CHAPTER 3

Customizing the Simple CMS

The simple CMS you put together in the last chapter is already in pretty good shape; it's something that most developers wouldn't mind showing to clients as an initial prototype, for example. But so far, it uses just a few stock applications bundled with Django and doesn't offer any extra features on top of that. In this chapter, you'll see how to take this simple project as a foundation and start adding your own customizations, like rich-text editing in the admin and a search system for quickly finding particular pages.

Adding Rich-Text Editing

The default administrative interface Django provides for the `flatpages` application is already production-quality. Many Django-based sites already use it as is to provide an easy way to manage the occasional simple "About page" or to handle similar tasks. But you might want to make the web-based administrative interface just a little bit friendlier by adding a rich-text interface to it so that users don't have to type in raw HTML.

There are a number of JavaScript-based rich-text editors (RTEs), available with different features and configurations, but I'll be using one called TinyMCE. One of the most popular options, it has roughly the best cross-browser support of any of the existing RTEs. (Due to differences in the APIs implemented by web browsers, there's no truly consistent cross-platform RTE at the moment.) TinyMCE is also free and released under an open source license. You can download a copy of the latest stable version from `http://tinymce.moxiecode.com/`.

Once you've unpacked TinyMCE, you'll see it contains a `jscripts/` directory, inside which is a `tiny_mce` directory containing all the TinyMCE code. Make a note of where that directory is, and go to the project's `urls.py` file. In `urls.py`, add a new line so that it looks like the following:

```
from django.conf.urls.defaults import *

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Example:
    # (r'^cms/', include('cms.foo.urls')),
```

```
    # Uncomment the admin/doc line below and add 'django.contrib.admindocs'
    # to INSTALLED_APPS to enable admin documentation:
    # (r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    (r'^admin/', include(admin.site.urls)),
    (r'^tiny_mce/(?P<path>.*)$', 'django.views.static.serve',
                { 'document_root': '/path/to/tiny_mce/' }),
    (r'', include('django.contrib.flatpages.urls')),
)
```

Replace the `/path/to/tiny_mce` part with the actual location on your computer of the `tiny_mce` directory. For example, if the directory resides at `/Users/jbennett/javascript/TinyMCE/jscripts/tiny_mce`, you'd use that value.

ADMONITION: MEDIA FILES IN PRODUCTION VS. DEVELOPMENT

In production, you'll usually want to avoid having the same web server handle both Django and static media files, like style sheets or JavaScript. Because the web-server process needs to keep a copy of Django's code and your applications in memory, it's a waste of resources to use that same process for the simple task of serving a file off the disk.

For now, I'm using a helper function built into Django that can serve static files, but keep in mind that you should use this only for development on your own computer. Using it on a live, deployed site will severely impact your site's performance. When you deploy a Django application to a live web server, consult the official Django documentation at `http://docs.djangoproject.com/` to see instructions for your specific server setup.

Now you just need to add the appropriate JavaScript calls to the template used for adding and editing flat pages. In the last chapter, when you filled in the `TEMPLATE_DIRS` setting, I mentioned that Django can also look directly inside an application for templates and that this capability lets an application author provide default templates while still allowing individual projects to use their own. That's precisely what you're going to take advantage of here. The `admin` application is not only designed to use its own templates as a fallback, but it also lets you provide your own if you'd like to customize it.

By default, the `admin` application will look for a template in several places, using the first one it finds. The template names it looks for are as follows, in this order:

1. `admin/flatpages/flatpage/change_form.html`
2. `admin/flatpages/change_form.html`
3. `admin/change_form.html`

ADMONITION: CHOOSING FROM MULTIPLE TEMPLATES

Normally, when you write a Django view—the function that actually responds to an HTTP request—you'll set it up to use a single template for its output. (The applications you'll write in this book will typically need to specify only one template for each view.) However, there is a helper function, `django.template.loader.select_template`, which takes a list of template names, searches for template files matching those names, and uses the first one it finds. The `admin` application makes use of this helper function to precisely enable the sort of customization I'm making here. If you're ever writing an application where you need to do the same, keep that function in mind.

The `admin` application provides only the last template in this list—`admin/change_form.html`—and uses that for all adding and editing of items if you don't supply a custom template. But as you can see, there are a couple of other options. By using a list of possible template names, rather than a single prebuilt template, the `admin` application lets you override the interface for a specific application (in this case, the `flatpages` application, by supplying the template `admin/flatpages/change_form.html`) or for a specific data model (by supplying the template `admin/flatpages/flatpage/change_form.html`). Right now you want to customize the interface for only one specific model. So inside your `templates` directory, create an `admin` subdirectory. Then create a `flatpages` subdirectory inside `admin` and a `flatpage` subdirectory inside `flatpages`. Finally, copy the `change_form` template from `django/contrib/admin/templates/admin/change_form.html` in your copy of Django into the `admin/flatpages/flatpage/` directory you just created.

Now you can open up the `change_form.html` template in your `template` directory and edit it to add the appropriate JavaScript for TinyMCE. This template is going to look fairly complex—and it is, because the `admin` application has to adapt itself to provide appropriate forms for any data model—but the change you'll be making is pretty simple. On line 6 of the template, you'll see the following:

```
{{ media }}
```

Immediately below that, add the following:

```
<script type="text/javascript" src="/tiny_mce/tiny_mce.js"></script>
<script type="text/javascript">
tinyMCE.init({
  mode: "textareas",
  theme: "simple"
});
</script>
```

This will make use of the URL you set up to serve the TinyMCE files. Now save the file and go back to your web browser. The form displayed for adding and editing flat pages will now have the basic TinyMCE editor attached to the text area for the page's content, as shown in Figure 3-1.

