Django gotchas Documentation

Release 0.1

Usware Technologies and Contributers

Feb 07, 2018

Contents

Contents:

CHAPTER 1

Urls

1.1 Url patterns catching unexpected patterns

Consider these patterns,:

```
...
url('edit/', edit_view, )
url('edit/bulk/', edit_bulk_view)
...
```

So when you try to go to 'edit/bulk/' you would get the edit_view, as 'edit/' pattern matched 'edit/bulk/'.

Solution: Always use the smallest pattern which matches.:

```
...
url('^edit/$', edit_view, )
url('^edit/bulk/$', edit_bulk_view)
...
```

Tip: In most of the cases you should be starting you patterns with ^ and ending it with *$*.

1.2 Urlpatterns failing to catch expected patterns

You want to edit posts based on slugs, so you do:

```
url('^edit/(?P<entry_slug>\w-+)/$', edit_view, )
```

However this would not catch slugs with - in them.

Solution: Think of the each value the slug can take, if you allow charactors and dashes, the next line would catch all of them.

url('^edit/(?P<entry_slug>[w-]+)/$', edit_view,)

CHAPTER 2

Views

2.1 Not returning a HttpResponse

This is very common, but Django complains loudly. So very obvious and easy to fix.

Solution: Always return HttpResponse via any of the mechanisms.

2.2 Not including RequestContext

Sometime you mean this:

```
return render_to_response('templates/app/template.html', payload,␣
↪RequestContext(request))
```

But write:

```
return render_to_response('templates/app/template.html', payload,)
```

So any of your data from context processors would not be available. Eg, *media_url*.

Solution:

If you must use this, always be sure why you are doing it this way.

return render_to_response('templates/app/template.html', payload,)

2.3 Filtering on id

A common view function:

def edit_post(request, id): Post.objects.get(id = id)

id is a builtin in Python and overriding that is probably a bad idea. But this happens commonly as Django names the table's PK *id*.

Solution:

Name the variables descriptively or always use *pk*

def edit_post(request, post_id): Post.objects.get(pk = post_id)

2.4 Passing *locals()* to templates

It is sometimes very tempting to pass *locals()* to the templates, especially if you have used a lot of variables, and most of them are used in templates.

This is almost always a bad idea as,

1. it makes it harder to debug templates, as you do not know what variables are available in the template at one glance.
2. Make it harder to refactor views, as you can not remove unneeded variables from the view after refactoring, as you dont know whether they are being used in templates.

Solution: Don't use locals(), always explicitly create a dictionary to pass to the templates.

CHAPTER 3

Forms

3.1 request.POST binding to wrong parameter in forms

This is a common idiom in a view.:

```
def view_func(request):
    form = FormClass()
    if request.method == 'POST':
        form=FormClass(request.POST)#Gotcha
        ....
    payload = {'form':form}
    return render_to_response(...)
```

Now suppose we edited our form class to:

```
class FormClass(forms.Form):
    def __init__(self, user = None, *args, **kwargs):
        ...
```

request.POST will bind to user, when we meant to be the POST data.

Solution:

Rewrite the gotcha line as

form=FormClass(data=request.POST)

3.2 Using form.cleaned_data before calling form.is_valid

Even if you are sure your form passes validation, you still need to call *is_valid*. *form.cleaned_data* is calculated after *form.is_valid* is called, which you probably are using in your *form.save()*.

3.3 Overriding form.save in ModelForms:

Similar to *models.save*, forms.save are often (wrongly) overriden as *forms.save*:

```
def forms.save(self):
    ...
    super(FormCLass, self).save()
```

This won't work in many cases, for example *form.save(commit=False)*

Solution:

Always save as:

```
def forms.save(self, *args, **kwargs):
    ...
    super(FormClass, self).save(*args, **kwargs)
```

3.4 Not calling superclass __init__

You might want to change the widget of a particular form field and this can be done by overriding Form's __init__.:

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

class FormClass(forms.ModelForm):

    class Meta:
        model = Profile

    def __init__(self, *args, **kwargs):
        self.fields['picture'].widget = AdminFilewidget() #Gotcha
```

self.fields is populated after __init__ of ModelForm has executed. So, make sure to call superclass __init__ before accessing self.fields.:

