Using mixins with class-based views

> **Caution:** This is an advanced topic. A working knowledge of Django's class-based views is advised before exploring these techniques.

Django's built-in class-based views provide a lot of functionality, but some of it you may want to use separately. For instance, you may want to write a view that renders a template to make the HTTP response, but you can't use `TemplateView`; perhaps you need to render a template only on `POST`, with `GET` doing something else entirely. While you could use `TemplateResponse` directly, this will likely result in duplicate code.

For this reason, Django also provides a number of mixins that provide more discrete functionality. Template rendering, for instance, is encapsulated in the `TemplateResponseMixin`. The Django reference documentation contains full documentation of all the mixins.

Context and template responses

Two central mixins are provided that help in providing a consistent interface to working with templates in class-based views.

`TemplateResponseMixin`

Every built in view which returns a `TemplateResponse` will call the `render_to_response()` method that `TemplateResponseMixin` provides. Most of the time this will be called for you (for instance, it is called by the `get()` method implemented by both `TemplateView` and `DetailView`); similarly, it's unlikely that you'll need to override it, although if you want your response to return something not rendered via a Django template then you'll want to do it. For an example of this, see the JSONResponseMixin example.

`render_to_response()` itself calls `get_template_names()`, which by default will look up `template_name` on the class-based view; two other mixins (`SingleObjectTemplateResponseMixin` and `MultipleObjectTemplateResponseMixin`) override this to provide more flexible defaults when dealing with actual objects.

`ContextMixin`

Every built in view which needs context data, such as for rendering a template (including `TemplateResponseMixin` above), should call `get_context_data()` passing any data they want to ensure is in there as keyword arguments. `get_context_data()` returns a dictionary; in `ContextMixin` it returns its keyword arguments, but it is common to override this to add more members to the dictionary. You can also use the `extra_context` attribute.

Building up Django's generic class-based views

Let's look at how two of Django's generic class-based views are built out of mixins providing discrete functionality. We'll consider `DetailView`, which renders a "detail" view of an object, and `ListView`, which will render a list of objects, typically from a queryset, and optionally paginate them. This will introduce us to four mixins which between them provide useful functionality when working with either a single Django object, or multiple objects.

There are also mixins involved in the generic edit views (`FormView`, and the model-specific views `CreateView`, `UpdateView` and `DeleteView`), and in the date-based generic views. These are covered in the mixin reference documentation.

`DetailView`: working with a single Django object

To show the detail of an object, we basically need to do two things: we need to look up the object and then we need to make a `TemplateResponse` with a suitable template, and that object as context.

To get the object, `DetailView` relies on `SingleObjectMixin`, which provides a `get_object()` method that figures out the object based on the URL of the request (it looks for `pk` and `slug` keyword arguments as declared in the URLConf, and looks the object up either from the `model` attribute on the view, or the `queryset` attribute if that's provided). `SingleObjectMixin` also overrides `get_context_data()`, which is used across all Django's built in class-based views to supply context data for template renders.

To then make a `TemplateResponse`, `DetailView` uses `SingleObjectTemplateResponseMixin`, which extends `TemplateResponseMixin`, overriding `get_template_names()` as discussed above. It actually provides a fairly sophisticated set of options, but the main one that most people are going to use is `<app_label>/<model_name>_detail.html`. The `_detail` part can be changed by setting `template_name_suffix` on a subclass to something else. (For instance, the generic edit views use `_form` for create and update views, and `_confirm_delete` for delete views.)

`ListView`: working with many Django objects

Lists of objects follow roughly the same pattern: we need a (possibly paginated) list of objects, typically a `QuerySet`, and then we need to make a `TemplateResponse` with a suitable template using that list of objects.

To get the objects, `ListView` uses `MultipleObjectMixin`, which provides both `get_queryset()` and `paginate_queryset()`. Unlike with `SingleObjectMixin`, there's no need to key off parts of the URL to figure out the queryset to work with, so the default uses the `queryset` or `model` attribute on the view class. A common reason to override `get_queryset()` here would be to dynamically vary the objects, such as depending on the current user or to exclude posts in the future for a blog.

`MultipleObjectMixin` also overrides `get_context_data()` to include appropriate context variables for pagination (providing dummies if pagination is disabled). It relies on `object_list` being passed in as a keyword argument, which `ListView` arranges for it.

To make a `TemplateResponse`, `ListView` then uses `MultipleObjectTemplateResponseMixin`; as with `SingleObjectTemplateResponseMixin` above, this overrides `get_template_names()` to provide `a range of options`, with the most commonly-used being `<app_label>/<model_name>_list.html`, with the `_list` part again being taken from the `template_name_suffix` attribute. (The date based generic views use suffixes such as `_archive`, `_archive_year` and so on to use different templates for the various specialized date-based list views.)

