2404: Generic Autodiscovery

```
1 def generic_autodiscover(module_name):
2     """
3     Usage:
4     generic_autodiscover('commands') <-- find all commands.py and load 'em
5     """
6     import imp
7     from django.conf import settings
8
9     for app in settings.INSTALLED_APPS:
10         try:
11             import_module(app)
12             app_path = sys.modules[app].__path__
13             except AttributeError:
14                 continue
15         try:
16             imp.find_module(module_name, app_path)
17             except ImportError:
18                 continue
19         import_module('%s.%s' % (app, module_name))
20         app_path = sys.modules['%s.%s' % (app, module_name)]
```

How to use it

Sources

Useful Links

- [Generic Autodiscovery](#)
- [Looking at registration patterns in Django](#)
- [django-config-wizard autodiscover.py](#)

Where is it used?

Configurable Applications

Contributors: Corey Oordt

- *What problem does this pattern solve?*
- *When to use it*
- *Implementation*
 - *Basic Pattern with one setting*
 - *Requiring a value for a setting*
 - *Many settings for your application*
 - *Settings with nested dictionaries*
 - *Turning the keys into attributes*

- *How to use it*

What problem does this pattern solve?

You want to allow configuration of your app without having to modify its code. You may also want to provide reasonable defaults that users can override.

When to use it

Use this whenever project- or implementation-specific information is required at runtime or there are obvious choices or options for the application.

Good examples:

- API key
- Debugging flags
- Location(s) to look for files
- Which features should be used (feature flags)

Implementation

Create a `settings.py` file in your application

```
coolapp
- __init__.py
- admin.py
- models.py
- settings.py
- tests.py
- views.py
```

Basic Pattern with one setting

Inside the `settings.py` file, you will import Django's settings and use `getattr()` to retrieve the value, or use a default value. There are several parts to this:

- **Internal Name:** The name you will use within your application
- **Namespaced Name:** The name used in a project's `settings.py`, with a prefix to avoid collisions.
- **Default Value:** The value for this setting if the namespaced name is not in the project's `settings.py`.

`coolapp/settings.py`

```
1 from django.conf import settings
2
3 COOL_WORD = getattr(settings, 'COOLAPP_COOL_WORD', 'cool')
```

Here, `COOL_WORD` is the *internal name*, `COOLAPP_COOL_WORD` is the *namespaced name*, and `'cool'` is the *default value*.

Requiring a value for a setting

For something like an API key, you will want to draw attention if it's empty. You will do this by raising an `ImproperlyConfigured` exception.

coolapp/settings.py

```
1 from django.conf import settings
2 from django.core.exceptions import ImproperlyConfigured
3
4 API_KEY = getattr(settings, 'COOLAPP_API_KEY', None)
5
6 if API_KEY is None:
7     raise ImproperlyConfigured("You haven't set 'COOLAPP_API_KEY'.")
```

Many settings for your application

Django has internally began using dictionaries for groups of settings, such as DATABASES. Django debug toolbar, for example, uses one dictionary to store all its configurations.

debug_toolbar/toolbar/loader.py

```
1 """
2 The main DebugToolbar class that loads and renders the Toolbar.
3 """
4 from django.conf import settings
5 from django.template.loader import render_to_string
6
7 class DebugToolbar(object):
8
9     def __init__(self, request):
10         self.request = request
11         self.panels = []
12         base_url = self.request.META.get('SCRIPT_NAME', '')
13         self.config = {
14             'INTERCEPT_REDIRECTS': True,
15             'MEDIA_URL': u'%s/___debug___/m/' % base_url
16         }
17         # Check if settings has a DEBUG_TOOLBAR_CONFIG and updated config
18         self.config.update(getattr(settings, 'DEBUG_TOOLBAR_CONFIG', {}))
19
20         # ... more code below
```

It creates a standard set of configurations in line 13, and then uses the dictionaries `update()` method in line 18 to add or override current key/values.

