9 Common Patterns for Forms

Django forms are powerful, flexible, extensible, and robust. For this reason, the Django admin and CBVs use them extensively. In fact, all the major Django API frameworks use ModelForms as part of their validation because of their powerful validation features.

Combining forms, models, and views allows us to get a lot of work done for little effort. The learning curve is worth it: once you learn to work fluently with these components, you'll find that Django provides the ability to create an amazing amount of useful, stable functionality at an amazing pace.

PACKAGE TIP: Useful Form-Related Packages

- **django-floppyforms** for rendering Django inputs in HTML5.
- **django-crispy-forms** for advanced form layout controls. By default, forms are rendered with Twitter Bootstrap form elements and styles. This package plays well with django-floppyforms, so they are often used together.
- **django-forms-bootstrap** is a simple tool for rendering Django forms using Twitter Bootstrap styles. This package plays well with django-floppyforms but conflicts with django-crispy-forms.

9.1 The Power of Django Forms

You might not be aware of the fact that even if your Django project uses an API framework and doesn't serve HTML, you are probably still using Django forms. Django forms are not just for web pages; their powerful validation features are useful on their own.

Interestingly enough, the design that Django's API frameworks use is some form of class-based view. They might have their own implementation of CBVs (i.e. django-tastypie) or run off of Django's own CBVs (django-rest-framework), but the use of inheritance and composition is a constant. We would like to think this is proof of the soundness of both Django forms and the concept of CBVs.

With that in mind, this chapter goes explicitly into one of the best parts of Django: forms, models, and CBVs working in concert. This chapter covers five common form patterns that should be in every Django developer's toolbox.

9.2 Pattern 1: Simple ModelForm With Default Validators

The simplest data-changing form that we can make is a `ModelForm` using several default validators as-is, without modification. In fact, we already relied on default validators in chapter 8, *Best Practices for Class-Based Views*, subsection 8.4.1, "Views + ModelForm Example."

If you recall, using `ModelForms` with CBVs to implement add/edit forms can be done in just a few lines of code:

EXAMPLE 9.1

```
# flavors/views.py
from django.views.generic import CreateView, UpdateView

from braces.views import LoginRequiredMixin

from .models import Flavor

class FlavorCreateView(LoginRequiredMixin, CreateView):
    model = Flavor

class FlavorUpdateView(LoginRequiredMixin, UpdateView):
    model = Flavor
```

To summarize how we use default validation as-is here:

- `FlavorCreateView` and `FlavorUpdateView` are assigned `Flavor` as their model.

- Both views auto-generate a `ModelForm` based on the Flavor model.
- Those `ModelForms` rely on the default field validation rules of the Flavor model.

Yes, Django gives us a lot of great defaults for data validation, but in practice, the defaults are never enough. We recognize this, so as a first step, the next pattern will demonstrate how to create a custom field validator.

9.3 Pattern 2: Custom Form Field Validators in ModelForms

What if we wanted to be certain that every use of the `title` field across our project's dessert apps started with the word 'Tasty'?

This is a string validation problem that can be solved with a simple **custom field validator**.

In this pattern, we cover how to create custom single-field validators and demonstrate how to add them to both abstract models and forms.

Imagine for the purpose of this example that we have a project with two different dessert-related models: a `Flavor` model for ice cream flavors, and a `Milkshake` model for different types of milkshakes. Assume that both of our example models have `title` fields.

To validate all editable model titles, we start by creating a ***validators.py*** module:

Example 9.2

```
# core/validators.py
from django.core.exceptions import ValidationError

def validate_tasty(value):
    """ Raise a ValidationError if the
        value doesn't start with the
        word 'Tasty'
    """
    if not value.startswith(u"Tasty"):
        msg = u"Must start with Tasty"
        raise ValidationError(msg)
```

In Django, a custom field validator is simply a function that raises an error if the submitted argument doesn't pass its test.

Of course, while our `validate_tasty()` validator function just does a simple string check for the sake of example, it's good to keep in mind that form field validators can become quite complex in practice.

TIP: Test Your Validators Carefully

Since validators are critical in keeping corruption out of Django project databases, it's especially important to write detailed tests for them.

These tests should include thoughtful edge case tests for every condition related to your validators' custom logic.

In order to use our `validate_tasty()` validator function across different dessert models, we're going to first add it to an abstract model called `TastyTitleAbstractModel`, which we plan to use across our project.

Assuming that our `Flavor` and `Milkshake` models are in separate apps, it doesn't make sense to put our validator in one app or the other. Instead, we create a ***core/models.py*** module and place the `TastyTitleAbstractModel` there.

