Writing Django Views: Function Based Views, Class Based Views, and Class Based Generic Views

Written by Dan Sackett on September 19, 2014

One of things things I like most about Django is that you can really customize it to your workflow.

In essence, a Django Project is nothing more than a group of settings. The framework does the heavy lifting behind the scenes once configured. With that in mind, one of the things that's widely debated in the community is what kind of views should you use.

For those that don't know, a view is your typical "controller" in an MVC framework. It handles requests your users make and can be used to process data. Keeping views lean is the key but keeping them understandable is also key. We have three ways to write these views and each is a little different. Those three methods are:

- Function Based Views (FBVs)
- Class Based Views (CBVs)
- Class Based Generic Views (CBGVs)

Now before you ask what the difference between CBVs and CBGVs, I recommend you check out this video from DjangoCon 2014: Class-based Views: Past, Present and Future. It's a great look at where class based views started and where they are now. In that you'll learn that:

- Class Based Views inherit the generic View class `django.views.generic.base.View`
- Class Based Generic Views inherit from a child of the generic View
- Class Based Generic Views are based on components and mixins combined to create a common patten Django users may encounter

I highly recommend watching that video to see more on that.

For our sake, let's take a look at the three different view styles. My goal isn't to tell you what to use, but rather to show you the different styles so you can decide what you like best. In Django, each view type has a best practice so saying that one of these types is a best practice would be false.

Before I show the views, let's get the other components specified before we start. Note that these don't change based on the views. We can reuse the same templates and forms no matter what the view looks like.

I have created an app called Demo in my project. In this Demo application the files are:

models.py

```
from django.db import models

class Demo(models.Model):
    field = models.CharField(max_length=100)
```

forms.py

```
from django import forms
from .models import Demo

class DemoForm(forms.ModelForm):
    class Meta:
        model = Demo
        fields = ('field',)
```

my_template.html (**Note:** The URL should be changed for each view based on the name you assign it in your URLconf)

```
<h1>Demo Form</h1>

<form role="form" method="post" action="{% url 'function' %}">
    {% csrf_token %}
    {{ form }}
    <button type="submit">Save</button>
</form>
```

With those setup, let's learn about the view we are going to make. We'll implement a create object form. The pseudocode for this will be:

```
initialize form
    if request is POST
        if form is valid
            save form
            redirect back to form (should usually go somewhere else)
        else
            show an error message
    else
        render the view for GET
```

This is the usual flow for creating an object. We want to check that there is post data and if so we want to check if the data is valid. If it is valid, save the data and redirect to the form. If it's not valid, show an error message. If there is no post data, treat the view as a get request and serve the blank form.

Function Based Views

Function based views are the most common views implemented. They're explicit and honestly follow the pseudocode word for word in execution. Let's see a function based view.

views.py

```
from django.http import HttpResponse
from django.shortcuts import render, redirect

from demo.forms import DemoForm
from demo.models import Demo

def function_based_view(request):
    """
    Function based views take a request and are very explicit in the actions.
    They have more code in some regards, but they go step by step in execution.
    """
    form = DemoForm(request.POST or None)
```

```
    if request.method == 'POST':
        if form.is_valid():
            form.save()
            return redirect('function')
        else:
            return HttpResponse('Error!')

    context = {'form': form}

    return render(request, 'my_template.html', context)
```

urls.py

```
from django.conf.urls import patterns, include, url

urlpatterns = patterns('',
    url(r'^function/$', 'demo.views.function_based_view', name='function'),
)
```

As I said, very straightforward and highly legible. Line by line this reads well and covers all of the cases that we wanted to. Things to notice about this:

- The view is a function that takes a request variable. Since this is a function based view, we should expect that this would be a function.
- We have GET and POST sections combined into one function
- In our `urls.py` we reference the import path to this view as a string

I've used function based views for the past three years primarily because my first job did them this way. Some of the perks of this approach are:

- The logic is all in one function and easy to trace
- We can instantiate the form once for GET and POST

And some of the downsides of this approach:

- Making if calls sometimes can look sloppy in my mind. If this, if that can build up and deep nesting can occur.
- This is such a common pattern that you'll find yourself repeating this boilerplate over and over in your views.

In the end, I think the perks outweigh the downsides and function based views are actually a great way to go. They give you full control over logic.