Settings with nested dictionaries

If your settings dictionary has a dictionary as a value, you need to take a slightly different approach. `dict.update()` will completely overwrite the nested dictionaries, not merge them. To make things trickier, `dict.update()` doesn't return a value, so

```
1 DEFAULT_SETTINGS.update(getattr(settings, 'FOOBAR_SETTINGS', {}))
2 DEFAULT_SETTINGS['FOO'] = DEFAULT_FOO.update(DEFAULT_SETTINGS.get('FOO', {}))
```

leaves `DEFAULT_SETTINGS['FOO']` with a value of `None`. So lets try something else.

supertagging/settings.py

```

1  DEFAULT_SETTINGS = {
2      'ENABLED': False,
3      'DEBUG': False,
4
5      # ... other settings
6  }
7
8  DEFAULT_MARKUP_SETTINGS = {
9      'ENABLED': False,
10     'FIELD_SUFFIX': "tagged",
11     'EXCLUDE': [],
12     'CONTENT_CACHE_TIMEOUT': 3600,
13     'MIN_RELEVANCE': 0,
14 }
15
16 temp_settings = getattr(settings, 'SUPERTAGGING_SETTINGS', {})
17 USER_SETTINGS = dict(DEFAULT_SETTINGS.items() + temp_settings.items())
18 USER_SETTINGS['MARKUP'] = dict(
19     DEFAULT_MARKUP_SETTINGS.items() + USER_SETTINGS.get('MARKUP', {}).items()
20 )

```

In this example taken from `django-supertagging`, line 8 shows the default values for `SUPERTAGGING_SETTINGS['MARKUP']`. Line 16 retrieves the `SUPERTAGGING_SETTINGS` dictionary into a temporary variable using `getattr`.

Line 17 merges the `DEFAULT_SETTINGS` dictionary with the dictionary retrieved in line 16 into a new copy. By converting each dictionary into a list of tuple-pairs with the `items()` method, it can combine them using the `+` operator. When this list is converted back into a dictionary, it uses the last found key-value pair.

Lines 18-20 merge the defaults for `MARKUP` with whatever the user has specified.

Turning the keys into attributes

Having one dictionary for all your applications settings is all well and good, but requires more typing. Instead of typing:

```

1  from settings import USER_SETTINGS
2  if USER_SETTINGS['ENABLED']:
3      # do something
4      pass

```

it would be nice to type:

```

1  from settings import ENABLED
2  if ENABLED:
3      # do something
4      pass

```

What we want to do is convert the first set of keys into variables. Python has a built-in function called `globals()` that returns a dictionary of the symbol table of the current module.

If you printed the value of `globals()` in an empty Python script, you would see something like:

```
>>> print globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
```

Since `globals()` returns a dictionary, you can use its `update()` method to alter its contents in place.

```
>>> d = {'foo': 1, 'bar': 2}
>>> globals().update(d)
>>> print globals()
{'bar': 2, 'd': {'foo': 1, 'bar': 2}, '__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'foo': 1, '__doc__': None, '__package__': None}
```

Warning: While the effect of this is localized to this module, you must be careful with the names of the dictionary keys. If there is a naming conflict, your dictionary wins. That is probably not what you want.

Using the supertagging example above, adding:

```
globals().update(USER_SETTINGS)
```

to the bottom of the `supertagging/settings.py` file, gives you access to all the top-level keys of `USER_SETTINGS` as attributes of `settings.py`

How to use it

Access to your settings is a simple import. As shown in the previous section, you can import the entire dictionary of settings or, if you added them to the settings module, you can import them individually.

Decorator App Patterns

Introduction to decorator applications

Contributors: Corey Oordt

- *What is a decorator app?*
- *Benefits*
 - *Abstraction of metadata*
 - *Metadata aggregation*
 - *Metadata customization*
 - *Alternative data handling*
- *Warnings*

What is a decorator app?

A decorator app is an application that adds functionality to another application without the application's awareness. It is different than a Python decorator, rather it follows the idea from [Design Patterns: Elements of Reusable Object-Oriented Software](#):

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Also Known As: Wrapper

Motivation: Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.

One way to add responsibilities is with inheritance. Inheriting a border from another class puts a border around every subclass instance. This is inflexible, however, because the choice of border is made statically. A client can't control how and when to decorate the component with a border.

It is different than a true decorator in that Django has no way to wrap models or applications. Instead of wrapping the model, and because we're using a dynamic language like Python, we will inject independent code into the model at runtime.

Benefits

Abstraction of metadata

When developing an application to manage data, images for example, you include data and metadata. The data is the image, or path to that image. Any other information is metadata.

For this image application, how much metadata do you include? Some metadata may seem straightforward enough to include: name, width, height, resolution and format come to mind. What about less common things such as author, usage license, categories, and tags? And some of that metadata might be shared across other data applications. A video application might also include usage license, categories and tags. Should each application store their metadata separately?