Example 9.3

```
# core/models.py
from django.db import models

from .validators import validate_tasty

class TastyTitleAbstractModel(models.Model):

    title = models.CharField(max_length=255, validators=[validate_tasty])

    class Meta:
        abstract = True
```

The last two lines of the above example code for ***core/models.py*** make `TastyTitleAbstractModel` an abstract model, which is what we want.

Let's alter the original ***flavors/models.py*** `Flavor` code to use `TastyTitleAbstractModel` as the parent class:

EXAMPLE 9.4

```
# flavors/models.py
from django.core.urlresolvers import reverse
from django.db import models

from core.models import TastyTitleAbstractModel

class Flavor(TastyTitleAbstractModel):
    slug = models.SlugField()
    scoops_remaining = models.IntegerField(default=0)

    def get_absolute_url(self):
        return reverse("flavor_detail", kwargs={"slug": self.slug})
```

This works with the `Flavor` model, and it will work with any other tasty food-based model such as a `WaffleCone` or `Cake` model. Any model that inherits from the `TastyTitleAbstractModel` class will throw a validation error if anyone attempts to save a model with a title that doesn't start with 'Tasty'.

Now, let's explore a couple of questions that might be forming in your head:

- What if we wanted to use `validate_tasty()` in just forms?
- What if we wanted to assign it to other fields besides the title?

To support these behaviors, we need to create a custom `FlavorForm` that utilizes our custom field validator:

EXAMPLE 9.5

```
# flavors/forms.py
from django import forms
```

```
from core.validators import validate_delicious
from .models import Flavor

class FlavorForm(forms.ModelForm):
    def __init__(self, *args, **kwargs):
        super(FlavorForm, self).__init__(*args, **kwargs)
        self.fields["title"].validators.append(validate_delicious)
        self.fields["slug"].validators.append(validate_delicious)

    class Meta:
        model = Flavor
```

A nice thing about both examples of validator usage in this pattern is that we haven't had to change the `validate_tasty()` code at all. Instead, we just import and use it in new places.

Attaching the custom form to the views is our next step. The default behavior of Django model-based edit views is to auto-generate the ModelForm based on the view's model attribute. We are going to override that default and pass in our custom `FlavorForm`. This occurs in the ***flavors/views.py*** module, where we alter the create and update forms as demonstrated below:

Example 9.6

```
# flavors/views.py
from django.contrib import messages
from django.views.generic import CreateView, UpdateView, DetailView

from braces.views import LoginRequiredMixin

from .models import Flavor
from .forms import FlavorForm

class FlavorActionMixin(object):

    @property
    def action(self):
        msg = "{0} is missing action.".format(self.__class__)
        raise NotImplementedError(msg)
```

```
    def form_valid(self, form):
        msg = "Flavor {0}!".format(self.action)
        messages.info(self.request, msg)
        return super(FlavorActionMixin, self).form_valid(form)

class FlavorCreateView(LoginRequiredMixin, FlavorActionMixin,
                       CreateView):
    model = Flavor
    action = "created"
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorUpdateView(LoginRequiredMixin, FlavorActionMixin,
                       UpdateView):
    model = Flavor
    action = "updated"
    # Explicitly attach the FlavorForm class
    form_class = FlavorForm

class FlavorDetailView(DetailView):
    model = Flavor
```

The `FlavorCreateView` and `FlavorUpdateView` views now use the new `FlavorForm` to validate incoming data.

Note that with these modifications, the `Flavor` model can either be identical to the one at the start of this chapter, or it can be an altered one that inherits from `TastyTitleAbstractModel`.

9.4 Pattern 3: Overriding the Clean Stage of Validation

Let's discuss some interesting validation use cases:

- Multi-field validation
- Validation involving existing data from the database that has already been validated

Both of these are great scenarios for overriding the `clean()` and `clean_<field_name>()` methods with custom validation logic.

After the default and custom field validators are run, Django provides a second stage and process for validating incoming data, this time via the `clean()` method and `clean_<field_name>()` methods. You might wonder why Django provides more hooks for validation, so here are our two favorite arguments:

1. The `clean()` method is the place to validate two or more fields against each other, since it's not specific to any one particular field.
2. The clean validation stage is a better place to attach validation against persistent data. Since the data already has some validation, you won't waste as many database cycles on needless queries.

Let's explore this with another validation example. Perhaps we want to implement an ice cream ordering form, where users could specify the flavor desired, add toppings, and then come to our store and pick them up.

