Chapter 7: Forms

HTML forms are the backbone of interactive Web sites, from the simplicity of Google's single search box to ubiquitous blog comment-submission forms to complex custom data-entry interfaces. This chapter covers how you can use Django to access user-submitted form data, validate it and do something with it. Along the way, we'll cover `HttpRequest` and `Form` objects.

Getting Data From the Request Object

We introduced `HttpRequest` objects in Chapter 3 when we first covered view functions, but we didn't have much to say about them at the time. Recall that each view function takes an `HttpRequest` object as its first parameter, as in our `hello()` view:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello world")
```

`HttpRequest` objects, such as the variable `request` here, have a number of interesting attributes and methods that you should familiarize yourself with, so that you know what's possible. You can use these attributes to get information about the current request (i.e., the user/Web browser that's loading the current page on your Django-powered site), at the time the view function is executed.

Information About the URL

`HttpRequest` objects contain several pieces of information about the currently requested URL:

Attribute/method	Description	Example
`request.path`	The full path, not including the domain but including the leading slash.	`"/hello/"`
`request.get_host()`	The host (i.e., the "domain," in common parlance).	`"127.0.0.1:8000"` or `"www.example.com"`
`request.get_full_path()`	The `path`, plus a query string (if available).	`"/hello/?print=true"`
`request.is_secure()`	`True` if the request was made via HTTPS. Otherwise, `False`.	`True` or `False`

Always use these attributes/methods instead of hard-coding URLs in your views. This makes for more flexible code that can be reused in other places. A simplistic example:

```
# BAD!
def current_url_view_bad(request):
    return HttpResponse("Welcome to the page at /current/")

# GOOD
```

```
def current_url_view_good(request):
    return HttpResponse("Welcome to the page at %s" % request.path)
```

Other Information About the Request

`request.META` is a Python dictionary containing all available HTTP headers for the given request – including the user's IP address and user agent (generally the name and version of the Web browser). Note that the full list of available headers depends on which headers the user sent and which headers your Web server sets. Some commonly available keys in this dictionary are:

- `HTTP_REFERER` – The referring URL, if any. (Note the misspelling of `REFERER`.)
- `HTTP_USER_AGENT` – The user's browser's user-agent string, if any. This looks something like: `"Mozilla/5.0 (X11; U; Linux i686; fr-FR; rv:1.8.1.17) Gecko/20080829 Firefox/2.0.0.17"`.
- `REMOTE_ADDR` – The IP address of the client, e.g., `"12.345.67.89"`. (If the request has passed through any proxies, then this might be a comma-separated list of IP addresses, e.g., `"12.345.67.89,23.456.78.90"`.)

Note that because `request.META` is just a basic Python dictionary, you'll get a `KeyError` exception if you try to access a key that doesn't exist. (Because HTTP headers are *external* data – that is, they're submitted by your users' browsers – they shouldn't be trusted, and you should always design your application to fail gracefully if a particular header is empty or doesn't exist.) You should either use a `try`/`except` clause or the `get()` method to handle the case of undefined keys:

```
# BAD!
def ua_display_bad(request):
    ua = request.META['HTTP_USER_AGENT']  # Might raise KeyError!
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 1)
def ua_display_good1(request):
    try:
        ua = request.META['HTTP_USER_AGENT']
    except KeyError:
        ua = 'unknown'
    return HttpResponse("Your browser is %s" % ua)

# GOOD (VERSION 2)
def ua_display_good2(request):
    ua = request.META.get('HTTP_USER_AGENT', 'unknown')
    return HttpResponse("Your browser is %s" % ua)
```

We encourage you to write a small view that displays all of the `request.META` data so you can get to know what's in there. Here's what that view might look like:

