Building a form

The work that needs to be done

Suppose you want to create a simple form on your website, in order to obtain the user’s name. You’d need

something like this in your template:

<form action="/your-name/" method="post">

<label for="your\_name">Your name: </label>

<input id="your\_name" type="text" name="your\_name" value="{{ current\_name }}">

<input type="submit" value="OK">

</form>

This tells the browser to return the form data to the URL /your-name/, using the POST method. It will

display a text field, labeled “Your name:”, and a button marked “**OK**”. If the template context contains a

**current\_name** variable, that will be used to pre-fill the **your\_name** field.

You’ll need a view that renders the template containing the HTML form, and that can supply the

**current\_name** field as appropriate.

When the form is submitted, the **POST** request which is sent to the server will contain the form data.

Now you’ll also need a view corresponding to that /**your-name/ URL** which will find the appropriate

key/value pairs in the **request**, and then process them.

# Building a form in Django

The Form class

We already know what we want our HTML form to look like. Our starting point for it in Django is this:

Listing 10: forms.py

from django import forms

class NameForm(forms.Form):

your\_name = forms.CharField(label="Your name", max\_length=100)

This defines a Form class with a single field (**your\_name**). We’ve applied a human-friendly label to the field,

which will appear in the **<label>** when it’s rendered (although in this case, the label we specified is actually

the same one that would be generated automatically if we had omitted it).

The field’s maximum allowable length is defined by **max\_length**. This does two things. It puts a

**maxlength="100**" on the **HTML <input>** (so the browser should prevent the user from entering more than

that number of characters in the first place). It also means that when Django receives the form back from

the browser, it will validate the length of the data.

A Form instance has an **is\_valid()** method, which runs validation routines for all its fields. When this

method is called, if all fields contain valid data, it will:

• return True

• place the form’s data in its **cleaned\_data** attribute.

The whole form, when rendered for the first time, will look like:

<label for="your\_name">Your name: </label>

<input id="your\_name" type="text" name="your\_name" maxlength="100" required>

The view

Form data sent back to a Django website is processed by a view, generally the same view which published

the form. This allows us to reuse some of the same logic.

To handle the form we need to instantiate it in the view for the URL where we want it to be published:

Listing 11: views.py

from django.http import HttpResponseRedirect

from django.shortcuts import render

from .forms import NameForm

def get\_name(request):

# if this is a POST request we need to process the form data

if request.method == "POST":

# create a form instance and populate it with data from the request:

form = NameForm(request.POST)

# check whether it's valid:

if form.is\_valid():

# process the data in form.cleaned\_data as required

# ...

# redirect to a new URL:

return HttpResponseRedirect("/thanks/")

# if a GET (or any other method) we'll create a blank form

else:

form = NameForm()

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

If we arrive at this view with a **GET** request, it will create an empty form instance and place it in the template

context to be rendered. This is what we can expect to happen the first time we visit the **URL**.

If the form is submitted using a POST request, the view will once again create a form instance and populate

it with data from the request: **form = NameForm(request.POST)** This is called “binding data to the form”

(it is now a bound form).

We call the form’s **is\_valid()** method; if it’s not True, we go back to the template with the form. This

time the form is no longer empty (unbound) so the **HTML** form will be populated with the data previously

submitted, where it can be edited and corrected as required.

If i**s\_valid()** is **True**, we’ll now be able to find all the validated form data in its **cleaned\_data** attribute.

We can use this data to update the database or do other processing before sending an HTTP redirect to the

browser telling it where to go next.

# The template - - -

We don’t need to do much in our name.html template:

<form action="/your-name/" method="post">

{% csrf\_token %}

{{ form }}

<input type="submit" value="Submit">

</form>

All the form’s fields and their attributes will be unpacked into HTML markup from that {{ form }} by Django’s template language

HTML5 input types and browser validation

If your form includes a **URLField,** an **EmailField** or any integer field type, Django will use the url, email and

number HTML5 input types. By default, browsers may apply their own validation on these fields, which may

be stricter than Django’s validation. If you would like to disable this behavior, set the **novalidate** attribute

on the form tag, or specify a different widget on the field, like **TextInput**

Forms and Cross Site Request Forgery protection

Django ships with an easy-to-use protection against Cross Site Request Forgeries. When submitting a form

via POST with CSRF protection enabled you must use the **csrf\_token** template tag as in the preceding example. However, since CSRF protection is not directly tied to forms in templates, this tag is omitted from the

following examples in this document.

All form classes are created as subclasses of either **django.forms.Form** or **django.forms.ModelForm**. You

can think of **ModelForm** as a subclass of **Form**. **Form** and **ModelForm** actually inherit common functionality

from a (private) **BaseForm** class, but this implementation detail is rarely important.

Models and Forms

In fact, if your form is going to be used to directly add or edit a **Django model**, a **ModelForm** can save you a

great deal of time, effort, and code, because it will build a form, along with the appropriate fields and their

attributes, from a **Model** **class**.

Bound and unbound form instances

The distinction between Bound and unbound forms is important:

• An unbound form has no data associated with it. When rendered to the user, it will be empty or will

contain default values.

• A bound form has submitted data, and hence can be used to tell if that data is valid. If an invalid bound form is rendered, it can include inline error messages telling the user what data to correct.

The form’s is\_bound attribute will tell you whether a form has data bound to it or not.

## More on fields - -

Consider a more useful form than our minimal example above, which we could use to implement “contact

me” functionality on a personal website:

Listing 12: forms.py

from django import forms

class ContactForm(forms.Form):

subject = forms.CharField(max\_length=100)

message = forms.CharField(widget=forms.Textarea)

sender = forms.EmailField()

cc\_myself = forms.BooleanField(required=False)

Our earlier form used a single field, **your\_name**, a CharField. In this case, our form has four **fields**: subject,

message, sender and **cc\_myself**. **CharField**, **EmailField** and **BooleanField** are just three of the available

field types; a full list can be found in Form fields.

## Widgets

Each form field has a corresponding Widget class, which in turn corresponds to an HTML form widget such

as <input type="text">.

In most cases, the field will have a sensible default widget. For example, by default, a **CharField** will have a

**TextInput** widget, that produces an <input type="text"> in the HTML. If you needed **<textarea>** instead,

you’d specify the appropriate widget when defining your form field, as we have done for the message field.

## Field data

Whatever the data submitted with a form, once it has been successfully validated by calling **is\_valid()**

(and **is\_valid()** has returned **True**), the validated form data will be in the **form.cleaned\_data** dictionary.

This data will have been nicely converted into Python types for you.

Note: You can still access the unvalidated data directly from **request.POST** at this point, but the validated

data is better.

In the contact form example above**, cc\_myself** will be a **boolean** value. Likewise, fields such as **IntegerField**

and **FloatField** convert values to a **Python** **int** and **float** respectively.

Here’s how the **form data** could be processed in the view that handles this form:

Listing 13: views.py

from django.core.mail import send\_mail

if form.is\_valid():

subject = form.cleaned\_data["subject"]

message = form.cleaned\_data["message"]

sender = form.cleaned\_data["sender"]

cc\_myself = form.cleaned\_data["cc\_myself"]

recipients = ["info@example.com"]

if cc\_myself:

recipients.append(sender)

send\_mail(subject, message, sender, recipients)

return HttpResponseRedirect("/thanks/")

Some field types need some extra handling. For example, files that are uploaded using a form need to be

handled differently (they can be retrieved from **request.FILES**, rather than **request.POST**).

----------------------------- END ---------------