```
def __init__(self, *args, **kwargs):
    super(FormClass, self).__init__(*args, **kwargs)
    self.fields['picture'].widget = AdminFilewidget()
```

CHAPTER 4

Models

4.1 Not adding a max_length to models.CharField

If you do not add a max_length to CharFields your models will not validate.

Solution:

Add a max_length. If you want a field with unlimited length, use a TextField.

4.2 Adding null=True on subclasses of CharField

Read about null to find out why null=True should not be used for CharField. Often people keep this in mind but still use null=True on fields like EmailField, SlugField etc. Note that these fields are subclasses of CharField and hence not making them as null applies to these fields too.

4.3 Not handling for ModelClass.DoesNotExist and ModelClass.MultipleObjectsReturned

If you are doing a *ModelClass.objects.get(this = that) unless you are doing it for a unique key, this has a potential to raise 'ModelClass.MultipleObjectsReturned* and *ModelClass.DoesNotExist*. This needs to be handled.

Solution: If one of the query criteria is on an unique field, put it in a *try: .. except ModelClass.DoesNotExits:...* If there criteria has no unique field, handle both *DoesNotExist* and *MultipleObjectsReturned*. In a view function use django.shortcuts.get_object_or_404(ModelClass, criteria = criteria)

4.4 Overriding .save()

Most of the times you would be using *model_obj.save()* without any parameters, so you may override *.save* as:

```
def save(self):
    ...
    super(ModelClass, self).save()
```

This would work until, *.save* is called with a parameter. For example, when you use *get_or_create*, *.save* is called with as *.save(force_insert = True)*, which will fail.

Solution: Override save as:

```
def save(self, *args, **kwargs):
    ...
    super(ModelClass, self).save(*args, **kwargs)
```

4.5 Trying qs.get('foo')

Django *.get* and '.filter' takes only keywords argument. So *ModelClass.objects.get(foo)* gives an error. Similarly creating objects as *ModelClass(foo=foo)* takes only keyword arguments.

Solution: Use keyword arguments.

4.6 Trying to use an stale object

What is wrong with code?:

```
def foo(req):
  obj =ModelClass.objects.get(pk = 5)
  magic(obj.pk)
  payload{'obj':obj}
  return render_to_response(template, payload, ..)
```

Django does not have object identity, hence any change made to the database within obj within *magic* won't be available in the function.

Solution: There is no easy solution to it, but you can keep youself from being bitten by this behaviour. Pull objects close to their use. Keep track of when an used object is updated and pull them again from the database if updated.

CHAPTER 5

Templates

5.1 Relative imports and extends in templates

Django templates can not import via relative imports. This is therefore wrong.:

```
{% extends '../base.html' %}

{% include '../../breadcrumbs_frag.html' %}
```

Solution: Always import as specified from the application/template directory, or the *TEMPLATE_DIRS* specified in *settings.py*

5.2 Not wrapping {% for %} in {% if %}

A common pattern is:

```
<ul><!-- or any other element, eg table -->
{% for comments in post.comment_set.all %}
<li>
{{ comment }}
</li>
{% endfor %}
</ul>
```

This doesnot handle an empty queryset, and will create a empty *
*, which might be visible and break your design depending on your css.

Solution:

Wrap it in *{% if %}*:

```
{% if post.comment_set.all %}
<ul><!-- or any other element, eg table -->
```

```
{% for comments in post.comment_set.all %}
<li>
{{ comment }}
</li>
{% endfor %}
</ul>
{# optional else #}
{% else %}
<div>No comment</div>
{% endif %}
```

Or (Django 1.1 only)

> <!– or any other element, eg table –> {% for comments in post.comment_set.all %} {{ comment }} {% empty %} No comment {% endfor %}

5.3 Template Silencing:

Any empty variable is rendered empty in the templates. But errors are rendered empty as well. Django templates silence many variable errors and render nothing at that position.

Solution:

During development set the template string to a convenient value when the template string is invalid.

This setting helps.:

TEMPLATE_STRING_IF_INVALID = '{{ %s }}'

Tip: This should be one of the settings in your *localsettings*, or one of conditions within *if DEBUG:*

Warning: The breaks the admin design.

CHAPTER 6

Contrib

6.1 Manually Authenticating a User:

django.contrib.auth defines a method named *login*, with the following signature (in the file *__init__.py*):

```
def login(request, user):
        """
        Persist a user id and a backend in the request. This way a user doesn't
        have to reauthenticate on every request.
    """
```

login takes request and the user object. However, passing a standard user object and a request, doesn't log the user in, when logging in manually, because *authenticate()* sets an extra argument on the user object, that is used for logging in.

Solution:

When loggining in the user manually, first use the authenticate method.:

```
user_auth = authenticate(username=new_user.username, password=pw1)
login(request, user_auth)
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search