Working with forms

About this document: This document provides an introduction to the basics of web forms and how they are handled in Django. For a more detailed look at specific areas of the forms API, see The Forms API, Form fields, and Form and field validation.

Unless you're planning to build websites and applications that do nothing but publish content, and don't accept input from your visitors, you're going to need to understand and use forms.

Django provides a range of tools and libraries to help you build forms to accept input from site visitors, and then process and respond to the input.

HTML forms

In HTML, a form is a collection of elements inside `<form>...</form>` that allow a visitor to do things like enter text, select options, manipulate objects or controls, and so on, and then send that information back to the server.

Some of these form interface elements - text input or checkboxes - are built into HTML itself. Others are much more complex; an interface that pops up a date picker or allows you to move a slider or manipulate controls will typically use JavaScript and CSS as well as HTML form `<input>` elements to achieve these effects.

As well as its `<input>` elements, a form must specify two things:

- *where*: the URL to which the data corresponding to the user's input should be returned
- *how*: the HTTP method the data should be returned by

As an example, the login form for the Django admin contains several `<input>` elements: one of `type="text"` for the username, one of `type="password"` for the password, and one of `type="submit"` for the "Log in" button. It also contains some hidden text fields that the user doesn't see, which Django uses to determine what to do next.

It also tells the browser that the form data should be sent to the URL specified in the `<form>` 's `action` attribute - `/admin/` - and that it should be sent using the HTTP mechanism specified by the `method` attribute - `post` .

When the `<input type="submit" value="Log in">` element is triggered, the data is returned to `/admin/` .

`GET` and `POST`

`GET` and `POST` are the only HTTP methods to use when dealing with forms.

Django's login form is returned using the `POST` method, in which the browser bundles up the form data, encodes it for transmission, sends it to the server, and then receives back its response.

`GET` , by contrast, bundles the submitted data into a string, and uses this to compose a URL. The URL contains the address where the data must be sent, as well as the data keys and values. You can see this in action if you do a search in the Django documentation, which will produce a URL of the form `https://docs.djangoproject.com/search/?q=forms&release=1` .

`GET` and `POST` are typically used for different purposes.

Any request that could be used to change the state of the system - for example, a request that makes changes in the database - should use `POST`. `GET` should be used only for requests that do not affect the state of the system.

`GET` would also be unsuitable for a password form, because the password would appear in the URL, and thus, also in browser history and server logs, all in plain text. Neither would it be suitable for large quantities of data, or for binary data, such as an image. A web application that uses `GET` requests for admin forms is a security risk: it can be easy for an attacker to mimic a form's request to gain access to sensitive parts of the system. `POST`, coupled with other protections like Django's CSRF protection offers more control over access.

On the other hand, `GET` is suitable for things like a web search form, because the URLs that represent a `GET` request can easily be bookmarked, shared, or resubmitted.

Django's role in forms

Handling forms is a complex business. Consider Django's admin, where numerous items of data of several different types may need to be prepared for display in a form, rendered as HTML, edited using a convenient interface, returned to the server, validated and cleaned up, and then saved or passed on for further processing.

Django's form functionality can simplify and automate vast portions of this work, and can also do it more securely than most programmers would be able to do in code they wrote themselves.

Django handles three distinct parts of the work involved in forms:

- preparing and restructuring data to make it ready for rendering
- creating HTML forms for the data
- receiving and processing submitted forms and data from the client

It is *possible* to write code that does all of this manually, but Django can take care of it all for you.

Forms in Django

We've described HTML forms briefly, but an HTML `<form>` is just one part of the machinery required.

In the context of a web application, 'form' might refer to that HTML `<form>`, or to the Django `Form` that produces it, or to the structured data returned when it is submitted, or to the end-to-end working collection of these parts.

The Django class `Form`

At the heart of this system of components is Django's `Form` class. In much the same way that a Django model describes the logical structure of an object, its behavior, and the way its parts are represented to us, a `Form` class describes a form and determines how it works and appears.

In a similar way that a model class's fields map to database fields, a form class's fields map to HTML form `<input>` elements. (A `ModelForm` maps a model class's fields to HTML form `<input>` elements via a `Form`; this is what the Django

admin is based upon.)

A form's fields are themselves classes; they manage form data and perform validation when a form is submitted. A `DateField` and a `FileField` handle very different kinds of data and have to do different things with it.

A form field is represented to a user in the browser as an HTML "widget" - a piece of user interface machinery. Each field type has an appropriate default Widget class, but these can be overridden as required.

Instantiating, processing, and rendering forms

When rendering an object in Django, we generally:

1. get hold of it in the view (fetch it from the database, for example)
2. pass it to the template context
3. expand it to HTML markup using template variables

Rendering a form in a template involves nearly the same work as rendering any other kind of object, but there are some key differences.

In the case of a model instance that contained no data, it would rarely if ever be useful to do anything with it in a template. On the other hand, it makes perfect sense to render an unpopulated form - that's what we do when we want the user to populate it.

So when we handle a model instance in a view, we typically retrieve it from the database. When we're dealing with a form we typically instantiate it in the view.

