Django design patterns Documentation

Release 0.2

Agiliq and Contributors

April 13, 2018

Contents

This is a collection of patterns which we have found occurring commonly with Django. All of these either make collaboration easier, coding simpler or code more maintainable. None of them are design patterns in the sense of GoF design patterns. We call them design patterns as none other seem closer or more convenient.

These are guidelines, which need to be overridden (very commonly, in some cases). Use your judgement when using them. As PEP8 says, “Foolish consistency is the hobgoblin of small minds.”

Contents:

CHAPTER 1

Chapters

1.1 Django Design Patterns

1.1.1 Prerequisites

1. You know Python. If you do not, try Dive into Python
2. You know Django. You work with Django 2.0+. If you do not please read Django Docs

1.1.2 Getting the docs

The documents are generated using Sphinx. The code is available on Github. The generated Html is available at Django Design Patterns

To generate the docs, do *make html* within the top level folder of repo.

1.1.3 License

This work is dual licensed under Creative Commons Attribution-Share Alike 3.0 Unported License and GNU Free Documentation License

1.2 Urls

1.2.1 Projects and apps

There should be one *urls.py* at the project level, and one *urls.py* at each app level. The project level *urls.py* should include each of the *urls.py* under a prefix.:

```
#project urls.py

urlpatterns = patterns(
    '',
    (r'^', include('mainpages.urls')),
    (r'^admin/', include(admin.site.urls)),
    (r'^captcha/', include('yacaptcha.urls')),
    .....
)

#app urls.py
urlpatterns = patterns(
    'app.views',
    url(r'^$', 'index'),
    url(r'^what/$', 'what_view')
    .....
)
```

1.2.2 Naming urls

Urlpatterns should be named.[1] This is done as:

```
url(r'^$', 'index', name='main_index'),
```

This enables calling *{% url urlpatternname %}* much easier.

The pattern name should be of the form *appname_viewname*. If the same view is used in multiple urlpatterns, the name should be of form *appname_viewname_use*, as in *search_advanced_product* and *search_advanced_content*.:

```
#urls.py for app search
urlpatterns = patterns(
'search.views'
url(r'^advanced_product_search/$', 'advanced', name='search_advanced_product'),
url(r'^advanced_content_search/$', 'advanced', name='search_advanced_content'),
...
)
```

Here the same view *advanced* is used at two different urls and has two different names.

1.2.3 References

1.3 Models

1.3.1 Multiple managers

A Model class can have multiple managers, depending upon your needs. Suppose you do not want to display any object on your site which is unapproved(is_approved = False in your Model).:

```
class ModelClassApprovedOnlyManager(models.Manager):
    def get_query_set(*args, **kwargs):
        return super(ModelClassApprovedOnlyManager, self).get_query_set(*args,␣
↪**kwargs).filter(is_approved = True)
```

[1] http://github.com/agiliq/django-blogango/blob/9525dfa621ca54219eed0c0e9c1624de89948045/blogango/urls.py#L23

```
class ModelClass(models.Model):
    ...
    is_approved = models.BooleanField(default = False)

    objects = models.Manager()
    approved_objects = ModelClassApprovedOnlyManager()
```

If you use multiple managers, the first manager should be the default manager. This is as the first manager is accessible as *ModelClass._default_manager*, which is used by admin to get all objects.

1.3.2 Custom Manager Methods

Imagine you have a query like this:

```
Event.objects.filter(is_published=True).filter(start_date__gte=datetime.datetime.
↪now()).order_by('start_date')
```

you probably will need to filter by status and created date again, to avoid duplicating code you could add custom methods to your default manager:

```
class EventQuerySet(models.query.QuerySet):
    def published(self):
        return self.filter(is_published=True)

    def upcoming(self):
        return self.filter(start_date__gte=datetime.datetime.now())

class EventManager(models.Manager):
    def get_query_set(self):
        return EventQuerySet(self.model, using=self._db) # note the `using` parameter,
↪ new in 1.2

    def published(self):
        return self.get_query_set().published()

    def upcoming(self):
        return self.get_query_set().upcoming()

class Event(models.Model):
    is_published = models.BooleanField(default=False)
    start_date = models.DateTimeField()
    ...

    objects = EventManager()    # override the default manager
```