Since we want to prevent users from ordering flavors that are out of stock, we'll put in a `clean_slug()` method. With our flavor validation, our form might look like:

EXAMPLE 9.7

```
# flavors/forms.py
from django import forms
from flavors.models import Flavor

class IceCreamOrderForm(forms.Form):
    """ Normally done with forms.ModelForm. But we use forms.Form here
        to demonstrate that these sorts of techniques work on every
        type of form.
    """

    slug = forms.ChoiceField("Flavor")
    toppings = forms.CharField()

    def __init__(self, *args, **kwargs):
        super(IceCreamOrderForm, self).__init__(*args,
                    **kwargs)
        # We dynamically set the choices here rather than
```

```
        # in the flavor field definition. Setting them in
        # the field definition means status updates won't
        # be reflected in the form without server restarts.
        self.fields["slug"].choices = [
            (x.slug, x.title) for x in Flavor.objects.all()
        ]
        # NOTE: We could filter by whether or not a flavor
        #       has any scoops, but this is an example of
        #       how to use clean_slug, not filter().

    def clean_slug(self):
        slug = self.cleaned_data["slug"]
        if Flavor.objects.get(slug=slug).scoops_remaining <= 0:
            msg = u"Sorry, we are out of that flavor."
            raise forms.ValidationError(msg)
        return slug
```

For HTML-powered views, the `clean_slug()` method in our example, upon throwing an error, will attach a "Sorry, we are out of that flavor" message to the flavor HTML input field. This is a great shortcut for writing HTML forms!

Now imagine if we get common customer complaints about orders with too much chocolate. Yes, it's silly and quite impossible, but we're just using 'too much chocolate' as a completely mythical example for the sake of making a point.

In any case, let's use the `clean()` method to validate the flavor and toppings fields against each other.

Example 9.8

```
    # attach this code to the previous example (9.13)
    def clean(self):
        cleaned_data = super(IceCreamOrderForm, self).clean()
        slug = cleaned_data.get("slug", "")
        toppings = cleaned_data.get("toppings", "")

        # Silly "too much chocolate" validation example
```

```
        if u"chocolate" in slug.lower() and \
               u"chocolate" in toppings.lower():
            msg = u"Your order has too much chocolate."
            raise forms.ValidationError(msg)
        return cleaned_data
```

There we go, an implementation against the impossible condition of too much chocolate!

9.5 Pattern 4: Hacking Form Fields (2 CBVs, 2 Forms, 1 Model)

This is where we start to get fancy. We're going to cover a situation where two views/forms correspond to one model. We'll hack Django forms to produce a form with custom behavior.

It's not uncommon to have users create a record that contains a few empty fields which need additional data later. An example might be a list of stores, where we want each store entered into the system as fast as possible, but want to add more data such as phone number and description later. Here's our `IceCreamStore` model:

Example 9.9

```
# stores/models.py
from django.core.urlresolvers import reverse
from django.db import models

class IceCreamStore(models.Model):
    title = models.CharField(max_length=100)
    block_address = models.TextField()
    phone = models.CharField(max_length=20, blank=True)
    description = models.TextField(blank=True)

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

The default `ModelForm` for this model forces the user to enter the `title` and `block_address` field but allows the user to skip the `phone` and `description` fields. That's great for initial data entry, but

as mentioned earlier, we want to have future updates of the data to require the phone and description fields.

The way we implemented this in the past before we began to delve into their construction was to override the phone and description fields in the edit form. This resulted in heavily-duplicated code that looked like this:

Bad Example 9.1

```
# stores/forms.py
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):
    # Don't do this! Duplication of the model field!
    phone = forms.CharField(required=True)
    # Don't do this! Duplication of the model field!
    description = forms.TextField(required=True)

    class Meta:
        model = IceCreamStore
```

This form should look very familiar. Why is that?

Well, we're nearly copying the `IceCreamStore` model!

This is just a simple example, but when dealing with a lot of fields on a model, the duplication becomes extremely challenging to manage. In fact, what tends to happen is copy-pasting of code from models right into forms, which is a gross violation of **Don't Repeat Yourself**.

Want to know how gross? Using the above approach, if we add a simple `help_text` attribute to the `description` field in the model, it will not show up in the template until we also modify the `description` field definition in the form. If that sounds confusing, that's because it is.

A better way is to rely on a useful little detail that's good to remember about Django forms: instantiated form objects store fields in a dict-like attribute called `fields`.

Instead of copy-pasting field definitions from models to forms, we can simply apply new attributes to each field in the `__init__()` method of the `ModelForm`:

Example 9.10