When we instantiate a form, we can opt to leave it empty or prepopulate it, for example with:

- data from a saved model instance (as in the case of admin forms for editing)
- data that we have collated from other sources
- data received from a previous HTML form submission

The last of these cases is the most interesting, because it's what makes it possible for users not just to read a website, but to send information back to it too.

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.

This is a very simple form. In practice, a form might contain dozens or hundreds of fields, many of which might need to be prepopulated, and we might expect the user to work through the edit-submit cycle several times before concluding the operation.

We might require some validation to occur in the browser, even before the form is submitted; we might want to use much more complex fields, that allow the user to do things like pick dates from a calendar and so on.

At this point it's much easier to get Django to do most of this work for us.

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:

`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>
```

Note that it **does not** include the `<form>` tags, or a submit button. We'll have to provide those ourselves in the template.

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:

`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 `is_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.

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.

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`.

We now have a working web form, described by a Django `Form`, processed by a view, and rendered as an HTML `<form>`.

That's all you need to get started, but the forms framework puts a lot more at your fingertips. Once you understand the basics of the process described above, you should be prepared to understand other features of the forms system and ready to learn a bit more about the underlying machinery.

More about Django Form classes

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:

`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:

`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/")
```

> **Tip:** For more on sending email from Django, see Sending email.

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`). For details of how to handle file uploads with your form, see Binding uploaded files to a form.

Working with form templates

All you need to do to get your form into a template is to place the form instance into the template context. So if your form is called `form` in the context, `{{ form }}` will render its `<label>` and `<input>` elements appropriately.

Additional form template furniture: Don't forget that a form's output does *not* include the surrounding `<form>` tags, or the form's `submit` control. You will have to provide these yourself.

Reusable form templates

The HTML output when rendering a form is itself generated via a template. You can control this by creating an appropriate template file and setting a custom `FORM_RENDERER` to use that `form_template_name` site-wide. You can also customize per-form by overriding the form's `template_name` attribute to render the form using the custom template, or by passing the template name directly to `Form.render()`.

The example below will result in `{{ form }}` being rendered as the output of the `form_snippet.html` template.

In your templates:

```
# In your template:
{{ form }}

# In form_snippet.html:
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

Then you can configure the `FORM_RENDERER` setting:

`settings.py`

```
from django.forms.renderers import TemplatesSetting


class CustomFormRenderer(TemplatesSetting):
    form_template_name = "form_snippet.html"


FORM_RENDERER = "project.settings.CustomFormRenderer"
```

... or for a single form:

```
class MyForm(forms.Form):
    template_name = "form_snippet.html"
```

```
    ...
```

... or for a single render of a form instance, passing in the template name to the `Form.render()`. Here's an example of this being used in a view:

```
def index(request):
    form = MyForm()
    rendered_form = form.render("form_snippet.html")
    context = {"form": rendered_form}
    return render(request, "index.html", context)
```

See Outputting forms as HTML for more details.

Reusable field group templates

New in Django 5.0.

Each field is available as an attribute of the form, using `{{ form.name_of_field }}` in a template. A field has a `as_field_group()` method which renders the related elements of the field as a group, its label, widget, errors, and help text.

This allows generic templates to be written that arrange fields elements in the required layout. For example:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
  {{ form.subject.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.message.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.sender.as_field_group }}
</div>
<div class="fieldWrapper">
  {{ form.cc_myself.as_field_group }}
</div>
```

By default Django uses the `"django/forms/field.html"` template which is designed for use with the default `"django/forms/div.html"` form style.

The default template can be customized by setting `field_template_name` in your project-level `FORM_RENDERER`:

```
from django.forms.renderers import TemplatesSetting


class CustomFormRenderer(TemplatesSetting):
    field_template_name = "field_snippet.html"
```

... or on a single field:

```
class MyForm(forms.Form):
    subject = forms.CharField(template_name="my_custom_template.html")
    ...
```

... or on a per-request basis by calling `BoundField.render()` and supplying a template name:

```
def index(request):
    form = ContactForm()
    subject = form["subject"]
    context = {"subject": subject.render("my_custom_template.html")}
    return render(request, "index.html", context)
```

Rendering fields manually

More fine grained control over field rendering is also possible. Likely this will be in a custom field template, to allow the template to be written once and reused for each field. However, it can also be directly accessed from the field attribute on the form. For example:

```
{{ form.non_field_errors }}
<div class="fieldWrapper">
    {{ form.subject.errors }}
    <label for="{{ form.subject.id_for_label }}">Email subject:</label>
    {{ form.subject }}
</div>
<div class="fieldWrapper">
    {{ form.message.errors }}
    <label for="{{ form.message.id_for_label }}">Your message:</label>
    {{ form.message }}
</div>
<div class="fieldWrapper">
    {{ form.sender.errors }}
    <label for="{{ form.sender.id_for_label }}">Your email address:</label>
    {{ form.sender }}
</div>
<div class="fieldWrapper">
    {{ form.cc_myself.errors }}
    <label for="{{ form.cc_myself.id_for_label }}">CC yourself?</label>
    {{ form.cc_myself }}
</div>
```

Complete `<label>` elements can also be generated using the `label_tag()`. For example:

```
<div class="fieldWrapper">
    {{ form.subject.errors }}
    {{ form.subject.label_tag }}
    {{ form.subject }}
</div>
```

Rendering form error messages

The price of this flexibility is a bit more work. Until now we haven't had to worry about how to display form errors, because that's taken care of for us. In this example we have had to make sure we take care of any errors for each field and any errors for the form as a whole. Note `{{ form.non_field_errors }}` at the top of the form and the template lookup for errors on each field.