This way you keep your logic in your model. Why do you need a custom QuerySet? To be able to chain method calls. Now that query could be:

```
Event.objects.published().upcoming().order_by('start_date')
```

1.3.3 Hierarchical Relationships

You may want to model hierarchical relationships. The simplest way to do this is:

```
class ModelClass(models.Model):
    ...
    parent = models.ForeignKey('ModelClass')
```

This is called adjacency list model, and is very inefficient for large trees. If your trees are very shallow you can use this. Otherwise you want to use a more efficient but complex modeling called MPTT. Fortunately, you can just use django-mptt.

1.3.4 Singleton classes

Sometimes you want to make sure that only one Object of a Model can be created.

1.3.5 Logging

To make sure, when an object is create/edited/deleted, there is a log.

1.3.6 Audit Trail and rollback

When an object is modified or deleted, to be able to go back to the previous version.

1.3.7 Define an __unicode___

Until you define an *__unicode__* for your ModelClass, in Admin and at various other places you will get an *<ModelClass object>* where the object needs to be displayed. Define a meaningful *__unicode__* for you ModelClass, to get meaningful display. Once you define *__unicode__*, you do not need to define *__str__*.

1.3.8 Define a get_absolute_url()

get_absolute_url is used at various places by Django. (In Admin for "view on site" option, and in feeds framework).

1.3.9 Use reverse() for calculating get_absolute_url

You want only one canonical representation of your urls. This should be in urls.py

The *permalink* decorator is no longer recommended for use.

If you write a class like:

```
class Customer(models.Model)
    ...

    def get_absolute_url(self):
        return /customer/%s/ % self.slug
```

You have this representation at two places. You instead want to do:

```
class Customer(models.Model)
    ...

    def get_absolute_url(self):
        return reverse('customers.detail', args=[self.slug])
```

1.3.10 AuditFields

You want to keep track of when an object was created and updated. Create two DateTimeFields with *auto_now* and *auto_now_add*.:

```
class ItemSold(models.Model):
    name = models.CharField(max_length = 100)
    value = models.PositiveIntegerField()
    ...
    #Audit field
    created_on = models.DateTimeField(auto_now_add = True)
    updated_on = models.DateTimeField(auto_now = True)
```

Now you want, created_by and updated_by. This is possible using the threadlocals technique, but since we do not want to do that, we will need to pass user to the methods.:

```
class ItemSoldManager(models.Manager):
    def create_item_sold(self, user, ...):


class ItemSold(models.Model):
    name = models.CharField(max_length = 100)
    value = models.PositiveIntegerField()
    ...
    #Audit field
    created_on = models.DateTimeField(auto_now_add = True)
    updated_on = models.DateTimeField(auto_now = True)
    created_by = models.ForeignKey(User, ...)
    updated_by = models.ForeignKey(User, ...)

    def set_name(self, user, value):
        self.created_by = user
        self.name = value
        self.save()

    ...

objects = ItemSoldManager()
```

1.3.11 Working with denormalised fields

Working with child tables.

You want to keep track of number of employees of a department.:

```
class Department(models.Model):
    name = models.CharField(max_length = 100)
    employee_count = models.PositiveIntegerField(default = 0)


class Employee(models.Model):
    department = models.ForeignKey(Department)
```

One way to do so would be to override, *save* and *delete*.:

```
class Employee(models.Model):
    ...

    def save(self, *args, **kwargs):
        if not self.id:
            #this is a create, not an update
            self.department.employee_count += 1
            self.department.save()
        super(Employee, self).save(*args, **kwargs)

    def delete(self):
        self.department.employee_count -= 1
        self.department.save()
        super(Employee, self).delete()
```

Other option would be to attach listeners for *post_save* and *post_delete*.:

```
from django.db.models import signals

def increment_employee_count(sender, instance, created, raw, **kwargs):
    if created:
        instance.department.employee_count += 1
        instance.department.save()

def decrement_employee_count(sender, instance, **kwargs):
    instance.department.employee_count -= 1
    instance.department.save()

signals.post_save.connect(increment_employee_count, sender=Employee)
signals.post_delete.connect(decrement_employee_count, sender=Employee)
```

1.3.12 Abstract custom queries in Manager methods.

If you have some complex Sql query, not easily representable via Django ORM, you can write custom Sql. These should be abstracted as Manager methods.

