Form handling with class-based views

Form processing generally has 3 paths:

- Initial GET (blank or prepopulated form)
- POST with invalid data (typically redisplay form with errors)
- POST with valid data (process the data and typically redirect)

Implementing this yourself often results in a lot of repeated boilerplate code (see Using a form in a view). To help avoid this, Django provides a collection of generic class-based views for form processing.

Basic forms

Given a contact form:

`forms.py`

```
from django import forms


class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=forms.Textarea)

    def send_email(self):
        # send email using the self.cleaned_data dictionary
        pass
```

The view can be constructed using a `FormView`:

`views.py`

```
from myapp.forms import ContactForm
from django.views.generic.edit import FormView


class ContactFormView(FormView):
    template_name = "contact.html"
    form_class = ContactForm
    success_url = "/thanks/"

    def form_valid(self, form):
        # This method is called when valid form data has been POSTed.
        # It should return an HttpResponse.
        form.send_email()
        return super().form_valid(form)
```

Notes:

- FormView inherits `TemplateResponseMixin` so `template_name` can be used here.
- The default implementation for `form_valid()` simply redirects to the `success_url`.

Model forms

Generic views really shine when working with models. These generic views will automatically create a `ModelForm`, so long as they can work out which model class to use:

- If the `model` attribute is given, that model class will be used.
- If `get_object()` returns an object, the class of that object will be used.
- If a `queryset` is given, the model for that queryset will be used.

Model form views provide a `form_valid()` implementation that saves the model automatically. You can override this if you have any special requirements; see below for examples.

You don't even need to provide a `success_url` for `CreateView` or `UpdateView` - they will use `get_absolute_url()` on the model object if available.

If you want to use a custom `ModelForm` (for instance to add extra validation), set `form_class` on your view.

> **Note:** When specifying a custom form class, you must still specify the model, even though the `form_class` may be a `ModelForm`.

First we need to add `get_absolute_url()` to our `Author` class:

`models.py`

```
from django.db import models
from django.urls import reverse


class Author(models.Model):
    name = models.CharField(max_length=200)

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

Then we can use `CreateView` and friends to do the actual work. Notice how we're just configuring the generic class-based views here; we don't have to write any logic ourselves:

`views.py`

```
from django.urls import reverse_lazy
from django.views.generic.edit import CreateView, DeleteView, UpdateView
from myapp.models import Author
```

```
class AuthorCreateView(CreateView):
    model = Author
    fields = ["name"]


class AuthorUpdateView(UpdateView):
    model = Author
    fields = ["name"]


class AuthorDeleteView(DeleteView):
    model = Author
    success_url = reverse_lazy("author-list")
```

Note: We have to use `reverse_lazy()` instead of `reverse()`, as the urls are not loaded when the file is imported.

The `fields` attribute works the same way as the `fields` attribute on the inner `Meta` class on `ModelForm`. Unless you define the form class in another way, the attribute is required and the view will raise an `ImproperlyConfigured` exception if it's not.

If you specify both the `fields` and `form_class` attributes, an `ImproperlyConfigured` exception will be raised.

Finally, we hook these new views into the URLconf:

`urls.py`

```
from django.urls import path
from myapp.views import AuthorCreateView, AuthorDeleteView, AuthorUpdateView

urlpatterns = [
    # ...
    path("author/add/", AuthorCreateView.as_view(), name="author-add"),
    path("author/<int:pk>/", AuthorUpdateView.as_view(), name="author-update"),
    path("author/<int:pk>/delete/", AuthorDeleteView.as_view(), name="author-delete"),
]
```

Note: These views inherit `SingleObjectTemplateResponseMixin` which uses `template_name_suffix` to construct the `template_name` based on the model.

In this example:

- `CreateView` and `UpdateView` use `myapp/author_form.html`
- `DeleteView` uses `myapp/author_confirm_delete.html`

If you wish to have separate templates for `CreateView` and `UpdateView`, you can set either `template_name` or `template_name_suffix` on your view class.

Models and `request.user`

To track the user that created an object using a `CreateView`, you can use a custom `ModelForm` to do this. First, add the foreign key relation to the model:

`models.py`

```
from django.contrib.auth.models import User
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=200)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)

    # ...
```

In the view, ensure that you don't include `created_by` in the list of fields to edit, and override `form_valid()` to add the user:

`views.py`

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic.edit import CreateView
from myapp.models import Author


class AuthorCreateView(LoginRequiredMixin, CreateView):
    model = Author
    fields = ["name"]

    def form_valid(self, form):
        form.instance.created_by = self.request.user
        return super().form_valid(form)
```

`LoginRequiredMixin` prevents users who aren't logged in from accessing the form. If you omit that, you'll need to handle unauthorized users in `form_valid()`.

Content negotiation example

Here is an example showing how you might go about implementing a form that works with an API-based workflow as well as 'normal' form POSTs:

```
from django.http import JsonResponse
from django.views.generic.edit import CreateView
from myapp.models import Author


class JsonableResponseMixin:
    """
```

```
    Mixin to add JSON support to a form.
    Must be used with an object-based FormView (e.g. CreateView)
    """

    def form_invalid(self, form):
        response = super().form_invalid(form)
        if self.request.accepts("text/html"):
            return response
        else:
            return JsonResponse(form.errors, status=400)

    def form_valid(self, form):
        # We make sure to call the parent's form_valid() method because
        # it might do some processing (in the case of CreateView, it will
        # call form.save() for example).
        response = super().form_valid(form)
        if self.request.accepts("text/html"):
            return response
        else:
            data = {
                "pk": self.object.pk,
            }
            return JsonResponse(data)


class AuthorCreateView(JsonableResponseMixin, CreateView):
    model = Author
    fields = ["name"]
```