```
def display_meta(request):
    values = request.META.items()
    values.sort()
    html = []
    for k, v in values:
        html.append('<tr><td>%s</td><td>%s</td></tr>' % (k, v))
    return HttpResponse('<table>%s</table>' % '\n'.join(html))
```

As an exercise, see whether you can convert this view to use Django's template system instead of hard-coding the HTML. Also try adding `request.path` and the other `HttpRequest` methods from the previous section.

Information About Submitted Data

Beyond basic metadata about the request, `HttpRequest` objects have two attributes that contain information submitted by the user: `request.GET` and `request.POST`. Both of these are dictionary-like objects that give you access to `GET` and `POST` data.

> **Dictionary-like objects**
>
> When we say `request.GET` and `request.POST` are "dictionary-like" objects, we mean that they behave like standard Python dictionaries but aren't technically dictionaries under the hood. For example, `request.GET` and `request.POST` both have `get()`, `keys()` and `values()` methods, and you can iterate over the keys by doing `for key in request.GET`.
>
> So why the distinction? Because both `request.GET` and `request.POST` have additional methods that normal dictionaries don't have. We'll get into these in a short while.
>
> You might have encountered the similar term "file-like objects" – Python objects that have a few basic methods, like `read()`, that let them act as stand-ins for "real" file objects.

`POST` data generally is submitted from an HTML `<form>`, while `GET` data can come from a `<form>` or the query string in the page's URL.

A Simple Form-Handling Example

Continuing this book's ongoing example of books, authors and publishers, let's create a simple view that lets users search our book database by title.

Generally, there are two parts to developing a form: the HTML user interface and the backend view code that processes the submitted data. The first part is easy; let's just set up a view that displays a search form:

```
from django.shortcuts import render

def search_form(request):
    return render(request, 'search_form.html')
```

As we learned in Chapter 3, this view can live anywhere on your Python path. For sake of argument, put it in `books/views.py`.

The accompanying template, `search_form.html`, could look like this:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

The URLpattern in `urls.py` could look like this:

```
from mysite.books import views
```

```
urlpatterns = patterns('',
    # ...
    url(r'^search-form/$', views.search_form),
    # ...
)
```

(Note that we're importing the `views` module directly, instead of something like `from mysite.views import search_form`, because the former is less verbose. We'll cover this importing approach in more detail in Chapter 8.)

Now, if you run the `runserver` and visit `http://127.0.0.1:8000/search-form/`, you'll see the search interface. Simple enough.

Try submitting the form, though, and you'll get a Django 404 error. The form points to the URL `/search/`, which hasn't yet been implemented. Let's fix that with a second view function:

```
# urls.py

urlpatterns = patterns('',
    # ...
    (r'^search-form/$', views.search_form),
    (r'^search/$', views.search),
    # ...
)

# views.py

def search(request):
    if 'q' in request.GET:
        message = 'You searched for: %r' % request.GET['q']
    else:
        message = 'You submitted an empty form.'
    return HttpResponse(message)
```

For the moment, this merely displays the user's search term, so we can make sure the data is being submitted to Django properly, and so you can get a feel for how the search term flows through the system. In short:

1. The HTML `<form>` defines a variable `q`. When it's submitted, the value of `q` is sent via `GET` (`method="get"`) to the URL `/search/`.
2. The Django view that handles the URL `/search/` (`search()`) has access to the `q` value in `request.GET`.

An important thing to point out here is that we explicitly check that `'q'` exists in `request.GET`. As we pointed out in the `request.META` section above, you shouldn't trust anything submitted by users or even assume that they've submitted anything in the first place. If we didn't add this check, any submission of an empty form would raise `KeyError` in the view:

```
# BAD!
def bad_search(request):
    # The following line will raise KeyError if 'q' hasn't
    # been submitted!
    message = 'You searched for: %r' % request.GET['q']
    return HttpResponse(message)
```

Query string parameters

Because `GET` data is passed in the query string (e.g., `/search/?q=django`), you can use `request.GET` to access query string variables. In Chapter 3's introduction of Django's URLconf system, we compared Django's pretty URLs to more traditional PHP/Java URLs such as `/time/plus?hours=3` and said we'd show you how to do the latter in Chapter 7. Now you know how to access query string parameters in your views (like `hours=3` in this example) – use `request.GET`.

`POST` data works the same way as `GET` data – just use `request.POST` instead of `request.GET`. What's the difference between `GET` and `POST`? Use `GET` when the act of submitting the form is just a request to "get" data. Use `POST` whenever the act of submitting the form will have some side effect – *changing* data, or sending an e-mail, or something else that's beyond simple *display* of data. In our book-search example, we're using `GET` because the query doesn't change any data on our server. (See http://www.w3.org/2001/tag/doc/whenToUseGet.html if you want to learn more about `GET` and `POST`.)

Now that we've verified `request.GET` is being passed in properly, let's hook the user's search query into our book database (again, in `views.py`):