You can design data applications that store minimal amounts of metadata (metadata that is easily extracted from the image, for example) and leave other metadata to specialized decorator applications.

Metadata aggregation

It is likely that you would want to manage taxonomy metadata like categories or tags the same way throughout a project. It's rather cumbersome if every third-party application allows for a different system for handling it. A decorator application can provide a single way to manage that metadata and aggregate it throughout a project. It is easy then to query all objects, across all applications, that are tagged *foo*, or are categorized as *bar*.

Metadata customization

"No two projects are alike." I always say, and that includes how they want to handle metadata. A checkbox stating you have reproduction rights might work in one project while another requires a much more specific licensing description. A decorator app for licensing allows the image application to ignore that bit of metadata in its code. When both apps are included in a project, however, the same image application can show different licensing options, depending on the project configuration.

Alternative data handling

Decorators aren't just good for metadata; they can also alter how the data is managed. Take an application to handle blog entries, for example. The primary data is the text of the entry. A good question for the application is "How does the user enter text into that field?" Most apps force a decision on the user, such as a markup language such as Textile, reStructuredText, or Markdown or a WYSIWYG editor like TinyMCE.

If no two projects are alike, might that also include text formatting? In one project, the users might want a WYSIWYG editor, while others prefer a specific markup language. A decorator app can manage that for the data app, especially if the data app includes some hooks to make it easier.

Warnings

Django doesn't have any native way to add functionality to other applications. Therefore accomplishing this task requires modifying class definitions at runtime. Depending on the amount and type of modification, there could be unforeseen consequences.

All the patterns discussed are used in production and heavily tested. Each pattern does significant error checking to make sure any inserted code doesn't clobber existing functionality.

Adding fieldsets to a model's admin

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

```
for model, modeladmin in admin.site._registry.items():
    if model in model_registry.values() and modeladmin.fieldsets:
        fieldsets = getattr(modeladmin, 'fieldsets', ())
        fields = [cat.split('.')[2] for cat in registry if registry[cat] == model]
        # check each field to see if already defined
        for cat in fields:
            for k,v in fieldsets:
                if cat in v['fields']:
                    fields.remove(cat)
        # if there are any fields left, add them under the categories fieldset
        if len(fields) > 0:
            print fields
            admin.site.unregister(model)
            admin.site.register(model, type('newadmin', (modeladmin.__class__,), {
                'fieldsets': fieldsets + (('Categories', {
                    'fields': fields
                })),
            }))
        )))
```

Change the admin widget of a field

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

Django TinyMCE allows you to add TinyMCE functionality to your app if you make certain modifications to your app. This is great if it is your code. However, it doesn't work so well, if it is someone else's code. Justin forked Django-TinyMCE to provide this lazy customization.

The configuration is simple: the app.model name is the key, and then value is a list of fields to have TinyMCE on in the admin.

```
TINYMCE_ADMIN_FIELDS = {
    'app1.model1': ('body',),
    'app1.model2': ('blog_text', 'blog_teaser')
}
```

There are several steps to this process.

The first is creating a REGISTRY variable to hold the Model and field specifications in our settings.py

```
from django.db.models import get_model
import django.conf import settings

REGISTRY = {}
ADMIN_FIELDS = getattr(settings, 'TINYMCE_ADMIN_FIELDS', {})

for model_name, field in ADMIN_FIELDS.items():
    if isinstance(model_name, basestring):
        model = get_model(*model_name.split('.'))
        if model in registry:
            return
        REGISTRY[model] = field
```

Next in our admin.py, we declare a Model admin class, with one new attribute: editor_fields. We are also going to override a standard model admin method:

formfield_for_dbfield. This is the method that given a database field will return the form field to render.

our overridden method checks to see if this field is in our list of editor_fields, and if so, returns a version using the TinyMCE widget.

if the field is not in our list, we punt it back to the super class.

admin.py

```
# Define a new ModelAdmin subclass

class TinyMCEAdmin(admin.ModelAdmin):
    editor_fields = ()

    def formfield_for_dbfield(self, db_field, **kwargs):
        if db_field.name in self.editor_fields:
            return db_field.formfield(widget=TinyMCE())
        return super(TinyMCEAdmin, self).formfield_for_dbfield(
            db_field, **kwargs)
```

œœ Finally, we put the two pieces together. At the bottom of admin.py we loop through the admin's current admin registry.