1.4 Views

1.4.1 Generic views

Use generic views where possible.

1.4.2 Generic views are just functions

This means you can use them instead of calling say, *render_to_response*. For example, suppose you want to show a list of objects, so you would like to use *django.views.generic.object_list*. However, you also want to allow comments to be posted on these objects, which this generic view does not allow.[1]

[1] http://github.com/mightylemon/mightylemon/blob/ff916fec3099d0edab5ba7b07f4cf838ba6fec7b/apps/events/views.py

```
def object_list_comment(request):
    if request.method == 'POST':
        form = CommentForm(request.POST)
        if form.is_valid():
            obj = form.save()
            ...
            #redirect
    #Handle get or invalid form Post
    queryset = ModelClass.object.filter(...)
    payload = {'form':form}
    return object_list(request, queryset, extra_context = payload)
```

1.4.3 Handle GET and POST in same view function

This keeps things grouped logically together.[2] Eg.:

```
def foo(request):
    form = FormClass()
    if request.method == 'POST':
        #Handle POST and form saving etc.
        #Redirect etc
    #Any more GET handling
    payload = {'form': form, ...}
    return render_to_response(...)
```

1.4.4 Querysets are chainable and lazy

This means that your view can keep on creating querysets and they would be evaluated only when used. Suppose you have an advanced search view which can take multiple criteria all of which are optional.:

```
def advanced_search(request, criteria1=None, criteria2=None, criteria3=None):
    queryset = ModelClass.objects.all()
    if criteria1:
        queryset = queryset.filter(critera1=critera1)
    if criteria2:
        queryset = queryset.filter(critera2=critera2)
    if criteria3:
        queryset = queryset.filter(critera3=critera3)
    return objects_list(request, queryset=queryset)
```

1.4.5 References

1.5 Forms

1.5.1 Prefer ModelForm to Form

ModelForm already know the correct UI widgets for your underlying Models. In most of the cases ModelForm would suffice instead of Forms.

Some common scenarios

[2] http://github.com/agiliq/django-blogango/blob/9525dfa621ca54219eed0c0e9c1624de89948045/blogango/views.py#L65

Hiding some fields from ModelForm which are needed for a DB save.

Eg, you want to create a profile for the logged in user.:

```
#in models.py
class Profile(models.Model):
    user = models.OneToOneField(User)
    company = models.CharField(max_length=50)

#in forms.py
class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['company',]

#In views.py:
form = ProfileForm(request.POST)
profile = form.save(commit = False)
profile.user = request.user
profile.save()
```

Or:

```
class ProfileForm(forms.ModelForm):

    class Meta:
        model = Profile
        fields =['company',]

    def __init__(self, user, *args, **kwargs)
        self.user = user
        super(ProfileForm, self).__init__(*args, **kwargs)

    def save(self, *args, **kwargs):
        self.instance.user = self.user
        super(ProfileForm, self).save(*args, **kwargs)
```

Customizing widgets in ModelForm fields

Sometimes you just need to override the widget of a field that's already on your ModelForm. Instead of duplicating the field definition (with *help_text*, *required*, *max_length*, etc). You can do this:

```
from django.contrib.admin.widgets import AdminFileWidget

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['picture', 'company']

    def __init__(self, *args, **kwargs):
        super(ProfileForm, self).__init__(*args, **kwargs)
        # note that self.fields is available just after calling super's __init__
        self.fields['picture'].widget = AdminFileWidget()
```