```
from django.http import HttpResponse
from django.shortcuts import render
from mysite.books.models import Book

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
            {'books': books, 'query': q})
    else:
        return HttpResponse('Please submit a search term.')
```

A couple of notes on what we did here:

- Aside from checking that `'q'` exists in `request.GET`, we also make sure that `request.GET['q']` is a non-empty value before passing it to the database query.
- We're using `Book.objects.filter(title__icontains=q)` to query our book table for all books whose title includes the given submission. The `icontains` is a lookup type (as explained in Chapter 5 and Appendix B), and the statement can be roughly translated as "Get the books whose title contains `q`, without being case-sensitive."

  This is a very simple way to do a book search. We wouldn't recommend using a simple `icontains` query on a large production database, as it can be slow. (In the real world, you'd want to use a custom search system of some sort. Search the Web for *open-source full-text search* to get an idea of the possibilities.)
- We pass `books`, a list of `Book` objects, to the template. The template code for `search_results.html` might include something like this:

  ```
  <p>You searched for: <strong>{{ query }}</strong></p>

  {% if books %}
      <p>Found {{ books|length }} book{{ books|pluralize }}.</p>
      <ul>
          {% for book in books %}
  ```

```
            <li>{{ book.title }}</li>
            {% endfor %}
        </ul>
    {% else %}
        <p>No books matched your search criteria.</p>
    {% endif %}
```

Note usage of the `pluralize` template filter, which outputs an "s" if appropriate, based on the number of books found.

Improving Our Simple Form-Handling Example

As in previous chapters, we've shown you the simplest thing that could possibly work. Now we'll point out some problems and show you how to improve it.

First, our `search()` view's handling of an empty query is poor – we're just displaying a `"Please submit a search term."` message, requiring the user to hit the browser's back button. This is horrid and unprofessional, and if you ever actually implement something like this in the wild, your Django privileges will be revoked.

It would be much better to redisplay the form, with an error above it, so that the user can try again immediately. The easiest way to do that would be to render the template again, like this:

```
from django.http import HttpResponse
from django.shortcuts import render
from mysite.books.models import Book

def search_form(request):
    return render(request, 'search_form.html')

def search(request):
    if 'q' in request.GET and request.GET['q']:
        q = request.GET['q']
        books = Book.objects.filter(title__icontains=q)
        return render(request, 'search_results.html',
            {'books': books, 'query': q})
    else:
        return render(request, 'search_form.html', {'error': True})
```

(Note that we've included `search_form()` here so you can see both views in one place.)

Here, we've improved `search()` to render the `search_form.html` template again, if the query is empty. And because we need to display an error message in that template, we pass a template variable. Now we can edit `search_form.html` to check for the `error` variable:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term.</p>
    {% endif %}
```

```
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

We can still use this template from our original view, `search_form()`, because `search_form()` doesn't pass `error` to the template – so the error message won't show up in that case.

With this change in place, it's a better application, but it now begs the question: is a dedicated `search_form()` view really necessary? As it stands, a request to the URL `/search/` (without any `GET` parameters) will display the empty form (but with an error). We can remove the `search_form()` view, along with its associated URLpattern, as long as we change `search()` to hide the error message when somebody visits `/search/` with no `GET` parameters:

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                {'books': books, 'query': q})
    return render(request, 'search_form.html',
        {'error': error})
```

In this updated view, if a user visits `/search/` with no `GET` parameters, he'll see the search form with no error message. If a user submits the form with an empty value for `'q'`, he'll see the search form *with* an error message. And, finally, if a user submits the form with a non-empty value for `'q'`, he'll see the search results.

We can make one final improvement to this application, to remove a bit of redundancy. Now that we've rolled the two views and URLs into one and `/search/` handles both search-form display and result display, the HTML `<form>` in `search_form.html` doesn't have to hard-code a URL. Instead of this:

```
<form action="/search/" method="get">
```

It can be changed to this:

```
<form action="" method="get">
```

The `action=""` means "Submit the form to the same URL as the current page." With this change in place, you won't have to remember to change the `action` if you ever hook the `search()` view to another URL.

