Introduction to class-based views

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

- Organization of code related to specific HTTP methods (`GET` , `POST` , etc.) can be addressed by separate methods instead of conditional branching.
- Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

The relationship and history of generic views, class-based views, and class-based generic views

In the beginning there was only the view function contract, Django passed your function an `HttpRequest` and expected back an `HttpResponse`. This was the extent of what Django provided.

Early on it was recognized that there were common idioms and patterns found in view development. Function-based generic views were introduced to abstract these patterns and ease view development for the common cases.

The problem with function-based generic views is that while they covered the simple cases well, there was no way to extend or customize them beyond some configuration options, limiting their usefulness in many real-world applications.

Class-based generic views were created with the same objective as function-based generic views, to make view development easier. However, the way the solution is implemented, through the use of mixins, provides a toolkit that results in class-based generic views being more extensible and flexible than their function-based counterparts.

If you have tried function based generic views in the past and found them lacking, you should not think of class-based generic views as a class-based equivalent, but rather as a fresh approach to solving the original problems that generic views were meant to solve.

The toolkit of base classes and mixins that Django uses to build class-based generic views are built for maximum flexibility, and as such have many hooks in the form of default method implementations and attributes that you are unlikely to be concerned with in the simplest use cases. For example, instead of limiting you to a class-based attribute for `form_class` , the implementation uses a `get_form` method, which calls a `get_form_class` method, which in its default implementation returns the `form_class` attribute of the class. This gives you several options for specifying what form to use, from an attribute, to a fully dynamic, callable hook. These options seem to add hollow complexity for simple situations, but without them, more advanced designs would be limited.

Using class-based views

At its core, a class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP `GET` in a view function would look something like:

```
from django.http import HttpResponse


def my_view(request):
    if request.method == "GET":
        # <view logic>
        return HttpResponse("result")
```

In a class-based view, this would become:

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


class MyView(View):
    def get(self, request):
        # <view logic>
        return HttpResponse("result")
```

Because Django's URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an `as_view()` class method which returns a function that can be called when a request arrives for a URL matching the associated pattern. The function creates an instance of the class, calls `setup()` to initialize its attributes, and then calls its `dispatch()` method. `dispatch` looks at the request to determine whether it is a `GET`, `POST`, etc, and relays the request to a matching method if one is defined, or raises `HttpResponseNotAllowed` if not:

```
# urls.py
from django.urls import path
from myapp.views import MyView

urlpatterns = [
    path("about/", MyView.as_view()),
]
```

It is worth noting that what your method returns is identical to what you return from a function-based view, namely some form of `HttpResponse`. This means that http shortcuts or `TemplateResponse` objects are valid to use inside a class-based view.

While a minimal class-based view does not require any class attributes to perform its job, class attributes are useful in many class-based designs, and there are two ways to configure or set class attributes.

The first is the standard Python way of subclassing and overriding attributes and methods in the subclass. So that if your parent class had an attribute `greeting` like this:

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


class GreetingView(View):
    greeting = "Good Day"

    def get(self, request):
        return HttpResponse(self.greeting)
```

You can override that in a subclass:

```
class MorningGreetingView(GreetingView):
    greeting = "Morning to ya"
```

Another option is to configure class attributes as keyword arguments to the `as_view()` call in the URLconf:

```
urlpatterns = [
    path("about/", GreetingView.as_view(greeting="G'day")),
]
```

Note: While your class is instantiated for each request dispatched to it, class attributes set through the `as_view()` entry point are configured only once at the time your URLs are imported.

Using mixins

Mixins are a form of multiple inheritance where behaviors and attributes of multiple parent classes can be combined.

For example, in the generic class-based views there is a mixin called `TemplateResponseMixin` whose primary purpose is to define the method `render_to_response()`. When combined with the behavior of the `View` base class, the result is a `TemplateView` class that will dispatch requests to the appropriate matching methods (a behavior defined in the `View` base class), and that has a `render_to_response()` method that uses a `template_name` attribute to return a `TemplateResponse` object (a behavior defined in the `TemplateResponseMixin`).

