Coderbook

Django's Class Based Views vs Function Based Views

01 Feb 2019

As a Web Framework, one of the main uses for Django is to serve HTTP Responses back to the client that is visiting the website. This is done through what Django calls "Views". A View is simply just a callable that returns a `HTTPResponse`.

For any beginner who starts reading through the Django Documentation, they will quickly be introduced to what we call Function-Based Views (FBV), this is just a simple function that returns a response.

For example

```
from django.http import HttpResponse

def index(request):
    if request.method == "GET":
        return HttpResponse("Hello World")
    elif request.method == "POST":
        return HttpResponse("Hello World")
```

As you can see, it's extremely simple and you can generate a view that will return a proper response in just a handful of lines of code. This was the first way of creating views in Django, and since it was the first, and also the simplest, it's also probably the reason why the documentation choose to introduce beginners to it right from the get-go.

The only problem with function based views is that you don't get much for free with them. Each time you want to process a POST request from a form, render a template, query the database for objects to display, you have to do it manually over and over again. It's not very DRY.

Because of this, the Django contributors decided to introduce something that they called Class-Based Views. As you can infer from the name, instead of defining views as a function, we

can instead define it as a class.

```
from django.http import HttpResponse
from django.views.generic import View

class IndexView(View):
    def get(self, request, *args, **kwargs):
        return HttpResonse("Hello World")

    def post(self, request, *args, **kwargs):
        return HttpResonse("Hello World")
```

By looking at the examples you can see that they both achieve the same thing, they take requests and they route the action based on what kind of HTTP Method that is being used.

So why would we want to use Class-Based Views? Why would they be superior, or inferior to Function-Based Views? Let's explore that.

Function-Based Views in Django

As we could see in the example displayed in the previous section, a function based view can be simple and easy to understand. At least in the context of small examples.

But what happens as our view gets more complex and require more logic? It's rarely the case

that we just want to have a single line of code where we return an `HTTPResponse`. Instead, other common behaviors that we might want to add would be:

- Use templates and render them as the response.
- Automatically add additional data to template context.
- Display a single, detailed view of some model.
- Display a list of items of a model.
- Process Forms and do some action when they are submitted.

Most of the item on our list are things that we want to be done on almost every single view. This means that perhaps the impression that we get from Function-Based Views from the tiny and small code examples are a bit skewed, perhaps they aren't that "simple" when it comes to the real world.

A more realistic example of a function based view might be something like the code example below.

```
from django.http import Http404
from django.shortcuts import render, get_object_or_404
from polls.models import Poll
from .forms import PollForm


def detail(request, poll_id):
    context = get_template_context(poll_id)

    if request.method == "POST":
        form = PollForm(request.POST)
        if form.is_valid():
            Poll.objects.create(**form.cleaned_data)
            return HttpResponseRedirect('/thanks/')
    else:
        form = PollForm()

    context["form"] = form
    return render(request, 'polls/detail.html', context)

def get_template_context(poll_id):
    poll = get_object_or_404(Poll, pk=poll_id)
    return {
```

```
        "poll": poll,
        "meta": {
            "title": poll.name,
            "description": " ".join(poll.description.split(" ")[:25]),poll.descriptior
        },
    }
```

That example is just to process a form and to add some metadata such as a title and description to our template context. You can imagine that these tasks will be repeated over and over and over again for each View that you create within your application.

I'd like to summarize my pros and cons about Function-Based Views as the following:

Pros:

- Initially simple and clean code.
- Easy to learn, grasp and to teach new programmers.

Cons:

- The cases where they remain small, simple and clean are rare. They almost always will expand into a bit more complexity which suddenly removes all the elegance from them.
- No clear, predetermined pattern between developers. You might have looked at the code example above and thought to yourself "I'd never do that! I would make it in a different way!". In my mind that is not a good thing, the fact that each programmer can construct their view uniquely to their preferences make the code harder to read.
- Cannot automatically reuse functionality with inheritance. Even if you create utility functions for a lot of common tasks, you still have to call them in each and every view.