Simple validation

Our search example is still reasonably simple, particularly in terms of its data validation; we're merely checking to make sure the search query isn't empty. Many HTML forms include a level of validation that's more complex than making sure the value is non-empty. We've all seen the error messages on Web sites:

- "Please enter a valid e-mail address. 'foo' is not an e-mail address."
- "Please enter a valid five-digit U.S. ZIP code. '123' is not a ZIP code."

- "Please enter a valid date in the format YYYY-MM-DD."
- "Please enter a password that is at least 8 characters long and contains at least one number."

> **A note on JavaScript validation**
>
> This is beyond the scope of this book, but you can use JavaScript to validate data on the client side, directly in the browser. But be warned: even if you do this, you *must* validate data on the server side, too. Some people have JavaScript turned off, and some malicious users might submit raw, unvalidated data directly to your form handler to see whether they can cause mischief.
>
> There's nothing you can do about this, other than *always* validate user-submitted data server-side (i.e., in your Django views). You should think of JavaScript validation as a bonus usability feature, not as your only means of validating.

Let's tweak our `search()` view so that it validates that the search term is less than or equal to 20 characters long. (For sake of example, let's say anything longer than that might make the query too slow.) How might we do that? The simplest possible thing would be to embed the logic directly in the view, like this:

```
def search(request):
    error = False
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            error = True
        elif len(q) > 20:
            error = True
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                {'books': books, 'query': q})
    return render(request, 'search_form.html',
        {'error': error})
```

Now, if you try submitting a search query greater than 20 characters long, it won't let you search; you'll get an error message. But that error message in `search_form.html` currently says `"Please submit a search term."` – so we'll have to change it to be accurate for both cases:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if error %}
        <p style="color: red;">Please submit a search term 20 characters or shorter.</p>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

There's something ugly about this. Our one-size-fits-all error message is potentially confusing. Why should the

error message for an empty form submission mention anything about a 20-character limit? Error messages should be specific, unambiguous and not confusing.

The problem is in the fact that we're using a simple boolean value for `error`, whereas we should be using a *list* of error message strings. Here's how we might fix that:

```
def search(request):
    errors = []
    if 'q' in request.GET:
        q = request.GET['q']
        if not q:
            errors.append('Enter a search term.')
        elif len(q) > 20:
            errors.append('Please enter at most 20 characters.')
        else:
            books = Book.objects.filter(title__icontains=q)
            return render(request, 'search_results.html',
                {'books': books, 'query': q})
    return render(request, 'search_form.html',
        {'errors': errors})
```

Then, we need make a small tweak to the `search_form.html` template to reflect that it's now passed an `errors` list instead of an `error` boolean value:

```
<html>
<head>
    <title>Search</title>
</head>
<body>
    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}
    <form action="/search/" method="get">
        <input type="text" name="q">
        <input type="submit" value="Search">
    </form>
</body>
</html>
```

Making a Contact Form

Although we iterated over the book search form example several times and improved it nicely, it's still fundamentally simple: just a single field, `'q'`. Because it's so simple, we didn't even use Django's form library to deal with it. But more complex forms call for more complex treatment – and now we'll develop something more complex: a site contact form.

This will be a form that lets site users submit a bit of feedback, along with an optional e-mail return address. After the form is submitted and the data is validated, we'll automatically send the message via e-mail to the site staff.

We'll start with our template, `contact_form.html`.

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form action="/contact/" method="post">
        <p>Subject: <input type="text" name="subject"></p>
        <p>Your e-mail (optional): <input type="text" name="email"></p>
        <p>Message: <textarea name="message" rows="10" cols="50"></textarea></p>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

We've defined three fields: the subject, e-mail address and message. The second is optional, but the other two fields are required. Note we're using `method="post"` here instead of `method="get"` because this form submission has a side effect – it sends an e-mail. Also, we copied the error-displaying code from our previous template `search_form.html`.

If we continue down the road established by our `search()` view from the previous section, a naive version of our `contact()` view might look like this:

```
from django.core.mail import send_mail
from django.http import HttpResponseRedirect
from django.shortcuts import render

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('email', 'noreply@example.com'),
```

```
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render(request, 'contact_form.html',
        {'errors': errors})
```

(If you're following along, you may be wondering whether to put this view in the `books/views.py` file. It doesn't have anything to do with the books application, so should it live elsewhere? It's totally up to you; Django doesn't care, as long as you're able to point to the view from your URLconf. Our personal preference would be to create a separate directory, `contact`, at the same level in the directory tree as `books`. This would contain an empty `__init__.py` and `views.py`.)