TinyMCE is extremely customizable. You can rearrange the editing toolbar, choose which of the many built-in controls should appear on it, add your own controls, and write new themes to change the way it looks. And if you'd like to use another RTE or make other customizations to the admin interface, you can follow the same process.

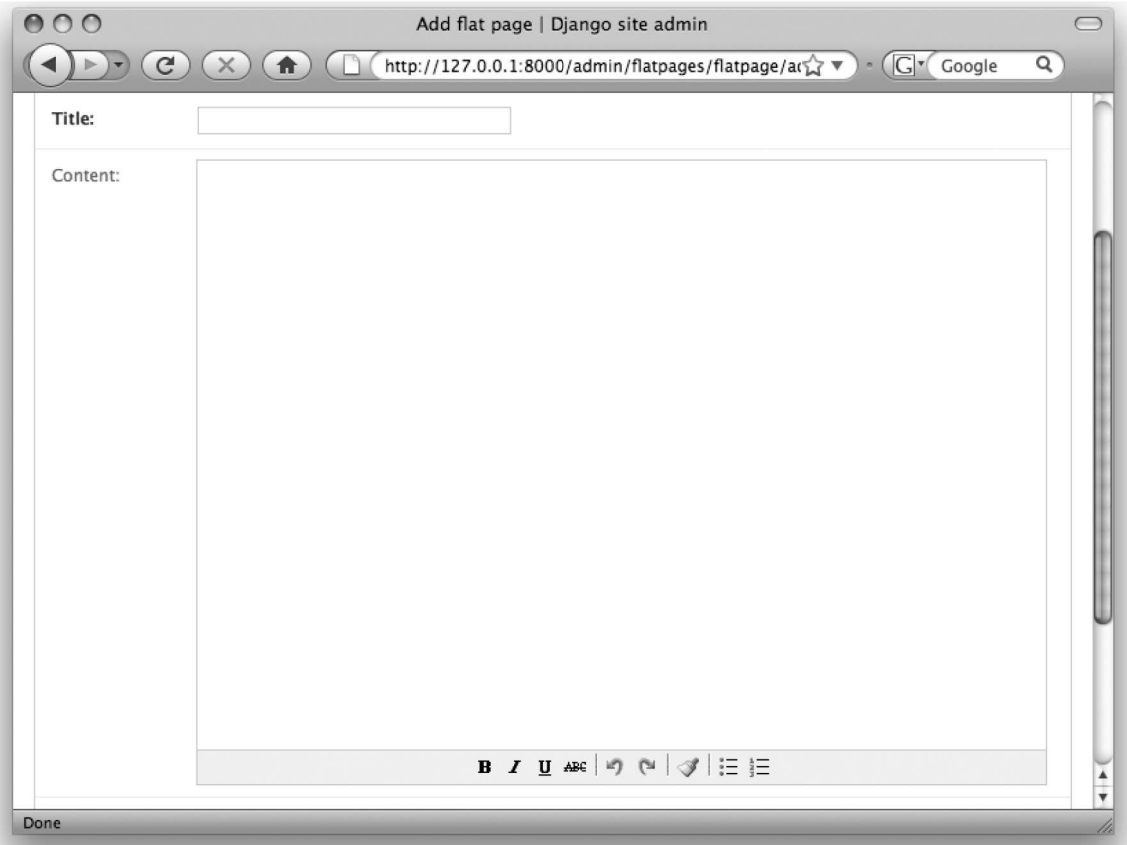

Figure 3-1. *The flat-pages admin form with rich-text editor*

Adding a Search System to the CMS

So far you've just been using the applications bundled with Django itself and making small customizations to the templates they use. Up to now that's accomplished a lot, but for most of your projects, you'll be writing your own applications in Python. So now you'll add a new feature—written in Python—to the simple CMS: a simple search system that lets users type in a query and get back a list of any pages whose titles or contents match.

It would be possible to add this directly to the `flatpages` application bundled with Django, but that's not really a good idea, for two reasons:

- It makes upgrading Django a hassle. You have extra Python code that didn't come with Django and the code needs to be preserved across the upgrade.
- A useful feature like a search system might need to be expanded later to work with other types of content, in which case it wouldn't make sense to have it be part of the `flatpages` application.

So let's make this into its own application. Go to your project directory and type the following command:

```
python manage.py startapp search
```

Just as the `startproject` command to `django-admin.py` created a new, empty project directory, the `startapp` command to `manage.py` creates a new, empty application module. It will set up the `search/` directory inside your project and add the following files to it:

- `__init__.py`: Just like the one in the project directory, this `__init__.py` file starts out empty. Its job is to indicate that the directory is also a Python module.
- `models.py`: This file will contain any data models defined for the application. A little later in this chapter, you'll write your first model in this file.
- `views.py`: This file will contain the view functions, which respond to HTTP requests and do most of the work of user interaction.
- `tests.py`: This is where you can place unit tests, which are functions that let you automatically verify that your application works as intended. You can safely ignore this file for now. (You'll learn more about unit tests in Chapter 11.)