Class Based Views

Class based views are not very different than function based views as you'll see. You still have full control over the logic only CBVs allow us to separate HTTP request types to give us a little more organization.

views.py

```
from django.http import HttpResponse
from django.shortcuts import render, redirect
from django.views.generic.base import View
```

```
from demo.forms import DemoForm
from demo.models import Demo

class ClassBasedView(View):
    """
    Class Based Views allow you to explicitly state your HTTP methods such as
    GET and POST. If a method is not specified then it a request of that type
    will be served a 405 error.
    """
    def get(self, request):
        form = DemoForm(None)
        context = {'form': form}
        return render(request, 'my_template.html', context)

    def post(self, request):
        form = DemoForm(request.POST)

        if form.is_valid():
            form.save()
            return redirect('cbv')

        return HttpResponse('Error!')
```

urls.py

```
from django.conf.urls import patterns, include, url

from demo.views import ClassBasedView

urlpatterns = patterns('',
    url(r'^cbv/$', ClassBasedView.as_view(), name='cbv'),
)
```

Some things to note:

- The view is defined as a class that inherits from the generic base view that Django has built in.
- We break the class into two functions - one for GET and one for POST.
- Our urls.py imports the actual view class and then you must use the `as_view()` method to render it correctly.

Some of the perks of this setup:

- We get the full control we like from function based views
- We have a separation of GET and POST requests so they don't run into each other

And some of the downsides:

- In each GET and POST we have to redefine the form. Now you can define the form on the class as a property as well, but that doesn't seem like common convention among Django projects that I've seen.
- We have to invoke the `as_view()` method on the view in our urls.py file. Minor gripe but it's something that can easily be forgotten if you aren't paying attention.

I'll admit that in the three years that I've been doing Django, I didn't take the time to learn CBVs until last week when a colleague proposed we try them in a project. I've been reading a lot about them (expect some blog posts about things I've learned) and when I look at this approach I actually

like it a lot. Maybe even moreso than FBVs.

One thing that you may not be aware of that I just learned is that if we were to only specify a POST function in our class and the user made a GET request then the application would return a 405 method not allowed error. In FBVs I typically import `django.views.decorators.http.require_POST` and use that as a decorator to specify that the view cannot serve GET requests. Seeing that it's one less thing I have to import and implement, I like CBVs even more.

Class Based Generic Views

Class based generic views are going to be the outlier in this set. They make your code more sparse and they let you trust that Django will handle things for us.

views.py

```
from django.http import HttpResponse
from django.shortcuts import render, redirect
from django.views.generic import FormView

from demo.forms import DemoForm
from demo.models import Demo

class ClassBasedGenericView(FormView):
    """
    Class Based Generic Views allow you to use a typical pattern and write
    minimal code as Django will handle the boilerplate for you behind the
    scenes.
    """
    template_name = 'my_template.html'
    form_class = DemoForm
    success_url = '/cbgv/'

    def form_valid(self, form):
        form.save()
        return super(ClassBasedGenericView, self).form_valid(self)

    def form_invalid(self, form):
        return HttpResponse('Error!')
```

urls.py

```
from django.conf.urls import patterns, include, url

from demo.views import ClassBasedGenericView

urlpatterns = patterns('',
    url(r'^cbgv/$', ClassBasedGenericView.as_view(), name='cbgv'),
)
```

Some things to note:

- We define our view as a Class like before but this time we are inheriting from the generic `FormView` class. More on this in a minute.
- We define our template_name, form_class, and success_url as class properties
- We define methods that we care about in the class. In this case, we care about form_valid

and form_invalid
- Our urls.py file uses the `as_view()` syntax as the CBVs did

Some of the perks of this approach:

- Less code and honestly, no logic in the view. We send the logic to Django.
- We avoid the common boilerplate pattern of forms in Django

And some of the downsides:

- We don't get to see any of the logic in the view.
- If you don't know how class based views work or just what methods FormView has, then you might have a hard time working with this view.

Looking at this, I like how slim it makes the view. No if statements, just functions that are run when a form is valid/invalid. It's clean and achieves exactly what we want at the expense of understandability for those that have never worked with CBGVs.

What's happening behind the scenes?

Well, a lot of classes in Django's source code are collaborating to create this view.

- FormView is basically a stub class that inherits from TemplateResponseMixin and BaseFormView
- BaseFormView is another stub class that inherits from FormMixin and ProcessFormView
- ProcessFormView inherits from generic View and it will handle rendering a form on GET and validating a form on POST
- FormMixin inherits from ContextMixin and gives us most of our available methods such as form_valid and form_invalid.
- ContextMixin inherits from object (topmost level) and passes the keyword arguments received by get_context_data as the template context.
- TemplateResponseMixin from the original FormView class inherits from object as well and gives us the ability to return a HTTP response object

Confusing?

Maybe a little. Django CBGVs are simply a series of classes that work together and combine functionality to satisfy a particular pattern. Those patterns are:

- Show a single object
- List objects
- Display and Process a form
- Allow creation, editing, and deleting model objects *Display date archives

A lot of these patterns can be tedious in function arrangements so if you want to lean on Django and type a little less code, CBGVs are for you.

Conclusion

In the end, it's really your call how you structure your views in Django. The best approach is to find

how you like to read your code. If you like to follow the logic line for line and want a single function then FBVs are for you. If you like the logic but want to keep your HTTP methods separate, CBVs are for you. If you want to save some time and stand on the shoulders of giants then CBGVs are for you.

What's your preferred method? I'd love to hear how you work with views!