Check if the current iteration is in our registry

if it is, we unregister that model's current admin

and then re-register the model with a dynamically-created class called newadmin

that is a subclass of our previously declared admin and the model's current admin

and we set that new class's editor-fields attribute to the fields in our registry

admin.py

```
for model, modeladmin in admin.site._registry.items():
    if model in REGISTRY:
        admin.site.unregister(model)
        admin.site.register(
            model,
            type('newadmin',
                (TinyMCEAdmin, modeladmin.__class__),
```

```

        {'editor_fields': REGISTRY[model], }
    )
)

```

Lazy Field Insertion

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

The idea is allow developers to decide which models will have categories in the project's settings.py, using a dictionary with the model as the key and the field or fields as the value.

```

'FK_REGISTRY': {
    'flatpages.flatpage': 'category',
    'simpletext.simpletext': (
        'primary_category',
        {'name': 'secondary_category', 'related_name': 'simpletext_sec_cat'},
    ),
},
'M2M_REGISTRY': {
    'simpletext.simpletext': {'name': 'categories', 'related_name': 'm2mcats'},
    'flatpages.flatpage': (
        {'name': 'other_categories', 'related_name': 'other_cats'},
        {'name': 'more_categories', 'related_name': 'more_cats'},
    ),
},

```

at the bottom of the category app's `__init__.py`, you can read the configuration from settings.

loop through them

Do a bit of error checking

Load the model class

Loop through the given fields

We make sure that the field doesn't already exist by attempting to get it

Finally we add the field to the model by instantiating the field and calling its `contribute_to_class` method.

```

import fields

from django.db.models import FieldDoesNotExist

class AlreadyRegistered(Exception):
    """
    An attempt was made to register a model more than once.
    """
    pass

# The field registry keeps track of the individual fields created.
# {'app.model.field': Field(**extra_params)}
# Useful for doing a schema migration
field_registry = {}

```

```

# The model registry keeps track of which models have one or more fields
# registered.
# {'app': [model1, model2]}
# Useful for admin alteration
model_registry = {}

def register_m2m(model, field_name='categories', extra_params={}):
    return _register(model, field_name, extra_params, fields.CategoryM2MField)

def register_fk(model, field_name='category', extra_params={}):
    return _register(model, field_name, extra_params, fields.CategoryFKField)

def _register(model, field_name, extra_params={}, field=fields.CategoryFKField):
    app_label = model._meta.app_label
    registry_name = ".".join((app_label, model.__name__, field_name)).lower()

    if registry_name in field_registry:
        return #raise AlreadyRegistered
    opts = model._meta
    try:
        opts.get_field(field_name)
    except FieldDoesNotExist:
        if app_label not in model_registry:
            model_registry[app_label] = []
        if model not in model_registry[app_label]:
            model_registry[app_label].append(model)
        field_registry[registry_name] = field(**extra_params)
        field_registry[registry_name].contribute_to_class(model, field_name)

from categories import settings
from django.core.exceptions import ImproperlyConfigured
from django.db.models import get_model

for key, value in settings.FK_REGISTRY.items():
    model = get_model(*key.split('.'))
    if model is None:
        raise ImproperlyConfigured('%s is not a model' % key)
    if isinstance(value, (tuple, list)):
        for item in value:
            if isinstance(item, basestring):
                register_fk(model, item)
            elif isinstance(item, dict):
                field_name = item.pop('name')
                register_fk(model, field_name, extra_params=item)
            else:
                raise ImproperlyConfigured("CATEGORY_SETTINGS['FK_REGISTRY'] doesn't
↳ recognize the value of %s" % key)
    elif isinstance(value, basestring):
        register_fk(model, value)
    elif isinstance(item, dict):
        field_name = item.pop('name')
        register_fk(model, field_name, extra_params=item)
    else:
        raise ImproperlyConfigured("CATEGORY_SETTINGS['FK_REGISTRY'] doesn't
↳ recognize the value of %s" % key)
for key, value in settings.M2M_REGISTRY.items():
    model = get_model(*key.split('.'))