Using Django's class-based view mixins

Now we've seen how Django's generic class-based views use the provided mixins, let's look at other ways we can combine them. We're still going to be combining them with either built-in class-based views, or other generic class-based views, but there are a range of rarer problems you can solve than are provided for by Django out of the box.

Warning: Not all mixins can be used together, and not all generic class based views can be used with all other mixins. Here we present a few examples that do work; if you want to bring together other functionality then you'll have to consider interactions between attributes and methods that overlap between the different classes you're using, and how method resolution order will affect which versions of the methods will be called in what order.

The reference documentation for Django's class-based views and class-based view mixins will help you in understanding which attributes and methods are likely to cause conflict between different classes and mixins.

If in doubt, it's often better to back off and base your work on `View` or `TemplateView`, perhaps with `SingleObjectMixin` and `MultipleObjectMixin`. Although you will probably end up writing more code, it is more likely to be clearly understandable to someone else coming to it later, and with fewer interactions to worry about you will save yourself some thinking. (Of course, you can always dip into Django's implementation of the generic class-based views for inspiration on how to tackle problems.)

Using `SingleObjectMixin` with View

If we want to write a class-based view that responds only to `POST`, we'll subclass `View` and write a `post()` method in the subclass. However if we want our processing to work on a particular object, identified from the URL, we'll want the functionality provided by `SingleObjectMixin`.

We'll demonstrate this with the `Author` model we used in the generic class-based views introduction.

`views.py`

```
from django.http import HttpResponseForbidden, HttpResponseRedirect
from django.urls import reverse
from django.views import View
from django.views.generic.detail import SingleObjectMixin
from books.models import Author


class RecordInterestView(SingleObjectMixin, View):
    """Records the current user's interest in an author."""

    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()

        # Look up the author we're interested in.
        self.object = self.get_object()
        # Actually record interest somehow here!

        return HttpResponseRedirect(
            reverse("author-detail", kwargs={"pk": self.object.pk})
        )
```

In practice you'd probably want to record the interest in a key-value store rather than in a relational database, so we've left that bit out. The only bit of the view that needs to worry about using `SingleObjectMixin` is where we want to look up the author we're interested in, which it does with a call to `self.get_object()`. Everything else is taken care of for us by the mixin.

We can hook this into our URLs easily enough:

`urls.py`

```
from django.urls import path
from books.views import RecordInterestView

urlpatterns = [
    # ...
    path(
        "author/<int:pk>/interest/",
        RecordInterestView.as_view(),
        name="author-interest",
    ),
]
```

Note the `pk` named group, which `get_object()` uses to look up the `Author` instance. You could also use a slug, or any of the other features of `SingleObjectMixin`.

Using `SingleObjectMixin` with `ListView`

`ListView` provides built-in pagination, but you might want to paginate a list of objects that are all linked (by a foreign key) to another object. In our publishing example, you might want to paginate through all the books by a particular publisher.

One way to do this is to combine `ListView` with `SingleObjectMixin`, so that the queryset for the paginated list of books can hang off the publisher found as the single object. In order to do this, we need to have two different querysets:

`Book queryset for use by ListView`

Since we have access to the `Publisher` whose books we want to list, we override `get_queryset()` and use the `Publisher`'s reverse foreign key manager.

`Publisher queryset for use in get_object()`

We'll rely on the default implementation of `get_object()` to fetch the correct `Publisher` object. However, we need to explicitly pass a `queryset` argument because otherwise the default implementation of `get_object()` would call `get_queryset()` which we have overridden to return `Book` objects instead of `Publisher` ones.

Note: We have to think carefully about `get_context_data()`. Since both `SingleObjectMixin` and `ListView` will put things in the context data under the value of `context_object_name` if it's set, we'll instead explicitly ensure the `Publisher` is in the context data. `ListView` will add in the suitable `page_obj` and `paginator` for us providing we remember to call `super()`.

Now we can write a new `PublisherDetailView`:

```
from django.views.generic import ListView
from django.views.generic.detail import SingleObjectMixin
from books.models import Publisher


class PublisherDetailView(SingleObjectMixin, ListView):
    paginate_by = 2
    template_name = "books/publisher_detail.html"

    def get(self, request, *args, **kwargs):
        self.object = self.get_object(queryset=Publisher.objects.all())
        return super().get(request, *args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["publisher"] = self.object
        return context

    def get_queryset(self):
        return self.object.book_set.all()
```

Notice how we set `self.object` within `get()` so we can use it again later in `get_context_data()` and `get_queryset()`. If you don't set `template_name`, the template will default to the normal `ListView` choice, which in this case would be `"books/book_list.html"` because it's a list of books; `ListView` knows nothing about `SingleObjectMixin`, so it doesn't have any clue this view is anything to do with a `Publisher`.