For now, you'll just be writing a simple view, so open up the `views.py` file. The first step is to import the things you'll be using. Part of Python's (and Django's) design philosophy is that you should be able to clearly see what's happening with as little implicit "magic" as possible. So each file needs to contain Python `import` statements for things it wants to reference from other Python modules. To start, you'll need three `import` statements:

```
from django.http import HttpResponse
from django.template import loader, Context
from django.contrib.flatpages.models import FlatPage
```

These statements give you a solid foundation for writing your search view:

- `HttpResponse` is the class Django uses to represent an HTTP response. When an `HttpResponse` is given as the return value of a view, Django will automatically convert it into the correct response format for the web server it's running under.
- The `loader` module in `django.template` provides functions for specifying the name of a template file, which will be located (assuming it's in a directory specified in `TEMPLATE_DIRS`), read from disk, and parsed for rendering.
- `Context` is a class used to represent the variables for a template. You pass it to a Python dictionary containing the names of the variables and their values. (If you're familiar with other programming languages, a Python dictionary is similar to what some languages call a *hash table* or *associative array*.)
- `FlatPage` is the model class that represents the pages in the CMS.

ADMONITION: PYTHON NAMING STYLE

Every programming language has a set of standard conventions for how to name things. Java, for example, tends to use camel case, where things are given `NamesThatLookLikeThis`, while PHP tends to favor underscores, or `names_that_look_like_this`.

The standard practice in Python is that classes should have capitalized names—hence `Context`—and use the camel-case style for multiword names like `HttpResponse` or `FlatPage`. Modules, functions, and normal variables use lowercase names and underscores to separate multiple words in a name. Following this convention will help Python programmers—including you—quickly understand a new piece of code when reading it for the first time.

If you're interested in learning more about standard Python style, you can read the official Python style guide online at `www.python.org/dev/peps/pep-0008/`.

Now you're ready to write a view function that will perform a basic search. Here's the code, which will go into `views.py` below the `import` statements you added:

```
def search(request):
    query = request.GET['q']
    results = FlatPage.objects.filter(content__icontains=query)
    template = loader.get_template('search/search.html')
    context = Context({ 'query': query, 'results': results })
    response = template.render(context)
    return HttpResponse(response)
```

Let's break this down line by line. First, you're defining a Python function using the keyword `def`. The function's name is `search`, and it takes one argument named `request`. This will be an HTTP request (an instance of the class `django.http.HttpRequest`), and Django will ensure that it's passed to the view function when needed.

Next, look at the HTTP `GET` variable `q` to see what the user searched for. Django automatically parsed the URL, so a URL like this:

```
http://www.example.com/search?q=foo
```

results in an `HttpRequest` whose `GET` attribute is a dictionary containing the name `q` and the value `foo`. Then you can read that value out of it just as you would access any Python dictionary.

The next line does the actual search. The `FlatPage` class, like nearly all Django data models, has an attribute named `objects` that can be used to perform queries on that model. In this case, you want to filter through all of the flat pages, looking for those whose contents contain the search term. To do this, you use the `filter` method and the argument `content__icontains=query`, storing the results in a variable named `results`. This will provide a list of `FlatPage` objects that matched the query.

ADMONITION: DJANGO DATABASE-LOOKUP SYNTAX

As you'll see shortly, a Django data model has special attributes called fields, which usually correspond to the names of the columns in the database. When you use Django's object-relational mapper (ORM) to run a query, each argument in the query comprises a combination of a field name and a lookup operator, separated by double underscores.

In this case, the field name is `content` because that's the field on the `FlatPage` model that represents the page's contents (each `FlatPage` also has fields named `title`, `url`, and so on). The lookup operator is `icontains`, which checks whether the value in that column contains the string you've passed to it. The `i` at the front means the operator performs a case-insensitive lookup, so a query for `hello` would match both `hello` and `Hello`, for example. The Django ORM supports a large number of other lookup operators, many of which you'll see in action throughout this book.

Now that you have the query and the results, you need to produce some HTML and return a response. So the next line uses the `get_template` function of the `loader` module you imported to load a template named `search/search.html`. Next, you need to give the template some data to work with, so create a `Context` containing two variables: `query` is the search query, and `results` contains the search results.

You then use the template's `render` method, passing in the `Context` you created, to generate the HTML for the response. And finally, you'll return an `HttpResponse` containing the rendered HTML.

Now save the `views.py` file. You'll come back to it in a moment and make some improvements, but for now you need to create a template so that the search view can generate its HTML. Go into your `templates` directory, create a new subdirectory called `search`, and inside that create a file called `search.html`. Next you'll open up the `search.html` file and add the following to it:

```
<html>
  <head>
    <title>Search</title>
  </head>
  <body>
    <p>You searched for "{{ query }}"; the results are listed below.</p>
    <ul>
      {% for page in results %}
        <li><a href="{{ page.get_absolute_url }}">{{ page.title }}</a></li>
      {% endfor %}
    </ul>
  </body>
</html>
```

This makes use of both the variables passed to it. It uses `{{ query }}` to display the query and loops over the results to display them in an unordered list. (Remember, you can directly output variables in Django templates by wrapping their names in double curly braces.)

Notice that I've also used a Django template tag, `for`, which lets you loop over a sequence of things and do something with each one. The syntax is pretty simple. In effect, it says, "for each page in the `results` variable, display the following HTML, filled in with the values from that page." You can probably guess that, within the `for` loop, `{{ page.title }}` refers to the `title` field of the current page in the loop, but `{{ page.get_absolute_url }}` is new. It's standard practice for a Django model class to define a method called `get_absolute_url()`, which will output a URL to be used for referring to the object, and the `FlatPage` model does so. (Its `get_absolute_url()` method simply returns the value of its `url` field; other models can and will have more complex ways of working out their URLs.)