Class-Based Views in Django

I was not the first one to get frustrated by the obvious downsides that come with Function-Based Views. The Django core developer team also felt that these common tasks were repeated over and over again and they came up with a better(!) way - Class-Based Views.

They introduced multiple class-based views that all inherit from the base `View` class. They can all be found in the `django.views.generic` module and they include classes such as:

- View
- TemplateView
- FormView

- CreateView
- UpdateView
- DeleteView
- DetailView
- ListView

They all fill the need for some very common tasks. For example, the `TemplateView` allows you to render any template, or the `DetailView` allows you to specify a Model that you want to get an instance of, and then it automatically populates your template context with that object by reading URL kwargs such as `pk` or `slug`.

We could easily clean up our previous Function-Based View example with the following code:

```
from django.views.generic import CreateView
from .models import Poll
from .mixins import MetaMixin


class PollDetailView(MetaMixin, CreateView):
    template_name = "polls/poll.html"
    model = Poll
```

These 3 lines of code would allow us to automatically implement the following things:

- `MetaMixin` can be a custom mixin that automatically populates the meta attributes of your view if an object is present. For example, it could use the string representation of the object to make a Meta title automatically.
- `CreateView` creates a `ModelForm` instance and pass it to our template context. It also automatically adds code for the POST request to validate the form request and to create a new instance of `Poll`.
- `CreateView` knows that it should generate the response from a template. by default, it attempts to construct the template file path by itself based on the class name, but in this case, we explicitly set the file path with the `template_name` property.

Even though we've added some additional complexity by making a Class instead of a Function, this quickly gets balanced by the fact that we reduce our code down and make it incredibly readable, while also making sure that it is reusing as much code as possible.

The whole point of Class-Based Views is to allow us to be lazy as developers. We should be! Why should we recreate things that already exist? Why should we repeat our code over and

over again which both causes problems with its readability, but also makes it much more difficult to maintain?

Pros:

- Improved readability.
- Improved reusability of previously written code.
- Faster to write and iterate on.
- You only write the code that regards to real custom features. Not things that every other web application shares.

Cons:

- For new developers, an object-oriented design might be more difficult to grasp than functional approaches. Because of this, it might be more difficult to teach this approach to new developers.

Personal Preferences

After years of working with Django, I've come to have a solid opinion about how I create my own views in my Django projects. I'd go as far to say that I always go with the Class-Based View approach.

I can't even remember the last time that I felt that the functional approach was the best-suited option. It might be for the first initiation of our view, but I always know that any view will eventually need some kind of custom logic to it. No matter if it's a REST API, a simple Template or some kind of CRUD View where users can manage the instances of the application model.

As long as you have a good understanding of object-oriented programming, inheritance, classes, mixins etc. Just go with the Class-Based approach. I promise you that you won't regret it.

Related posts

How to make Singleton objects in Python 23 Apr 2020
How to make Lazy objects in Python 23 Apr 2020
Publish your documentation to GitHub Pages from Jenkins Pipeline 18 Nov 2019
A complete guide to CI/CD Pipelines with CircleCI, Docker and Terraform 14 Oct 2019
How to Write Unit Tests and Mock with Pandas 02 Aug 2019

ALSO ON **CODERBOOK**

TopTal Review - My First Year as a …

2 months ago • 6 comments

A little more than 1 year ago I decided to apply to become a freelancer with …

How to Unit Test Functions Without …

2 months ago • 2 comments

Unit Testing is one of the core concepts within programming that any …

Deep Dive into Python Mixins and Multiple …

2 months ago • 1 comment

In my opinion, Mixins is one of the most powerful, and useful features …

How with

2 mont

Djang permi allow

0 Comments Coderbook Disqus' Privacy Policy 1 Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

marcuslind90@gmail.com
marcuslind90