Mixins are an excellent way of reusing code across multiple classes, but they come with some cost. The more your code is scattered among mixins, the harder it will be to read a child class and know what exactly it is doing, and the harder it will be to know which methods from which mixins to override if you are subclassing something that has a deep inheritance tree.

Note also that you can only inherit from one generic view - that is, only one parent class may inherit from `View` and the rest (if any) should be mixins. Trying to inherit from more than one class that inherits from `View` - for example, trying to use a form at the top of a list and combining `ProcessFormView` and `ListView` - won't work as expected.

Handling forms with class-based views

A basic function-based view that handles forms may look something like this:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render

from .forms import MyForm


def myview(request):
    if request.method == "POST":
        form = MyForm(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")
    else:
        form = MyForm(initial={"key": "value"})

    return render(request, "form_template.html", {"form": form})
```

A similar class-based view might look like:

```
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.views import View

from .forms import MyForm


class MyFormView(View):
    form_class = MyForm
    initial = {"key": "value"}
    template_name = "form_template.html"

    def get(self, request, *args, **kwargs):
        form = self.form_class(initial=self.initial)
        return render(request, self.template_name, {"form": form})

    def post(self, request, *args, **kwargs):
        form = self.form_class(request.POST)
        if form.is_valid():
            # <process form cleaned data>
            return HttpResponseRedirect("/success/")

        return render(request, self.template_name, {"form": form})
```

This is a minimal case, but you can see that you would then have the option of customizing this view by overriding any of the class attributes, e.g. `form_class` , via URLconf configuration, or subclassing and overriding one or more of the methods (or both!).

Decorating class-based views

The extension of class-based views isn't limited to using mixins. You can also use decorators. Since class-based views aren't functions, decorating them works differently depending on if you're using `as_view()` or creating a subclass.

Decorating in URLconf

You can adjust class-based views by decorating the result of the `as_view()` method. The easiest place to do this is in the URLconf where you deploy your view:

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.generic import TemplateView

from .views import VoteView

urlpatterns = [
    path("about/", login_required(TemplateView.as_view(template_name="secret.html"))),
    path("vote/", permission_required("polls.can_vote")(VoteView.as_view())),
]
```

This approach applies the decorator on a per-instance basis. If you want every instance of a view to be decorated, you need to take a different approach.

Decorating the class

To decorate every instance of a class-based view, you need to decorate the class definition itself. To do this you apply the decorator to the `dispatch()` method of the class.

A method on a class isn't quite the same as a standalone function, so you can't just apply a function decorator to the method – you need to transform it into a method decorator first. The `method_decorator` decorator transforms a function decorator into a method decorator so that it can be used on an instance method. For example:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView


class ProtectedView(TemplateView):
    template_name = "secret.html"

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
```

Or, more succinctly, you can decorate the class instead and pass the name of the method to be decorated as the keyword argument `name` :

```
@method_decorator(login_required, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"
```

If you have a set of common decorators used in several places, you can define a list or tuple of decorators and use this instead of invoking `method_decorator()` multiple times. These two classes are equivalent:

```
decorators = [never_cache, login_required]


@method_decorator(decorators, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"


@method_decorator(never_cache, name="dispatch")
@method_decorator(login_required, name="dispatch")
class ProtectedView(TemplateView):
    template_name = "secret.html"
```

The decorators will process a request in the order they are passed to the decorator. In the example, `never_cache()` will process the request before `login_required()` .

In this example, every instance of `ProtectedView` will have login protection. These examples use `login_required` , however, the same behavior can be obtained by using `LoginRequiredMixin`.

Note: `method_decorator` passes `*args` and `**kwargs` as parameters to the decorated method on the class. If your method does not accept a compatible set of parameters it will raise a `TypeError` exception.