```
# stores/forms.py
# Call phone and description from the self.fields dict-like object
from django import forms

from .models import IceCreamStore

class IceCreamStoreUpdateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore

    def __init__(self, *args, **kwargs):
        # Call the original __init__ method before assigning
        # field overloads
        super(IceCreamStoreUpdateForm, self).__init__(*args,
                              **kwargs)
        self.fields["phone"].required = True
        self.fields["description"].required = True
```

This improved approach allows us to stop copy-pasting code and instead focus on just the field-specific settings.

An important point to remember is that when it comes down to it, Django forms are just Python classes. They get instantiated as objects, they can inherit from other classes, and they can act as superclasses.

Therefore, we can rely on inheritance to trim the line count in our ice cream store forms:

Example 9.11

```
# stores/forms.py
from django import forms

from .models import IceCreamStore
```

```
class IceCreamStoreCreateForm(forms.ModelForm):

    class Meta:
        model = IceCreamStore
        fields = ("title", "block_address", )

class IceCreamStoreUpdateForm(IceCreamStoreCreateForm):

    def __init__(self, *args, **kwargs):
        super(IceCreamStoreUpdateForm,
                self).__init__(*args, **kwargs)
        self.fields["phone"].required = True
        self.fields["description"].required = True

    class Meta(IceCreamStoreCreateForm.Meta):
        # show all the fields!
        fields = ("title", "block_address", "phone",
                "description", )
```

WARNING: Use Meta.fields and Never Use Meta.exclude

We use `Meta.fields` instead of `Meta.exclude` so that we know exactly what fields we are exposing. See chapter 21, *Security Best Practices*, section 21.12, 'Don't use ModelForms.Meta.exclude'.

Finally, now we have what we need to define the corresponding CBVs. We've got our form classes, so let's use them in the `IceCreamStore` create and update views:

Example 9.12

```
# stores/views
from django.views.generic import CreateView, UpdateView

from .forms import IceCreamStoreCreateForm
from .forms import IceCreamStoreUpdateForm
from .models import IceCreamStore
```

```
class IceCreamCreateView(CreateView):
    model = IceCreamStore
    form_class = IceCreamStoreCreateForm

class IceCreamUpdateView(UpdateView):
    model = IceCreamStore
    form_class = IceCreamStoreUpdateForm
```

We now have two views and two forms that work with one model.

9.6 Pattern 5: Reusable Search Mixin View

In this example, we're going to cover how to reuse a search form in two views that correspond to two different models.

Assume that both models have a field called `title` (this pattern also demonstrates why naming standards in projects is a good thing). This example will demonstrate how a single CBV can be used to provide simple search functionality on both the `Flavor` and `IceCreamStore` models.

We'll start by creating a simple search mixin for our view:

Example 9.13

```
# core/views.py
class TitleSearchMixin(object):

    def get_queryset(self):
        # Fetch the queryset from the parent's get_queryset
        queryset = super(TitleSearchMixin, self).get_queryset()

        # Get the q GET parameter
        q = self.request.GET.get("q")
        if q:
            # return a filtered queryset
            return queryset.filter(title__icontains=q)
```

```
        # No q is specified so we return queryset
        return queryset
```

The above code should look very familiar as we used it almost verbatim in the Forms + View example. Here's how you make it work with both the `Flavor` and `IceCreamStore` views. First the flavor views:

Example 9.14

```
# add to flavors/views.py
from django.views.generic import ListView

from core.views import TitleSearchMixin
from .models import Flavor

class FlavorListView(TitleSearchMixin, ListView):
    model = Flavor
```

And we'll add it to the ice cream store views:

Example 9.15

```
# add to stores/views.py
from django.views.generic import ListView

from core.views import TitleSearchMixin
from .models import Store

class IceCreamStoreListView(TitleSearchMixin, ListView):
    model = Store
```

As for the form? We just define it in HTML for each `ListView`:

Example 9.16

```
{# form to go into stores/store_list.html template #}
<form action="" method="GET">
```

```
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

and

Example 9.17

```
{# form to go into flavors/flavor_list.html template #}
<form action="" method="GET">
    <input type="text" name="q" />
    <button type="submit">search</button>
</form>
```

Now we have the same mixin in both views. Mixins are a good way to reuse code, but using too many mixins in a single class makes for very hard-to-maintain code. As always, try to keep your code as simple as possible.

9.7 Summary

We began this chapter with the simplest form pattern, using a `ModelForm`, CBV, and default validators. We iterated on that with an example of a custom validator.

Next, we explored more complex validation. We covered an example overriding the clean methods. We also closely examined a scenario involving two views and their corresponding forms that were tied to a single model.

Finally, we covered an example of creating a reusable search mixin to add the same form to two different apps.