```

```

if model is None:
    raise ImproperlyConfigured('%s is not a model' % key)
if isinstance(value, (tuple, list)):
    for item in value:
        if isinstance(item, basestring):
            register_m2m(model, item)
        elif isinstance(item, dict):
            field_name = item.pop('name')
            register_m2m(model, field_name, extra_params=item)
        else:
            raise ImproperlyConfigured("CATEGORY_SETTINGS['M2M_REGISTRY'] doesn't
↳recognize the value of %s: %s" % (key, item))
    elif isinstance(value, basestring):
        register_m2m(model, value)
    elif isinstance(value, dict):
        field_name = value.pop('name')
        register_m2m(model, field_name, extra_params=value)
    else:
        raise ImproperlyConfigured("CATEGORY_SETTINGS['M2M_REGISTRY'] doesn't
↳recognize the value of %s" % key)

```

Lazy Manager Insertion

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

Loosely based on Django-mptt. It is very similar to how we handled inserting a field.

```

COOLAPP_MODELS = {
    'app1.Model': 'cool_manager',
    'app2.Model': 'cool_manager',
}

```

At the bottom of this app's models.py, you can read the configuration from settings.

loop through them and Do a bit of error checking

Load the model class

Loop through the given fields

We make sure that the model doesn't have an attribute by the same name, we add the field to the model by instantiating the manager and calling its `contribute_to_class` method.

```

from django.db.models import get_model
import django.conf import settings
from coolapp.managers import CustomManager

MODELS = getattr(settings, 'COOLAPP_MODELS', {})

for model_name, mgr_name in MODELS.items():
    if not isinstance(model_name, basestring):
        continue

    model = get_model(*model_name.split('.'))

```

```
if not getattr(model, mgr_name, False):  
    manager = CustomManager()  
    manager.contribute_to_class(model, mgr_name)
```

Abstract Model Mixins

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

What problem does this pattern solve?

It creates a tool kit to build complex models.

When to use it

For models that you will commonly build in projects but have many different potential features, and you want your models to only contain the features necessary. Blogs are a good example, where there are many potential options to include within a blog, but you don't need all of them all the time.

Why should I use it?

This allows developers to fix bugs once, in the tool kit. Installing the new tool kit version will fix those bugs in each model created from it.

Where is it used?

I first saw this in [GLAMkit's blogtools](#) package. It is also used in Django 1.3's [class-based views](#).

Implementation

Models

First isolate all the potential or optional features from core features. The core features and each isolated set of optional features will make up individual abstract Django models.

GLAMkit isolated the one base model (`EntryBase`), and four optional features: add a *featured* flag, add a *status* field, add a tag field, and allow for the body and excerpt content to convert to HTML.

`EntryBase` includes seven fields—`author`, `title`, `pub_date`, `slug`, `enable_comments`, `excerpt`, and `body`—as well as `__unicode__()`, `get_absolute_url()` and `excerpt_or_body()` functions. The `Meta` class has `abstract=True` so that Django never tries to represent this model in a database. It must be subclassed.

`FeaturableEntryMixin` is an abstract class that merely defines an `is_featured` field.

`StatusableEntryMixin` is an abstract class that defines `LIVE_STATUS`, `DRAFT_STATUS`, and `HIDDEN_STATUS` values and choices. It defines a `status` field with those choices.

`TaggableEntryMixin` is an abstract class that is only available if Django-Tagging is installed, as it uses the `TagField` for the `tags` field it defines.

`HTMLFormattableEntryMixin` is a much more complex abstract class. It is only available if the `template_utils` package is available. It defines two text fields, `excerpt_html` and `body_html`. It also overrides the `save()` method so it can convert the contents of `excerpt` and `body` into HTML for `excerpt_html` and `body_html`, respectively. Finally it re-defines the `excerpt_or_body()` method to return the `excerpt_html` or `body_html` value.

Admin

It is difficult to provide a really good `ModelAdmin` class when you aren't sure what fields or features are included in the final model. GLAMkit provides a `EntryAdminBase` which is subclassed from `object` (not `ModelAdmin`). Providing other admin mixins would make sense if there were admin-specific features to provide, such as adding a WYSIWYG editor, autocomplete lookups or special filtering.

URLs

Views

Syndication Feeds

How to use it

Sources

Useful Links

GLAMKit (Gallery, Library, Archive, Museum) an Australian group, tackles this situation with a set of abstract classes that provide very basic features.

You create your model by subclassing the classes that provide the functionality you need.

And you don't have to stop there. You can add your own fields as well.


```
class PRBlog(EntryBase,
             StatusableEntryMixin):

    subhead = models.CharField()
    pdf = models.FileField()
```

Automatically Filling in a User in the Admin

What problem does this pattern solve?

- Easily keep track of the last user who modified a record
- Automatically set the “author” of a record

Why should I use it?