ADMONITION: CALLING AN OBJECT'S METHODS IN A DJANGO TEMPLATE

The Django template system lets you access methods on Python objects in the same way you access any other attributes: using a dot (`.`) character. For example, `{{ page.get_absolute_url }}` uses a dot to call the `get_absolute_url()` method. But note that in a template you don't use parentheses when calling a method, and you can't pass arguments to a method called in this way. This goes back to Django's philosophy of not allowing too much "programming" in templates—something that's complex enough to need arguments passed to it probably isn't purely presentational. The Django template system also forbids access to methods that alter the data in your database. Calls to those methods definitely belong in a view function and not in a template.

You can also access values from a dictionary by using the same dot syntax. As with the lack of parentheses in method calls, this is different from how you would do it in Python code (where dictionary access uses brackets, as in `request.GET['q']`), but it has the advantage of making the Django template syntax extremely uniform. The technique also serves as a reminder that Django templates are not simply Python code and therefore don't offer a full programming language.

Also, note that the `for` tag needs a matching `endfor` tag when you're done telling it what to do inside the loop. Most Django template tags that span a section of the template will need an explicit `end` tag to declare when you're done with them.

Now open your `flatpages/default.html` template and somewhere in it place the following HTML:

```
<form method="get" action="/search/">
  <p><label for="id_q">Search:</label>
  <input type="text" name="q" id="id_q" />
  <input type="submit" value="Submit" /></p>
</form>
```

This HTML adds a search box that will submit to the correct URL with the correct `GET` variable (`q`) for the search query.

Finally, open up your project's `urls.py`. After the lines for the admin and the TinyMCE JavaScript, but *before* the catch-all pattern for the flat pages, add the following:

```
(r'^search/$', 'cms.search.views.search'),
```

Remember that because this regular expression ends in a slash, you'll need to include it when you type the address into your browser. Unlike the URL patterns you've set up previously, which used the `include` directive to pull in other URLConf modules, this one maps the URL `search/` to a single specific view: the search view you just wrote. After saving the `urls.py` file, you should be able to type in a search query on any page in your CMS and get back a list of matching pages.

Improving the Search View

The search view works pretty well for something so short: it's only about a half dozen lines of code, plus a few `import` statements. But you can make it shorter, and it's a good idea to do so.

You'll notice that of the six lines of actual code in the search view, four are dedicated to loading the template, creating a `Context`, rendering the HTML, and returning the response. That's a series of steps you'll need to walk through on nearly every view you write, so Django provides a shortcut function called `django.shortcuts.render_to_response` that handles the process all in one step. So edit the `views.py` file to look like this:

```
from django.shortcuts import render_to_response
from django.contrib.flatpages.models import FlatPage

def search(request):
    query = request.GET['q']
    return render_to_response('search/search.html',
                              { 'query': query,
                                'results': FlatPage.objects.filter( ➥
                                           content__icontains=query) })
```

The `render_to_response` function gets two arguments here:

1. The name of the template file, `search/search.html`
2. The dictionary to use for the template's context

Given that information, it handles the entire process of loading the template, rendering the output, and creating the `HttpResponse`. Notice also that you're no longer using a separate line to fetch the results. They're only needed for the template context, so you can do the query right there inside the dictionary, trusting that its result will be assigned properly to the `results` variable. You've also broken up the arguments, including the dictionary, over several lines. Python allows you to do this any time you construct a list or dictionary (as well as in several other situations), and it makes the code much easier to read than if it were all sprawled out over one long line.

Save the `views.py` file, and then go back and perform a search again. You'll notice that it works exactly the same way, only now the search view is much shorter and simpler. And, importantly, it doesn't have the repetitive "boilerplate" of the template loading and rendering process. There will be times when you'll want to do that manually (for example, if you want to insert some extra processing before returning the response), but in general you should use the `render_to_response` shortcut whenever possible.

Another simple improvement would be to have the search view handle situations where it's accessed directly. Right now, if you just visit the URL `/search/` instead of accessing it through the search box on another page, you'll see an ugly error complaining that the key `q` wasn't found in the `request.GET` dictionary (because the `q` variable comes from performing a search). It would be much more helpful to simply display an empty search form, so let's rewrite the view to do the following:

```
def search(request):
    query = request.GET.get('q', '')
    results = []
    if query:
        results = FlatPage.objects.filter(content__icontains=query)
    return render_to_response('search/search.html',
                          { 'query': query,
                            'results': results })
```

Now you're using `request.GET.get('q', '')` to read the `q` variable. `get()`, a method available on any Python dictionary, lets you ask for the value of a particular key and specify a default to fall back on if the key doesn't exist (the default in this case is just an empty string). Then you can check the result to see whether there's a search query. If there isn't, you set `results` to an empty list, and that won't be changed. This means you can rewrite the template like this:

```
<html>
  <head>
    <title>Search</title>
  </head>
  <body>
    <form method="get" action="/search/">
      <p><label for="id_q">Search:</label>
      <input type="text" name="q" id="id_q" value="{{ query }}" />
      <input type="submit" value="Submit" /></p>
    </form>
    {% if results %}
      <p>You searched for "{{ query }}"; the results are listed below.</p>
      <ul>
        {% for page in results %}
          <li><a href="{{ page.get_absolute_url }}">{{ page.title }}</a></li>
        {% endfor %}
      </ul>
    {% else %}
      {% if query %}
        <p>No results found.</p>
      {% else %}
        <p>Type a search query into the box above, and press "Submit"
              to search.</p>
      {% endif %}
    {% endif %}
  </body>
</html>
```

Now the `search.html` template will show the same search box that appears on all the other pages in the CMS, and you'll notice that a `value` attribute has also been added to the HTML for the search input box. This way, if there was a query, it will be filled in as a reminder of what the user searched for.