Saving multiple Objects in one form

As:

```
class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['company',]

class UserForm(forms.ModelForm):
    class Meta:
        model = User
        fields = [...]

#in views.py
userform = UserForm(request.POST)
profileform =  ProfileForm(request.POST)
if userform.is_valid() and profileform.is_valid():
    #Only if both are valid together
    user = userform.save()
    profile = profileform.save(commit = False)
    profile.user = user
    profile.save()

{# In templates #}
<form ...>
{{ userform }}
{{ profileform }}
<input type="submit" />
</form>
```

1.5.2 Forms should know how to save themselves.

If your forms is a *forms.ModelForm*, it already knows how to save its data. If you write a forms.Form, it should have a *.save()*. This keeps things symmetrical with *ModelForms*, and allows you to do:

```
#in views.py
def view_func(request):
    if request.method == 'POST':
        form   = FormClass(request.POST)
        if form.is_valid():
            obj = form.save()
            ...
        ...
```

Instead of:

```
if form.is_valid():
    #handle the saving in DB inside of views.
```

The *.save()* should return a Model Object

1.5.3 The form should know what to do with it's data

If you're building a contact form, or something like this, the goal of your form is to send an email. So this logic should stay in the form:

```
class ContactForm(forms.Form):
    subject = forms.CharField(...)
    message = forms.TextField(...)
    email = forms.EmailField(...)
    ...

    def save(self):
        mail_admins(self.cleaned_data['subject'], self.cleaned_data['message'])
```

I've used *save()*, and not *send()*, even when i'm not really saving anything. This is just a convention, people prefer to use *save()* to keep the same interface to ModelForms. But it doesn't really matter, call it whatever you want.

1.6 Templates

1.6.1 Projects and apps.

There should be one base.html at the project level, and one *base.html* at each of the app levels. The app level *base.html* should extend the project level *base.html*.:

```
{# Eg Project base.html #}

<html>
<head>
<title>{% block title %}My Super project{% endblock %}</title>
...

{# app base.html #}

{% extends 'base.html' %}

{% block title %}{{ block.super }} - My duper app {% endblock %}
...


{# login.html #}

{% extends 'auth/base.html' %}
{% block title %}{{ block.super }} - Login {% endblock %}
...
```

1.6.2 Location of templates

The templates for an app should be available as *appname/template.html*. So the templates should be physically located at either

1. project/templates/app/template.html
2. project/app/templates/app/template.html

This allows two apps to have the same templates names.

1.6.3 Handling iterables which maybe empty

In your view you do:

```
posts = BlogPosts.objects.all()
...
payload = {'posts': posts}
return render_to_response('blog/posts.html', payload, ...)
```

Now *posts* may be empty, so in template we do,:

```
<ul>
{% for post in posts %}
    <li>{{ post.title }}</li>
    {% empty %}
    <li>Sorry, no posts yet!</li>
{% endfor %}
<ul>
```

Please, note about *empty* clause using. If *posts* is empty or could not be found, the *empty* clause will be displayed.

1.7 Workflow

1.7.1 Use a source control system

Either of Git and Mercurial are good choices.

1.7.2 Create a requirements.txt

Your project should have a requirements.txt file. A *pip install -r requirements.txt* should get all the third party apps which are not part of your source control system.

1.7.3 pin your requirements.txt

For predictable and deterministic

1.7.4 Use virtualenv and pip (sandbox)

Your various projects might require different versions of third party libraries. Use virtualenv to keep separate environments and use pip to manage dependencies.

If you use virtualenv a long time, you can try *virtualenvwrapper <http://virtualenvwrapper.readthedocs.org/en/latest/>*

1.7.5 Use pep8.py to check compliance with Python coding guidelines.

Your code would be using conforming to pep8, which are the standard coding guidelines. Pep8.py can check your code for deviations.

1.7.6 Use one code analyzer for static analysis.

- Pyflakes can find out some common mistakes.
- Pylint code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells.

Run it after each deploy.

1.7.7 Use a bug tracking tool.

Use one of

- Trello
- Jira
- Asana
- Basecamp
- Trac
- Assembla
- Unfuddle

1.7.8 Use south for schema migration

Django doesn't come with a built in tool for schema migration. South is the best tool for managing schema migrations and is well supported.