The `paginate_by` is deliberately small in the example so you don't have to create lots of books to see the pagination working! Here's the template you'd want to use:

```
{% extends "base.html" %}

{% block content %}
    <h2>Publisher {{ publisher.name }}</h2>

    <ol>
      {% for book in page_obj %}
        <li>{{ book.title }}</li>
      {% endfor %}
    </ol>

    <div class="pagination">
        <span class="step-links">
            {% if page_obj.has_previous %}
                <a href="?page={{ page_obj.previous_page_number }}">previous</a>
            {% endif %}

            <span class="current">
                Page {{ page_obj.number }} of {{ paginator.num_pages }}.
            </span>
```

```
            {% if page_obj.has_next %}
                <a href="?page={{ page_obj.next_page_number }}">next</a>
            {% endif %}
        </span>
    </div>
{% endblock %}
```

Avoid anything more complex

Generally you can use `TemplateResponseMixin` and `SingleObjectMixin` when you need their functionality. As shown above, with a bit of care you can even combine `SingleObjectMixin` with `ListView`. However things get increasingly complex as you try to do so, and a good rule of thumb is:

> **Hint:** Each of your views should use only mixins or views from one of the groups of generic class-based views: detail, list, editing and date. For example it's fine to combine `TemplateView` (built in view) with `MultipleObjectMixin` (generic list), but you're likely to have problems combining `SingleObjectMixin` (generic detail) with `MultipleObjectMixin` (generic list).

To show what happens when you try to get more sophisticated, we show an example that sacrifices readability and maintainability when there is a simpler solution. First, let's look at a naive attempt to combine `DetailView` with `FormMixin` to enable us to `POST` a Django `Form` to the same URL as we're displaying an object using `DetailView`.

Using `FormMixin` with `DetailView`

Think back to our earlier example of using `View` and `SingleObjectMixin` together. We were recording a user's interest in a particular author; say now that we want to let them leave a message saying why they like them. Again, let's assume we're not going to store this in a relational database but instead in something more esoteric that we won't worry about here.

At this point it's natural to reach for a `Form` to encapsulate the information sent from the user's browser to Django. Say also that we're heavily invested in REST, so we want to use the same URL for displaying the author as for capturing the message from the user. Let's rewrite our `AuthorDetailView` to do that.

We'll keep the `GET` handling from `DetailView`, although we'll have to add a `Form` into the context data so we can render it in the template. We'll also want to pull in form processing from `FormMixin`, and write a bit of code so that on `POST` the form gets called appropriately.

> **Note:** We use `FormMixin` and implement `post()` ourselves rather than try to mix `DetailView` with `FormView` (which provides a suitable `post()` already) because both of the views implement `get()`, and things would get much more confusing.

Our new `AuthorDetailView` looks like this:

```
# CAUTION: you almost certainly do not want to do this.
# It is provided as part of a discussion of problems you can
# run into when combining different generic class-based view
# functionality that is not designed to be used together.

from django import forms
from django.http import HttpResponseForbidden
from django.urls import reverse
from django.views.generic import DetailView
from django.views.generic.edit import FormMixin
from books.models import Author


class AuthorInterestForm(forms.Form):
    message = forms.CharField()


class AuthorDetailView(FormMixin, DetailView):
    model = Author
    form_class = AuthorInterestForm

    def get_success_url(self):
        return reverse("author-detail", kwargs={"pk": self.object.pk})

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
        self.object = self.get_object()
        form = self.get_form()
        if form.is_valid():
            return self.form_valid(form)
        else:
            return self.form_invalid(form)

    def form_valid(self, form):
        # Here, we would record the user's interest using the message
        # passed in form.cleaned_data['message']
        return super().form_valid(form)
```

`get_success_url()` provides somewhere to redirect to, which gets used in the default implementation of `form_valid()`. We have to provide our own `post()` as noted earlier.

A better solution

The number of subtle interactions between `FormMixin` and `DetailView` is already testing our ability to manage things. It's unlikely you'd want to write this kind of class yourself.

In this case, you could write the `post()` method yourself, keeping `DetailView` as the only generic functionality, although writing `Form` handling code involves a lot of duplication.

Alternatively, it would still be less work than the above approach to have a separate view for processing the form, which could use `FormView` distinct from `DetailView` without concerns.

An alternative better solution

What we're really trying to do here is to use two different class based views from the same URL. So why not do just that? We have a very clear division here: `GET` requests should get the `DetailView` (with the `Form` added to the context data), and `POST` requests should get the `FormView`. Let's set up those views first.