I'm also using another new template tag: `if`. The `if` tag works similarly to the `if` statement in Python, letting you test whether something is true or not and letting you do something based on the result. It also takes an optional `else` clause, which I'm using to show a different message if the user hasn't searched for anything yet. Also, just as the `for` tag needs an `endfor` tag, `if` needs an `endif`. And finally, notice that you can nest the `if` tag: inside the `else` clause I'm using another `if` tag to differentiate between the results being empty because there was no query and the results being empty because no pages matched the query.

ADMONITION: SECURITY CONSIDERATIONS

One of the most common types of security problems with web applications is vulnerability to a cross-site scripting attack, or XSS. This sort of vulnerability occurs when you blindly accept input from a user and display it in a page on your site, as I'm doing with the search query. The problem is that a hacker can send a search query that contains HTML and JavaScript, then lure someone into visiting a page for that query. The JavaScript will be executed as if it were part of your site and could be used to hijack a user's account.

There's also a risk of another form of attack, called SQL injection, where a hacker relies on a web site to include user input directly in a database query. For example, a hacker might send a search query containing the text "DROP DATABASE;" which could—if blindly executed—delete the entire database for the site.

Django provides some built-in protection from these types of attacks, however. First, Django templates automatically "escape" the contents of any variables you display (so that, for example, the < character becomes `<`, removing the ability for a variable to end up as HTML that's rendered by a web browser). Second, Django carefully constructs database queries so that SQL injection isn't possible.

However, you shouldn't let these mechanisms lull you into a false sense of invincibility. Any time you're dealing with user-submitted data, you need to carefully ensure that you're taking appropriate steps to preserve your site's security.

Improving the Search Function with Keywords

The search function you've just added to the CMS is pretty handy, but you can make it a little bit better by adding the ability to recognize specific keywords and automatically pull up particular pages in response. This will let the site's administrators provide helpful hints for users who are searching and also creates useful metadata that you might want to take advantage of later.

To add this feature, you'll need to create a Django data model; models go in the `models.py` file, so open that up. You'll see that it already has an `import` statement at the top:

```
from django.db import models
```

This statement imports the module that contains all of the necessary classes for creating Django data models, and the `startapp` command automatically added it to the `models.py` file to help you get started. Below that line, add the following:

```
from django.contrib.flatpages.models import FlatPage


class SearchKeyword(models.Model):
    keyword = models.CharField(max_length=50)
    page = models.ForeignKey(FlatPage)


    def __unicode__(self):
        return self.keyword
```

This is a simple Django model with two fields:

- `keyword`: This is a `CharField`, which means it will accept short strings. I've specified a `max_length` of 50, which means that up to 50 characters can go into this field. In the database, Django will turn this into a column declared as `VARCHAR(50)`.
- `page`: This is a foreign key pointing at the `FlatPage` model, meaning that each `SearchKeyword` is tied to a specific page. Django will turn this into a foreign-key column referencing the table that the flat pages are stored in.

Finally, there's one method on this model: `__unicode__()`. This is a standard method that all Django model classes should define, and it's used whenever a (Unicode) string representation of a `SearchKeyword` is needed. If you've ever worked with Java, this is like the `toString()` method on a Java class. The `__unicode__()` method should return something that can sensibly be used as a representation of the `SearchKeyword`, so it's defined to return the value of the `keyword` field.

ADMONITION: PYTHON'S TWO TYPES OF STRINGS

Python actually has two different classes that represent strings: `str` and `unicode`. (There's also a parent class, `basestring`, which can't be instantiated directly but does provide a useful way to check whether something is a string type.) Instances of `str` are sometimes called bytestrings because each one corresponds to a specific series of bytes in a specific character encoding. (The default for Python is ASCII, but you can easily create strings in other encodings.) Instances of `unicode`, meanwhile, are strings of Unicode characters, and need to be converted to a byte-based encoding such as UTF-8 or UTF-16 before being output. (Unicode itself is not an "encoding.")

Because of this, Python classes can define either of two specially named methods, `__str__()` or `__unicode__()`, to provide string representations of themselves or, if necessary, they can define both. All of Django's internals are built to work with `unicode` strings, so it's best simply to define `__unicode__()`. Strings stored by Django models will be converted to `unicode` strings when they're retrieved from your database, and Django will automatically convert to appropriately encoded bytestrings when producing output for an HTTP response.

Be aware that not all Python software is written to handle `unicode` strings (or even non-ASCII-encoded bytestrings) properly. When you write applications that rely on third-party software, you will sometimes have to work around this by manually converting a string. Django provides a set of utility functions to make this easier, and in later chapters you'll see them in action.

Save the `models.py` file, then open the project's `settings.py` file and scroll down to the `INSTALLED_APPS` setting. Add `cms.search` to the list, and save the file. This will tell Django that the `search` application inside the `cms` project directory is now part of the project and that its data model should be installed. Next, run `python manage.py syncdb`, and Django will create the new table for the `SearchKeyword` model.

ADMONITION: WHY DID THE SEARCH VIEW WORK BEFORE?

You've probably noticed that I used the search view already without adding the `search` application to `INSTALLED_APPS`. This worked because you can take advantage of any Python code on your computer when routing URLs to view functions, regardless of whether they're in an application that's listed in `INSTALLED_APPS` or not. In fact, they don't have to be part of a Django application module at all. This means, if you really want or need to, you can keep stand-alone libraries of code on your computer and call on them from your Django projects.

Django does need to know exactly which applications to install data models for, however. So now that you've got a model, it's necessary to add the `search` application to `INSTALLED_APPS` so that Django will create the database table for it. There are some other features that require you to have an application and list it in `INSTALLED_APPS`. Most of the time you'll want to do that, regardless of whether it's strictly necessary (if for no other reason than to provide a quick reminder of what your project is using), but it's useful sometimes to know what requires this and what doesn't.