Using `{{ form.name_of_field.errors }}` displays a list of form errors, rendered as an unordered list. This might look like:

```
<ul class="errorlist">
    <li>Sender is required.</li>
</ul>
```

The list has a CSS class of `errorlist` to allow you to style its appearance. If you wish to further customize the display of errors you can do so by looping over them:

```
{% if form.subject.errors %}
    <ol>
    {% for error in form.subject.errors %}
        <li><strong>{{ error|escape }}</strong></li>
    {% endfor %}
    </ol>
{% endif %}
```

Non-field errors (and/or hidden field errors that are rendered at the top of the form when using helpers like `form.as_p()`) will be rendered with an additional class of `nonfield` to help distinguish them from field-specific errors. For example, `{{ form.non_field_errors }}` would look like:

```
<ul class="errorlist nonfield">
    <li>Generic validation error</li>
</ul>
```

See The Forms API for more on errors, styling, and working with form attributes in templates.

Looping over the form's fields

If you're using the same HTML for each of your form fields, you can reduce duplicate code by looping through each field in turn using a `{% for %}` loop:

```
{% for field in form %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
        {% if field.help_text %}
          <p class="help" id="{{ field.auto_id }}_helptext">
```

```
            {{ field.help_text|safe }}
          </p>
        {% endif %}
    </div>
{% endfor %}
```

Useful attributes on `{{ field }}` include:

`{{ field.errors }}`

Outputs a `<ul class="errorlist">` containing any validation errors corresponding to this field. You can customize the presentation of the errors with a `{% for error in field.errors %}` loop. In this case, each object in the loop is a string containing the error message.

`{{ field.field }}`

The `Field` instance from the form class that this `BoundField` wraps. You can use it to access `Field` attributes, e.g. `{{ char_field.field.max_length }}`.

`{{ field.help_text }}`

Any help text that has been associated with the field.

`{{ field.html_name }}`

The name of the field that will be used in the input element's name field. This takes the form prefix into account, if it has been set.

`{{ field.id_for_label }}`

The ID that will be used for this field (`id_email` in the example above). If you are constructing the label manually, you may want to use this in lieu of `label_tag`. It's also useful, for example, if you have some inline JavaScript and want to avoid hardcoding the field's ID.

`{{ field.is_hidden }}`

This attribute is `True` if the form field is a hidden field and `False` otherwise. It's not particularly useful as a template variable, but could be useful in conditional tests such as:

```
{% if field.is_hidden %}
   {# Do something special #}
{% endif %}
```

`{{ field.label }}`

The label of the field, e.g. `Email address`.

`{{ field.label_tag }}`

The field's label wrapped in the appropriate HTML `<label>` tag. This includes the form's `label_suffix`. For example, the default `label_suffix` is a colon:

```
<label for="id_email">Email address:</label>
```

`{{ field.legend_tag }}`

Similar to `field.label_tag` but uses a `<legend>` tag in place of `<label>`, for widgets with multiple inputs wrapped in a `<fieldset>`.

`{{ field.use_fieldset }}`

This attribute is `True` if the form field's widget contains multiple inputs that should be semantically grouped in a `<fieldset>` with a `<legend>` to improve accessibility. An example use in a template:

```
{% if field.use_fieldset %}
  <fieldset>
  {% if field.label %}{{ field.legend_tag }}{% endif %}
{% else %}
  {% if field.label %}{{ field.label_tag }}{% endif %}
{% endif %}
{{ field }}
{% if field.use_fieldset %}</fieldset>{% endif %}
```

`{{ field.value }}`

The value of the field. e.g `someone@example.com`.

See also: For a complete list of attributes and methods, see `BoundField`.

Looping over hidden and visible fields

If you're manually laying out a form in a template, as opposed to relying on Django's default form layout, you might want to treat `<input type="hidden">` fields differently from non-hidden fields. For example, because hidden fields don't display anything, putting error messages "next to" the field could cause confusion for your users – so errors for those fields should be handled differently.

Django provides two methods on a form that allow you to loop over the hidden and visible fields independently: `hidden_fields()` and `visible_fields()`. Here's a modification of an earlier example that uses these two methods:

```
{# Include the hidden fields #}
{% for hidden in form.hidden_fields %}
{{ hidden }}
{% endfor %}
{# Include the visible fields #}
{% for field in form.visible_fields %}
    <div class="fieldWrapper">
        {{ field.errors }}
        {{ field.label_tag }} {{ field }}
    </div>
{% endfor %}
```

This example does not handle any errors in the hidden fields. Usually, an error in a hidden field is a sign of form tampering, since normal form interaction won't alter them. However, you could easily insert some error displays for those form errors, as well.