1.7.9 Create various entries in your /etc/hosts mapped to localhost

While development you probably want multiple users logged in to the site simultaneously. For example, while developing, I have one user logged in the admin, one normal user using the site. If both try to access the site from localhost, one will be logged out when other logs in.

If you have multiple entries mapped to localhost in /etc/hosts, you can use multiple users simultaneously logged in.

1.7.10 Do not commit the generated files

Django does not have a lot of auto generated files. However as you work with other Django apps, you may come across auto generated files. These should not be checked in the the Django repository. For example, for this book, we commit the source files and folder, but not the auto-generated build folders.

1.8 Misc

1.8.1 settings.py and localsettings.py

The settings for your project which are a machine specific should be refactored out of settings.py into localsettings.py. In your settings.py, you should do:

```
try:
    from localsettings import *
except ImportError:
    print 'localsettings could not be imported'
    pass #Or raise
```

This should be at the end of settings.py, so that localsetting.py override settings in settings.py

This file should not be checked in your repository.

1.8.2 Use relative path in settings.py

Instead of writing:

```
TEMPLATE_DIRS = '/home/user/project/templates'
```

Do:

```
#settings.py
import os

CURRENT_DIR = os.path.dirname(__file__)
TEMPLATE_DIRS = os.path.join(CURRENT_DIR, 'template')
```

1.8.3 Apps should provide default values for settings they are trying to read.

As far as possible, apps should have defaults for settings they are trying to read. Instead of:

```
DEFAULT_SORT_UP = settings.DEFAULT_SORT_UP
```

Use:

```
DEFAULT_SORT_UP = getattr(settings, 'DEFAULT_SORT_UP' , '&uarr;')
```

1.8.4 Use templatetag when the output does not depend on the request

In the sidebar, you want to show the 5 latest comments. You do not need the request to output this. Make it a templatetag.

1.8.5 Import as if your apps are on your project path

Instead of doing *from project.app.models import ModelClass* do *from app.models import ModelClass*. This makes you apps reusable as they are not tied to a project.

1.8.6 Naming things

Model class names should be singular, not plural.:

```
class Post(models.Model):
    ...
```

and not:

```
class Posts(models.Model):
    ...
```

Foreign key should use the name of the referenced class.:

```
class Post(models.Model):
    user = models.ForeignKey(User)
```

Querysets should be plural, instances should be singular.:

```
posts = Post.objects.all()
posts = Post.objects.filter(...)

post = Post.object.get(pk = 5)
post = Post.object.latest()
```

1.8.7 Using pdb remotely

Sometimes you will hit bugs which show up on server but not on your local system. To handle these, you need to debug on the server. Doing *manage.py runserver* only allows local connections. To allow remote connections, use:

```
python manage.py runserver 0.0.0.0:8000
```

So that your *pdb.set_trace()* which are on remote servers are hit when you access them from your local system.

1.8.8 Do not use primary keys in urls

If you use PK in urls you are giving away sensitive information, for example, the number of entries in your table. It also makes it trivial to guess other urls.

Use slugs in urls. This has the advantage of being both user and SEO friendly.

If slugs do not make sense, instead use a CRC algorithm.:

```
class Customer(models.Model):
    name = models.CharField(max_length = 100)

    def get_absolute_url(self):
        import zlib
        #Use permalink in real case
        return '/customer/%s/' % zlib.crc32(self.pk)
```

1.8.9 Code defensively in middleware and context processors.

Your middleware and context processors are going to be run for **all** requests. Have you handled all cases?

def process_request(request):

if user.is_authenticated(): profile = request.user.get_profile() # Hah, I create profiles during # registration so this is safe. . . .

Or it is? What about users created via *manage.py createsuperuser*? With the above middleware, the default user can not access even the admin site.

Hence handle all scenarios in middleware and context processors. This is one place where *try: .. except: ..* (bare except) blocks are acceptable. You do not want one middleware bringing down the entire site.

1.8.10 Move long running tasks to a message queue.

If you have long running requests they should be handled in a message queue, and not in the request thread. For example, using a lot of API calls, will make your pages crawl. Instead move the API processing to a message queue such as celery.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search