If you manually connect to your database and look at the table layout (consult the documentation for the specific database system you're using to see how to do this), you'll see that the new table was created with two columns corresponding to the fields on the `SearchKeyword` model. The table also has a third column, `id`, which is declared as the primary key and is an auto-incrementing integer. If you don't explicitly mark any of the fields in a model to serve as a primary key, Django will do this for you automatically.

Next, you'll want to enable the administrative interface for the new model. To do this, create a new file called `admin.py` and place the following code inside it:

```
from django.contrib import admin

from cms.search.models import SearchKeyword


class SearchKeywordAdmin(admin.ModelAdmin):
    pass


admin.site.register(SearchKeyword, SearchKeywordAdmin)
```

This code defines a subclass of `django.contrib.admin.ModelAdmin` called `SearchKeywordAdmin`. The `pass` statement means that you don't want to customize anything in this subclass (though in a moment you'll see how to make some changes to this type of class).

Then the `admin.site.register` function tells Django's administrative interface to associate this `ModelAdmin` subclass with the `SearchKeyword` model.

Now you can fire up the development web server again, and you'll see the new model appear in the index. You can add and edit keywords just as you can add and edit instances of any of the models fromthe other installed applications. Unfortunately, this interface is a little clunky: the keywords are added on a separate page, and you have to explicitly choose which page to associate each keyword with, as shown in Figure 3-2.

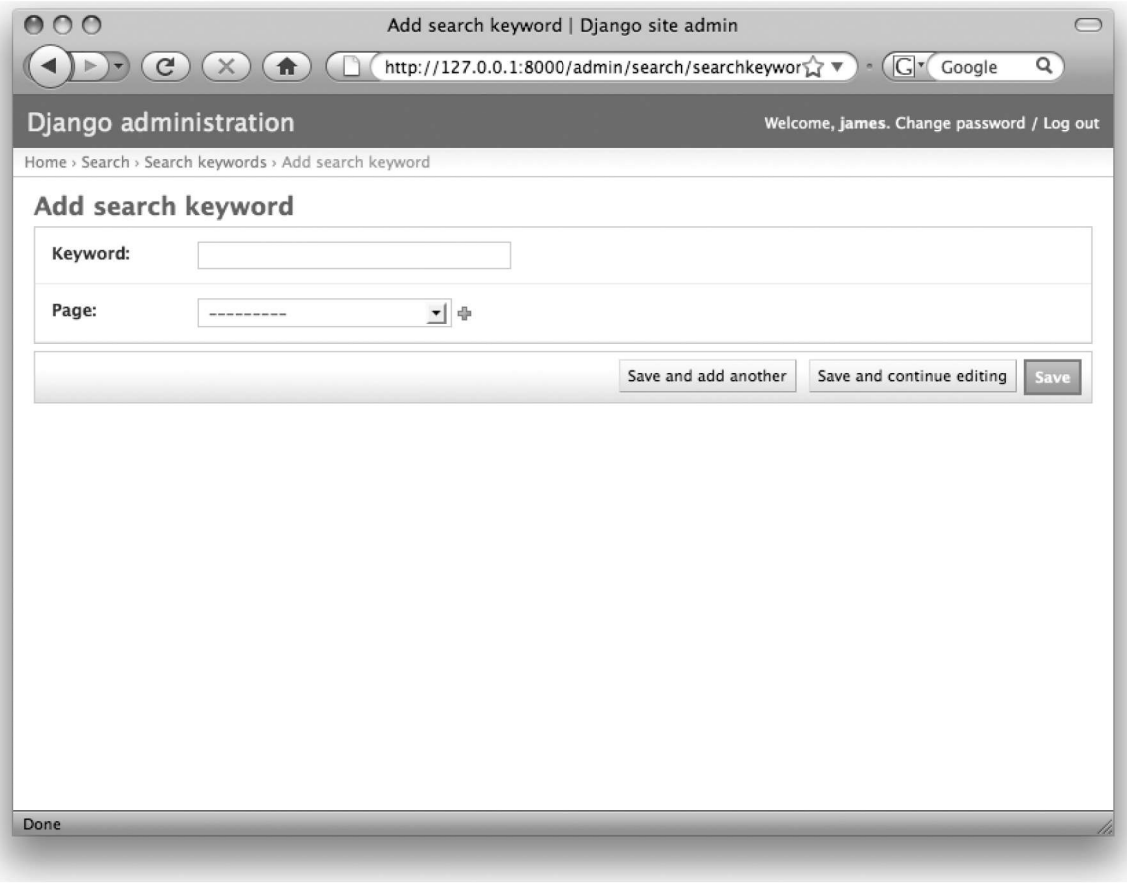

Figure 3-2. *The default admin form for a search keyword*

What you'd really like is to have the interface for the search keywords appear on the same page as the form for adding and editing pages. You can do that by making a few small changes to the `SearchKeyword` class so that it looks like this:

```
from django.contrib import admin
from django.contrib.flatpages.admin import FlatPageAdmin
from django.contrib.flatpages.models import FlatPage

from cms.search.models import SearchKeyword


class SearchKeywordInline(admin.StackedInline):
    model = SearchKeyword
```

```
class FlatPageAdminWithKeywords(FlatPageAdmin):
    inlines = [SearchKeywordInline]

admin.site.unregister(FlatPage)
admin.site.register(FlatPage, FlatPageAdminWithKeywords)
```

This code is doing several things. First, it defines a new type of class: a subclass of `django.contrib.admin.StackedInline`. This class allows a form for adding or editing one type of model to be embedded within the form for adding or editing a model it's related to. (There's another class for this as well, called `TabularInline`; the difference between these classes is in the way the form will look when embedded.) In this case, the class is told that its model is `SearchKeyword`, which means it will embed a form for adding or editing search keywords.