A couple of new things are happening here:

- We're checking that `request.method` is `'POST'`. This will only be true in the case of a form submission; it won't be true if somebody is merely viewing the contact form. (In the latter case, `request.method will be set to 'GET'`, because in normal Web browsing, browsers use `GET`, not `POST`.) This makes it a nice way to isolate the "form display" case from the "form processing" case.

- Instead of `request.GET`, we're using `request.POST` to access the submitted form data. This is necessary because the HTML `<form>` in `contact_form.html` uses `method="post"`. If this view is accessed via `POST`, then `request.GET` will be empty.

- This time, we have *two* required fields, `subject` and `message`, so we have to validate both. Note that we're using `request.POST.get()` and providing a blank string as the default value; this is a nice, short way of handling both the cases of missing keys and missing data.

- Although the `email` field is not required, we still validate it if it is indeed submitted. Our validation algorithm here is fragile – we're just checking that the string contains an `@` character. In the real world, you'd want more robust validation (and Django provides it, which we'll show you very shortly).

- We're using the function `django.core.mail.send_mail` to send an e-mail. This function has four required arguments: the e-mail subject, the e-mail body, the "from" address, and a list of recipient addresses. `send_mail` is a convenient wrapper around Django's `EmailMessage` class, which provides advanced features such as attachments, multipart e-mails, and full control over e-mail headers.

  Note that in order to send e-mail using `send_mail()`, your server must be configured to send mail, and Django must be told about your outbound e-mail server. See http://docs.djangoproject.com/en/dev/topics/email/ for the specifics.

- After the e-mail is sent, we redirect to a "success" page by returning an `HttpResponseRedirect` object. We'll leave the implementation of that "success" page up to you (it's a simple view/URLconf/template), but we should explain why we initiate a redirect instead of, for example, simply calling `render()` with a template right there.

  The reason: if a user hits "Refresh" on a page that was loaded via `POST`, that request will be repeated. This can often lead to undesired behavior, such as a duplicate record being added to the database – or, in our example, the e-mail being sent twice. If the user is redirected to another page after the `POST`, then there's no chance of repeating the request.

  You should *always* issue a redirect for successful `POST` requests. It's a Web development best practice.

This view works, but those validation functions are kind of crufty. Imagine processing a form with a dozen fields; would you really want to have to write all of those `if` statements?

Another problem is *form redisplay*. In the case of validation errors, it's best practice to redisplay the form *with* the previously submitted data already filled in, so the user can see what he did wrong (and also so the user doesn't

have to reenter data in fields that were submitted correctly). We *could* manually pass the `POST` data back to the template, but we'd have to edit each HTML field to insert the proper value in the proper place:

```
# views.py

def contact(request):
    errors = []
    if request.method == 'POST':
        if not request.POST.get('subject', ''):
            errors.append('Enter a subject.')
        if not request.POST.get('message', ''):
            errors.append('Enter a message.')
        if request.POST.get('email') and '@' not in request.POST['email']:
            errors.append('Enter a valid e-mail address.')
        if not errors:
            send_mail(
                request.POST['subject'],
                request.POST['message'],
                request.POST.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    return render(request, 'contact_form.html', {
        'errors': errors,
        'subject': request.POST.get('subject', ''),
        'message': request.POST.get('message', ''),
        'email': request.POST.get('email', ''),
    })

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if errors %}
        <ul>
            {% for error in errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    {% endif %}

    <form action="/contact/" method="post">
        <p>Subject: <input type="text" name="subject" value="{{ subject }}"></p>
        <p>Your e-mail (optional): <input type="text" name="email" value="{{ email }}"></p>
        <p>Message: <textarea name="message" rows="10" cols="50">**{{ message }}**</textarea></p
        <input type="submit" value="Submit">
    </form>
```

```
</body>
</html>
```

This is a lot of cruft, and it introduces a lot of opportunities for human error. We hope you're starting to see the opportunity for some higher-level library that handles form- and validation-related tasks.

Your First Form Class

Django comes with a form library, called `django.forms`, that handles many of the issues we've been exploring this chapter – from HTML form display to validation. Let's dive in and rework our contact form application using the Django forms framework.