It is a safe way to save the user time and still keep track of the information.

Implementation

The important parts of the implementation are:

1. The ForeignKey to User field needs `blank=True`
2. Create a ModelForm that assigns a fake User to the field.
3. Override the `ModelAdmin.save_model()` function to assign the `request.user` to the field.

For discussion we'll use this model with two relations to the User model:

coolblog/models.py

```
1 class Entry(models.Model):
2     title = models.CharField(max_length=250)
3     slug = models.SlugField()
4     pub_date = models.DateField(default=datetime.datetime.today)
5     author = models.ForeignKey(
6         User,
7         related_name='entries',
8         blank=True)
9     body = models.TextField()
10    last_modified = models.DateTimeField(auto_now=True)
11    last_modified_by = models.ForeignKey(
12        User,
13        related_name='entry_modifiers',
14        blank=True)
```

This Entry model has two fields that we want to fill automatically: `author` and `last_modified_by`. Notice both fields have `blank=True`. This is important so we can get past some initial Django validation.

Faking validation

Whenever you save a model, Django attempts to validate it. Validation will fail without special validation tomfoolery. The first stage of validation fakery was setting `blank=True` on the fields. The next stage involves setting a temporary value for each field.

coolblog/forms.py

```
1 from django.contrib.auth.models import User
2
3 class EntryForm(forms.ModelForm):
4     class Meta:
5         model = Entry
6
7     def clean_author(self):
8         if not self.cleaned_data['author']:
9             return User()
10        return self.cleaned_data['author']
11
12    def clean_last_modified_by(self):
13        if not self.cleaned_data['last_modified_by']:
14            return User()
15        return self.cleaned_data['last_modified_by']
```

The lower-level Django model validation actually checks if the value of a related field is an instance of the correct class. Since a form's validation happens before any attempt to save the model, we create a new `ModelForm`, called `EntryForm`. The `clean_author()` and `clean_last_modified_by()` methods check for an empty value and assigns it an unsaved and empty instance of `User`, and Django is happy.

Saving the model

In the model's `ModelAdmin`, we make a few adjustments.

coolblog/admin.py

```
1 class EntryAdmin(admin.ModelAdmin):
2     form = EntryForm
3
4     list_display = ('title', 'pub_date', 'author')
5     prepopulated_fields = {'slug': ['title']}
6     readonly_fields = ('last_modified', 'last_modified_by',)
7     fieldsets = ((
8         None, {
9             'fields': ('title', 'body', 'pub_date')
10        }), (
11        'Other Information', {
12            'fields': ('last_modified', 'last_modified_by', 'slug'),
13            'classes': ('collapse',)
14        })
15    )
16
17    def save_model(self, request, obj, form, change):
18        if not obj.author.id:
19            obj.author = request.user
20        obj.last_modified_by = request.user
21        obj.save()
```

First, we set the form attribute to the `EntryForm` we just created.

Then, since we don't want the author to worry about selecting themselves in the `author` field, we left it out of the fieldsets. We left in the `last_modified_by` field for reference and made it a read-only field.

The final magic comes in the overridden `save_model()` method on line 17. We check to see if the `author` attribute actually has an `id`. If it doesn't, it must be the empty instance we set in `EntryForm`, so we assign the `author` field to the current user. Since we always want to assign or re-assign the user modifying this record, the `last_modified_by` field is set every time.

Sources

[ModelAdmin.save_model documentation](#)

[Audit Fields](#)

[Users and the admin](#)

Configurable Options for Common Situations

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail [coreyoordt at gmail](mailto:coreyoordt@gmail.com).

A few, well-known of variations (e.g. Use `django.contrib.sites`?)

models.py

```

1  from django.db import models
2  from coolapp.settings import MULTIPLE_SITES, SINGLE_SITE
3
4  if MULTIPLE_SITES or SINGLE_SITE:
5      from django.contrib.sites.models import Site
6
7  class Entry(models.Model):
8      title = models.CharField(max_length=100)
9      # Other stuff
10
11     if MULTIPLE_SITES:
12         sites = models.ManyToManyField(Site)
13     if SINGLE_SITE:
14         sites = models.ForeignKey(Site)
```

Another example:

models.py

```

1  from django.db import models
2  from myapp.settings import USE_TAGGING
3
4  if USE_TAGGING:
5      from tagging.fields import TagField
6
7  class Entry(models.Model):
8      title = models.CharField(max_length=100)
9      # Other stuff
10
```

```
11     if USE_TAGGING:
12         tags = TagField()
```

You can provide for optional field settings.