The `AuthorDetailView` view is almost the same as when we first introduced AuthorDetailView; we have to write our own `get_context_data()` to make the `AuthorInterestForm` available to the template. We'll skip the `get_object()` override from before for clarity:

```
from django import forms
from django.views.generic import DetailView
from books.models import Author


class AuthorInterestForm(forms.Form):
    message = forms.CharField()


class AuthorDetailView(DetailView):
    model = Author

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["form"] = AuthorInterestForm()
        return context
```

Then the `AuthorInterestFormView` is a `FormView`, but we have to bring in `SingleObjectMixin` so we can find the author we're talking about, and we have to remember to set `template_name` to ensure that form errors will render the same template as `AuthorDetailView` is using on `GET`:

```
from django.http import HttpResponseForbidden
from django.urls import reverse
from django.views.generic import FormView
from django.views.generic.detail import SingleObjectMixin


class AuthorInterestFormView(SingleObjectMixin, FormView):
    template_name = "books/author_detail.html"
    form_class = AuthorInterestForm
    model = Author

    def post(self, request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponseForbidden()
```

```
        self.object = self.get_object()
        return super().post(request, *args, **kwargs)

    def get_success_url(self):
        return reverse("author-detail", kwargs={"pk": self.object.pk})
```

Finally we bring this together in a new `AuthorView` view. We already know that calling `as_view()` on a class-based view gives us something that behaves exactly like a function based view, so we can do that at the point we choose between the two subviews.

You can pass through keyword arguments to `as_view()` in the same way you would in your URLconf, such as if you wanted the `AuthorInterestFormView` behavior to also appear at another URL but using a different template:

```
from django.views import View


class AuthorView(View):
    def get(self, request, *args, **kwargs):
        view = AuthorDetailView.as_view()
        return view(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        view = AuthorInterestFormView.as_view()
        return view(request, *args, **kwargs)
```

This approach can also be used with any other generic class-based views or your own class-based views inheriting directly from `View` or `TemplateView`, as it keeps the different views as separate as possible.

More than just HTML

Where class-based views shine is when you want to do the same thing many times. Suppose you're writing an API, and every view should return JSON instead of rendered HTML.

We can create a mixin class to use in all of our views, handling the conversion to JSON once.

For example, a JSON mixin might look something like this:

```
from django.http import JsonResponse


class JSONResponseMixin:
    """
    A mixin that can be used to render a JSON response.
    """

    def render_to_json_response(self, context, **response_kwargs):
        """
        Returns a JSON response, transforming 'context' to make the payload.
        """
        return JsonResponse(self.get_data(context), **response_kwargs)
```

```
    def get_data(self, context):
        """
        Returns an object that will be serialized as JSON by json.dumps().
        """
        # Note: This is *EXTREMELY* naive; in reality, you'll need
        # to do much more complex handling to ensure that arbitrary
        # objects -- such as Django model instances or querysets
        # -- can be serialized as JSON.
        return context
```

Note: Check out the Serializing Django objects documentation for more information on how to correctly transform Django models and querysets into JSON.

This mixin provides a `render_to_json_response()` method with the same signature as `render_to_response()`. To use it, we need to mix it into a `TemplateView` for example, and override `render_to_response()` to call `render_to_json_response()` instead:

```
from django.views.generic import TemplateView


class JSONView(JSONResponseMixin, TemplateView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

Equally we could use our mixin with one of the generic views. We can make our own version of `DetailView` by mixing `JSONResponseMixin` with the `BaseDetailView` – (the `DetailView` before template rendering behavior has been mixed in):

```
from django.views.generic.detail import BaseDetailView


class JSONDetailView(JSONResponseMixin, BaseDetailView):
    def render_to_response(self, context, **response_kwargs):
        return self.render_to_json_response(context, **response_kwargs)
```

This view can then be deployed in the same way as any other `DetailView`, with exactly the same behavior – except for the format of the response.

If you want to be really adventurous, you could even mix a `DetailView` subclass that is able to return *both* HTML and JSON content, depending on some property of the HTTP request, such as a query argument or an HTTP header. Mix in both the `JSONResponseMixin` and a `SingleObjectTemplateResponseMixin`, and override the implementation of `render_to_response()` to defer to the appropriate rendering method depending on the type of response that the user requested:

```
from django.views.generic.detail import SingleObjectTemplateResponseMixin


class HybridDetailView(
    JSONResponseMixin, SingleObjectTemplateResponseMixin, BaseDetailView
```

```
):
    def render_to_response(self, context):
        # Look for a 'format=json' GET argument
        if self.request.GET.get("format") == "json":
            return self.render_to_json_response(context)
        else:
            return super().render_to_response(context)
```

Because of the way that Python resolves method overloading, the call to `super().render_to_response(context)` ends up calling the `render_to_response()` implementation of `TemplateResponseMixin`.