> **Django's "newforms" library**
>
> Throughout the Django community, you might see chatter about something called `django.newforms`. When people speak of `django.newforms`, they're talking about what is now `django.forms` – the library covered by this chapter.
>
> The reason for this name change is historic. When Django was first released to the public, it had a complicated, confusing forms system, `django.forms`. It was completely rewritten, and the new version was called `django.newforms` so that people could still use the old system. When Django 1.0 was released, the old `django.forms` went away, and `django.newforms` became `django.forms`.

The primary way to use the forms framework is to define a `Form` class for each HTML `<form>` you're dealing with. In our case, we only have one `<form>`, so we'll have one `Form` class. This class can live anywhere you want – including directly in your `views.py` file – but community convention is to keep `Form` classes in a separate file called `forms.py`. Create this file in the same directory as your `views.py`, and enter the following:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
    message = forms.CharField()
```

This is pretty intuitive, and it's similar to Django's model syntax. Each field in the form is represented by a type of `Field` class – `CharField` and `EmailField` are the only types of fields used here – as attributes of a `Form` class. Each field is required by default, so to make `email` optional, we specify `required=False`.

Let's hop into the Python interactive interpreter and see what this class can do. The first thing it can do is display itself as HTML:

```
>>> from contact.forms import ContactForm
>>> f = ContactForm()
>>> print f
<tr><th><label for="id_subject">Subject:</label></th><td><input type="text" name="subject" id="i
<tr><th><label for="id_email">Email:</label></th><td><input type="text" name="email" id="id_emai
<tr><th><label for="id_message">Message:</label></th><td><input type="text" name="message" id="i
```

Django adds a label to each field, along with `<label>` tags for accessibility. The idea is to make the default behavior as optimal as possible.

This default output is in the format of an HTML `<table>`, but there are a few other built-in outputs:

```
>>> print f.as_ul()
<li><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subject" />
<li><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" /></li>
<li><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" />
>>> print f.as_p()
<p><label for="id_subject">Subject:</label> <input type="text" name="subject" id="id_subject" />
<p><label for="id_email">Email:</label> <input type="text" name="email" id="id_email" /></p>
<p><label for="id_message">Message:</label> <input type="text" name="message" id="id_message" />
```

Note that the opening and closing `<table>`, `<ul>` and `<form>` tags aren't included in the output, so that you can add any additional rows and customization if necessary.

These methods are just shortcuts for the common case of "display the entire form." You can also display the HTML for a particular field:

```
>>> print f['subject']
<input type="text" name="subject" id="id_subject" />
>>> print f['message']
<input type="text" name="message" id="id_message" />
```

The second thing `Form` objects can do is validate data. To validate data, create a new `Form` object and pass it a dictionary of data that maps field names to data:

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com', 'message': 'Nice site!'}
```

Once you've associated data with a `Form` instance, you've created a "bound" form:

```
>>> f.is_bound
True
```

Call the `is_valid()` method on any bound `Form` to find out whether its data is valid. We've passed a valid value for each field, so the `Form` in its entirety is valid:

```
>>> f.is_valid()
True
```

If we don't pass the `email` field, it's still valid, because we've specified `required=False` for that field:

```
>>> f = ContactForm({'subject': 'Hello', 'message': 'Nice site!'})
>>> f.is_valid()
True
```

But, if we leave off either `subject` or `message`, the `Form` is no longer valid:

```
>>> f = ContactForm({'subject': 'Hello'})
>>> f.is_valid()
False
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.is_valid()
False
```

You can drill down to get field-specific error messages:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f['message'].errors
[u'This field is required.']
>>> f['subject'].errors
[]
>>> f['email'].errors
[]
```

Each bound `Form` instance has an `errors` attribute that gives you a dictionary mapping field names to error-message lists:

```
>>> f = ContactForm({'subject': 'Hello', 'message': ''})
>>> f.errors
{'message': [u'This field is required.']}
```

Finally, for `Form` instances whose data has been found to be valid, a `cleaned_data` attribute is available. This is a dictionary of the submitted data, "cleaned up." Django's forms framework not only validates data, it cleans it up by converting values to the appropriate Python types.