Import the setting from your own apps settings

Based on that setting, you can optionally import classes. And in your model definition...

Optionally declare fields. The only drawback of this depends on the type of field. Changing your mind after the initial table creation might require you to either manually add the field or drop the table and syncdb again.

[Link to way to do migration with south if field is added.](#)

Configurable Foreign Keys

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

We had a staff model that we wanted to related it to in some projects, but not all. So in the application settings we use a django function called `get_model`. This allows you to specify the model in an app-dot-model format in the project settings and then dynamically import it

```
from django.conf import settings
from django.db.models import get_model

model_string = getattr(settings, 'VIEWPOINT_AUTHOR_MODEL', 'auth.User')
AUTHOR_MODEL = get_model(*model_string.split('.'))
```

Now we simply import the `AUTHOR_MODEL` setting, which is a django model. And use it as the parameter for the `ForeignKey` field.

```
from viewpoint.settings import AUTHOR_MODEL

class Entry(models.Model):
    title = models.CharField(max_length=100)
    author = models.ForeignKey(AUTHOR_MODEL)
    ...
```

Flexible storage of uploaded files

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

```
from django.conf import settings
from django.core.files.storage import get_storage_class

DEFAULT_STORAGE = get_storage_class(
    getattr(settings, "MMEDIA_DEFAULT_STORAGE", settings.DEFAULT_FILE_STORAGE)
)
```

```
from massmedia.settings import IMAGE_UPLOAD_TO, DEFAULT_STORAGE

class Image(models.Model):
    file = models.FileField(
        upload_to = IMAGE_UPLOAD_TO,
        blank = True,
        null = True,
        storage=DEFAULT_STORAGE())
```


Easily overridable templates

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

Templates are an important part of your pluggable app. They demonstrate how your app works. The more complex your app, the more important templates are. Following a few practices can make your templates very useful. We have two goals: 1. Get demonstrable functionality in as short a time as possible, and 2. modify the fewest templates to do so.

Instead of putting your templates loose in your templates directory where they can conflict with other apps templates, *** put them in a directory within templates, named after your app to name space them.** * Then you reference them as the relative path from templates, in this case: coolapp/base.html

```
coolapp
+-templates
  +-coolapp
    +-base.html
```

Extend one template

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

If all your templates extend a template that you assume exists, such as `base.html`

You have to change each template when your project uses `site_base.html` instead.

If instead all your templates extend a known, base template in your name space and it extends the mythical `base.html`

when the inevitable happens and the base template name changes changing one template makes all the others work.

If your `coolapp/base.html` defines all the blocks that you use, it is also trivial to change them to match the project's template, just by enclosing your blocks in the appropriate base template blocks

`coolapp/base.html`

```
{% extends "site_base.html" %}

{% block extra_head %}
    {% block head %}
    {% endblock %}
{% endblock %}

{% block content %}
    {% block body %}
    {% endblock %}
{% endblock %}
```

Import your template blocks

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail [coreyoordt at gmail](mailto:coreyoordt@gmail.com).

If each template only focuses on one template block and imports other blocks, such as extra header information, you can selectively choose which template to modify: `extra_head.html` to add this functionality once for all templates, or `detail.html` to change the content.

```
{% extends "coolapp/base.html" %}

{% block extra_head %}
    {{ block.super }}
    {% import "coolapp/extra_head.html" %}
{% endblock %}

{% block content %}
    {# Important content stuff here #}

{% endblock %}
```


URLs that live under any prefix

Warning: This is just a stub document. It will be fleshed out more. If you wish to comment on it, please e-mail coreyoordt at gmail.

It's bad practice to hard code url paths or assume certain paths, for example that my blog app is always going to be under the path /blogs/

Django provides an easy way to abstractly reference a url and its view. All you have to do ...

Is add a url function in front of the pattern and add a name parameter. This allows you to ...

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    url(r'^$', 'coolapp_app.views.index', name='coolapp_index'),
)
```

```
<p>Go to the <a href="{% url coolapp_index %}">Index</a></p>
```

Retrieve the url using the url template tag or use the reverse function within your code.

```
from django.core.urlresolvers import reverse
```

```
def myview(request):
    return HttpResponseRedirect(reverse('coolapp_index', args=[]))
```