Next, the existing `admin` class for the `FlatPage` model is being imported and subclassed, and a new option is added to it: the `inlines` declaration, which should be a list of inline classes to use. This just lists the `SearchKeywordInline` class you've defined. Finally, the `admin.site.unregister` function removes the existing `admin` definition that the `flatpages` application provided, and a call to `admin.site.register` replaces it with the new definition you've just written.

Once you've saved this file, you can go back to the admin interface in your browser and see that each flat page now has several inline forms for search keywords (see Figure 3-3).

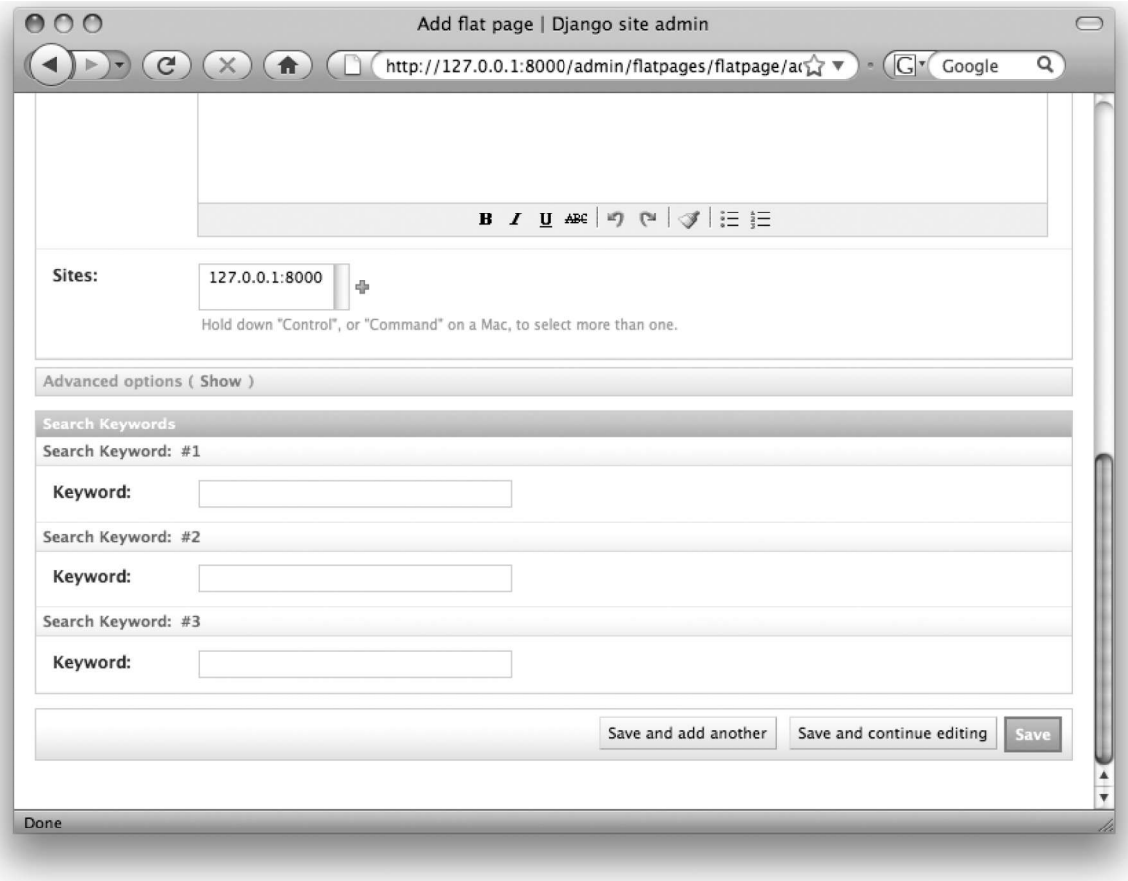

Figure 3-3. *Search keywords can be added and edited inline, alongside a flat page.*

Go ahead and add some keywords to the pages in your database; you'll want them to be available when you try out the improved keyword-based search.

Adding support for keywords in the search view is pretty easy. Just edit the view so that it looks like the following:

```
    def search(request):
        query = request.GET.get('q', '')
        keyword_results = []
        results = []
        if query:
            keyword_results = FlatPage.objects.filter( ➥
searchkeyword__keyword__in=query.split()).distinct()
            results = FlatPage.objects.filter(content__icontains=query)
        return render_to_response('search/search.html',
                                  { 'query': query,
                                    'keyword_results': keyword_results,
                                    'results': results })
```

You've added a second query in the preceding code, which looks up pages whose associated search keywords match the query. Though it may look daunting at first, it's actually pretty simple.

First you're using a call to `filter`, just as in the other query. This one, though, is interesting. It's actually reaching "across" the foreign key from the `SearchKeyword` model and looking in the `keyword` field there. Any time you have a relationship like this between models, you can chain lookups across the relationship by using double underscores: `searchkeyword__keyword` translates to "the keyword field on the related `SearchKeyword` model." The lookup operator here is `__in`, which takes a list of things to match against. You're feeding it `query.split()`. At this point the `query` variable is a string, and Python provides a `split()` method which, by default, splits on spaces. This is exactly what you want—to be able to handle queries that contain multiple words.