```
>>> f = ContactForm({'subject': 'Hello', 'email': 'adrian@example.com', 'message': 'Nice site!'}
>>> f.is_valid()
True
>>> f.cleaned_data
{'message': u'Nice site!', 'email': u'adrian@example.com', 'subject': u'Hello'}
```

Our contact form only deals with strings, which are "cleaned" into Unicode objects – but if we were to use an `IntegerField` or `DateField`, the forms framework would ensure that `cleaned_data` used proper Python integers or `datetime.date` objects for the given fields.

Tying Form Objects Into Views

With some basic knowledge about `Form` classes, you might see how we can use this infrastructure to replace some of the cruft in our `contact()` view. Here's how we can rewrite `contact()` to use the forms framework:

```
# views.py

from django.shortcuts import render
from mysite.contact.forms import ContactForm
from django.http import HttpResponseRedirect
from django.core.mail import send_mail

def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
```

```
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm()
    return render(request, 'contact_form.html', {'form': form})

# contact_form.html

<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <table>
            {{ form.as_table }}
        </table>
        {% csrf_token %}
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Look at how much cruft we've been able to remove! Django's forms framework handles the HTML display, the validation, data cleanup and form redisplay-with-errors.

Since we're creating a POST form (which can have the effect of modifying data), we need to worry about Cross Site Request Forgeries. Thankfully, you don't have to worry too hard, because Django comes with a very easy-to-use system for protecting against it. In short, all POST forms that are targeted at internal URLs should use the `{% csrf_token %}` template tag. More details about `{% csrf_token %}` can be found in *Chapter 16: django.contrib* and *Chapter 20: Security*.

Try running this locally. Load the form, submit it with none of the fields filled out, submit it with an invalid e-mail address, then finally submit it with valid data. (Of course, depending on your mail-server configuration, you might get an error when `send_mail()` is called, but that's another issue.)

Changing How Fields Are Rendered

Probably the first thing you'll notice when you render this form locally is that the `message` field is displayed as an `<input type="text">`, and it ought to be a `<textarea>`. We can fix that by setting the field's *widget*:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField()
    email = forms.EmailField(required=False)
```

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

The forms framework separates out the presentation logic for each field into a set of widgets. Each field type has a default widget, but you can easily override the default, or provide a custom widget of your own.

Think of the `Field` classes as representing *validation logic*, while widgets represent *presentation logic*.

Setting a Maximum Length

One of the most common validation needs is to check that a field is of a certain size. For good measure, we should improve our `ContactForm` to limit the `subject` to 100 characters. To do that, just supply a `max_length` to the `CharField`, like this:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)
```

An optional `min_length` argument is also available.

Setting Initial Values

As an improvement to this form, let's add an *initial value* for the `subject` field: `"I love your site!"` (A little power of suggestion can't hurt.) To do this, we can use the `initial` argument when we create a `Form` instance:

```
def contact(request):
    if request.method == 'POST':
        form = ContactForm(request.POST)
        if form.is_valid():
            cd = form.cleaned_data
            send_mail(
                cd['subject'],
                cd['message'],
                cd.get('email', 'noreply@example.com'),
                ['siteowner@example.com'],
            )
            return HttpResponseRedirect('/contact/thanks/')
    else:
        form = ContactForm(
            initial={'subject': 'I love your site!'}
        )
    return render(request, 'contact_form.html', {'form': form})
```

Now, the `subject` field will be displayed prepopulated with that kind statement.

Note that there is a difference between passing *initial* data and passing data that *binds* the form. The biggest difference is that if you're just passing *initial* data, then the form will be *unbound*, which means it won't have any error messages.

Custom Validation Rules

Imagine we've launched our feedback form, and the e-mails have started tumbling in. There's just one problem: some of the submitted messages are just one or two words, which isn't long enough for us to make sense of. We decide to adopt a new validation policy: four words or more, please.

There are a number of ways to hook custom validation into a Django form. If our rule is something we will reuse again and again, we can create a custom field type. Most custom validations are one-off affairs, though, and can be tied directly to the `Form` class.