Next, the call to `filter` is followed by `distinct()`. The nature of this query means that, if a single page has multiple keywords that match the search, multiple copies of that page will show up in the results. You want only one copy of each page, so you use the `distinct()` method, which adds the SQL keyword `DISTINCT` to the database query.

Finally, you add `keyword_results` to the context you'll be using with the template. The template will need to update. Though it's getting a little more complex because of the multiple cases it has to handle, it's still fairly straightforward to follow:

```
<html>
  <head>
    <title>Search</title>
  </head>
  <body>
    <form method="get" action="/search/">
      <p><label for="id_q">Search:</label>
      <input type="text" name="q" id="id_q" value="{{ query }}" />
      <input type="submit" value="Submit" /></p>
    </form>
```

```
    {% if keyword_results or results %}
      <p>You searched for "{{ query }}".</p>
      {% if keyword_results %}
        <p>Recommended pages:</p>
        <ul>
          {% for page in keyword_results %}
            <li><a href="{{ page.get_absolute_url }}">{{ page.title }}</a></li>
          {% endfor %}
        </ul>
      {% endif %}
      {% if results %}
        <p>Search results:</p>
        <ul>
          {% for page in results %}
            <li><a href="{{ page.get_absolute_url }}">{{ page.title }}</a></li>
          {% endfor %}
        </ul>
      {% endif %}
    {% endif %}
    {% if query and not keyword_results and not results %}
      <p>No results found.</p>
    {% else %}
      <p>Type a search query into the box above, and press "Submit"
            to search.</p>
    {% endif %}
  </body>
</html>
```

The complexity really comes from the nested `if` tags to deal with the various cases, but those nested tags let you cover every possibility. Also, notice the line that reads `{% if keyword_results or results %}`: the `if` tag lets you do some simple logic to test whether any or all of a set of conditions are met. In this case, it provides an easy way to handle the situation where there's *some* type of result, and then it tackles the different cases individually, as needed. If you've added some keywords to the pages in your database, try searching for those keywords now, and you'll see the appropriate pages show up in the search results.

Before I wrap up, let's add one more useful feature to the search view. If there's only one result that precisely matches a keyword, you'll redirect straight to that page and save the user a mouse click. You can accomplish this by using `HttpResponseRedirect`, a subclass of the `HttpResponse` class that issues an HTTP redirect to a URL you specify. Open up `views.py` and add the following line at the top:

```
from django.http import HttpResponseRedirect
```

This is necessary because, again, Python requires you to explicitly import anything you plan to use. Now edit the search view like this:

```
def search(request):
    query = request.GET.get('q', '')
    keyword_results = results = []
```

```
    if query:
        keyword_results = FlatPage.objects.filter( ➥
                          searchkeyword__keyword__in=query.split()).distinct()
        if keyword_results.count() == 1:
            return HttpResponseRedirect(keyword_results[0].get_absolute_url())
        results = FlatPage.objects.filter(content__icontains=query)
    return render_to_response('search/search.html',
                              { 'query': query,
                                'keyword_results': keyword_results,
                                'results': results })
```

Up until now, you've been treating the results of database queries like normal Python lists, and, although they can be used like that, they actually make up a special type of object called a `QuerySet`. `QuerySet` is a class Django uses to represent a database query. Each `QuerySet` has the methods you've seen so far—`filter()` and `distinct()`—plus several others, which you can "chain" together to build a progressively more complex query. A `QuerySet` also has a `count()` method, which will tell you how many rows in the database matched the query. (It does a `SELECT COUNT` to find this out, though for efficiency reasons, it can also take advantage of some other methods that don't require an extra query.)

ADMONITION: WHEN DOES DJANGO EXECUTE THE QUERY?

The single most important feature of `QuerySet` is that it's "lazy." Initially, it doesn't do anything except make a note of what query it's eventually supposed to execute in the database, which is why you can keep chaining extra things onto it to add filtering, a `DISTINCT` clause, or other conditions. The actual database query won't be executed until you do something that forces it to happen, like (in this case) counting or looping over the results.

By using `count()`, you can see whether a keyword search returned exactly one result and then issue a redirect. The URL you redirect to is `keyword_results[0].get_absolute_url()`; this bit of code pulls out the first (and, in this case, only) page in the results and calls its `get_absolute_url()` method to get the URL.

Go ahead and try this out. Add a new search keyword that's unique to one page, and then search for it. If you've set up the view as previously described, you'll immediately be redirected to that page.

Looking Ahead

In the last two chapters, you've gone from literally *nothing* to a useful, functional CMS with an easy web-based administrative interface. You added rich-text editing to prevent users from having to write raw HTML, and you added a search system that allows administrators to set up keyword-based results. Along the way, you've written fewer than a hundred lines of actual

code. Django did most of the heavy lifting, and you just supplied the templates and a little bit of code to enable the search function.

Best of all, you now have a simple, reusable solution for a common web-development task: a brochureware-style CMS. Any time you need to re-create it, you can set up Django and walk through these same easy steps (or even just make a copy of the project, changing the appropriate settings in the process). Doing this will save you time and free you from the tedium of a fairly repetitive situation.

Feel free to spend some time playing around with the CMS: add some style to the templates, customize the admin pages a bit more, or—if you're feeling really adventurous—even try adding a few features of your own. If you'd like a homework assignment of sorts, check out the Django database API documentation (online at `www.djangoproject.com/documentation/db-api/`) and see if you can work out how to add an index view that lists all of the pages in the database.

When you're ready for a new project, start reading the next chapter, where you'll be starting on your first application from scratch: a Django-powered weblog.