We want additional validation on the `message` field, so we add a `clean_message()` method to our `Form` class:

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False)
    message = forms.CharField(widget=forms.Textarea)

    def clean_message(self):
        message = self.cleaned_data['message']
        num_words = len(message.split())
        if num_words < 4:
            raise forms.ValidationError("Not enough words!")
        return message
```

Django's form system automatically looks for any method whose name starts with `clean_` and ends with the name of a field. If any such method exists, it's called during validation.

Specifically, the `clean_message()` method will be called *after* the default validation logic for a given field (in this case, the validation logic for a required `CharField`). Because the field data has already been partially processed, we pull it out of `self.cleaned_data`. Also, we don't have to worry about checking that the value exists and is non-empty; that's done by the default validator.

We naively use a combination of `len()` and `split()` to count the number of words. If the user has entered too few words, we raise a `forms.ValidationError`. The string attached to this exception will be displayed to the user as an item in the error list.

It's important that we explicitly return the cleaned value for the field at the end of the method. This allows us to modify the value (or convert it to a different Python type) within our custom validation method. If we forget the return statement, then `None` will be returned, and the original value will be lost.

Specifying labels

By default, the labels on Django's auto-generated form HTML are created by replacing underscores with spaces and capitalizing the first letter – so the label for the `email` field is `"Email"`. (Sound familiar? It's the same simple algorithm that Django's models use to calculate default `verbose_name` values for fields. We covered this in Chapter 5.)

But, as with Django's models, we can customize the label for a given field. Just use `label`, like so:

```
class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    email = forms.EmailField(required=False, label='Your e-mail address')
    message = forms.CharField(widget=forms.Textarea)
```

Customizing Form Design

Our `contact_form.html` template uses `{{ form.as_table }}` to display the form, but we can display the form in other ways to get more granular control over display.

The quickest way to customize forms' presentation is with CSS. Error lists, in particular, could do with some visual enhancement, and the auto-generated error lists use `<ul class="errorlist">` precisely so that you can target them with CSS. The following CSS really makes our errors stand out:

```
<style type="text/css">
    ul.errorlist {
        margin: 0;
        padding: 0;
    }
    .errorlist li {
        background-color: red;
        color: white;
        display: block;
        font-size: 10px;
        margin: 0 0 3px;
        padding: 4px 5px;
    }
</style>
```

While it's convenient to have our form's HTML generated for us, in many cases you'll want to override the default rendering. `{{ form.as_table }}` and friends are useful shortcuts while you develop your application, but everything about the way a form is displayed can be overridden, mostly within the template itself, and you'll probably find yourself doing this.

Each field's widget (`<input type="text">`, `<select>`, `<textarea>`, etc.) can be rendered individually by accessing `{{ form.fieldname }}` in the template, and any errors associated with a field are available as `{{ form.fieldname.errors }}`. With this in mind, we can construct a custom template for our contact form with the following template code:

```
<html>
<head>
    <title>Contact us</title>
</head>
<body>
    <h1>Contact us</h1>

    {% if form.errors %}
        <p style="color: red;">
            Please correct the error{{ form.errors|pluralize }} below.
        </p>
    {% endif %}

    <form action="" method="post">
        <div class="field">
            {{ form.subject.errors }}
            <label for="id_subject">Subject:</label>
            {{ form.subject }}
        </div>
```

```
        <div class="field">
            {{ form.email.errors }}
            <label for="id_email">Your e-mail address:</label>
            {{ form.email }}
        </div>
        <div class="field">
            {{ form.message.errors }}
            <label for="id_message">Message:</label>
            {{ form.message }}
        </div>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

`{{ form.message.errors }}` displays a `<ul class="errorlist">` if errors are present and a blank string if the field is valid (or the form is unbound). We can also treat `form.message.errors` as a Boolean or even iterate over it as a list. For example:

```
<div class="field{% if form.message.errors %} errors{% endif %}">
    {% if form.message.errors %}
        <ul>
        {% for error in form.message.errors %}
            <li><strong>{{ error }}</strong></li>
        {% endfor %}
        </ul>
    {% endif %}
    <label for="id_message">Message:</label>
    {{ form.message }}
</div>
```

In the case of validation errors, this will add an "errors" class to the containing `<div>` and display the list of errors in an unordered list.

What's Next?

This chapter concludes the introductory material in this book – the so-called "core curriculum." The next section of the book, Chapters 8 to 12, goes into more detail about advanced Django usage, including how to deploy a Django application (Chapter 12).

After these first seven chapters, you should know enough to start writing your own Django projects. The rest of the material in this book will help fill in the missing pieces as you need them.

We'll start in Chapter 8, by doubling back and taking a closer look at views and URLconfs (introduced first in chapter03).

« previous ◇ table of contents ◇ next »