Django and the Django Software Foundation

FRET FREE

We talk to one of the creators of the Django project about the formation of the Django Software Foundation, and we show you how to get started with this user-friendly web framework. **BY FRANK WILES**

In the summer of 2005, yet another web framework was released into the open source world [1]. Only three short years since Django's release, it has gained enough traction to inspire the formation of the Django Software Foundation [2]. With the formation of the DSF, Django joins an impressive list of other projects with their own foundations, including Apache, Perl, and Python.

What Is Django?

Django is a Python web development "framework," or set of libraries, that allows developers to work on the unique/interesting parts of an application without worrying much about the boring infrastructure under the hood. Django uses the MVC pattern like many other frameworks, such as Ruby on Rails and the various Perl and PHP frameworks.

One of Django's killer features is its incredibly slick admin interface that is automatically built for you. In this article, I will walk through the steps required to build a small Twitter-like application so you can see the admin in action.

Django has been used to build a lot of high-profile websites [3], such as *EveryBlock.com*, *Pownce.com*, and *Tabblo.com*. Also, it is the default framework included with Google's AppEngine, and I've heard that Google uses it to some extent internally. Django is also the foundation of the commercial CMS Ellington, which is used by several large news organizations, including The Washington Post.

Jacob Kaplan-Moss, President of the Django Software Foundation and one of the creators of Django, said that the Foundation was created so the project could take the next step in its life cycle as an open source project. "We've obviously succeeded in attracting a large, vibrant community, so we felt that it was time that the community really 'owned' Django. Having the Foundation around pretty much guarantees that Django will stick around, even if any individuals or companies lose interest," he says.

Kaplan-Moss points out that the project now accepts donations for improving Django, and in the near future, the Foundation will primarily support Django through developer sprints, user meet-ups, and other community activities. Many development sprints will occur before the release of Django 1.0, and Kaplan-Moss says the Foundation will help key people attend sprints and work together. "If the Foundation helps Django move forward even a tiny bit faster, I'll be thrilled," Kaplan-Moss says.

Getting Started

Django is scheduled to release an official version 1.0 in early September 2008. For this article, I'll use the bleeding-edge code from Subversion, which should closely match the release. Your best bet is to install the official 1.0 release [4] when available or grab the bleeding-edge code from Subversion with:

```
svn checkout ➥
http://code.djangoproject.com/➥
svn/django/trunk/
```

Regardless of which version you choose, Django installs easily. While you're con-

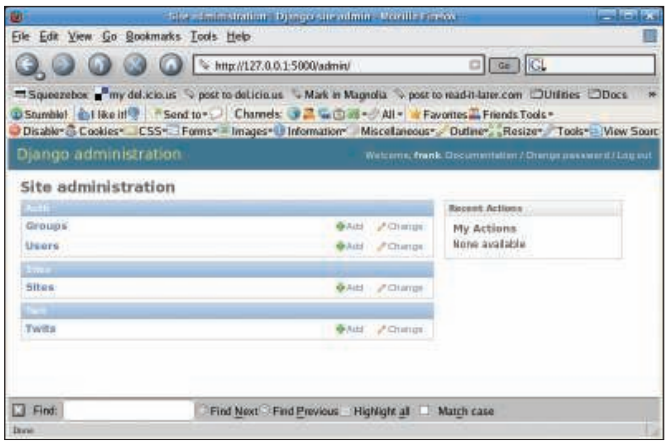

Figure 1: The Django administration screen.

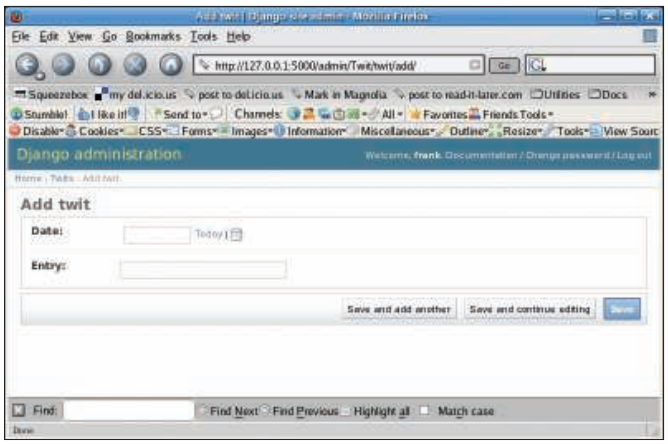

Figure 2: Add information for your personal app.

nected to the Internet, simply run the following as root

```
python setup.py install
```

to install Django into the *site-packages* directory of wherever your Python installation lives. For the example, I'll use SQLite as the database. However, Django has excellent support for PostgreSQL and MySQL.

To use SQLite, install the pysqlite2 package [5] and follow its installation instructions.

Django separates everything into "projects" and "apps." For example, if you build a large website with a blog, forum, and e-commerce section, the site itself is the project, and the blog, forum, and e-commerce code are apps. Really, this is just a way to organize sub-projects within your overall project.

To start a new project, run

```
django-admin.py startproject mytwit
```

Listing 1: mytwit/Twit/models.py

```
01 from django.db import models
02
03 class Twit(models.Model):
04     date = models.DateField('Date')
05     entry = models.CharField( max_length='500' )
06
07     def __unicode__(self):
08         return '%s %s' % (self.date, self.entry)
09
```

which creates the directory *mytwit* with a few initial stub configuration files and tools. Now you need Django to generate the stubs for the example app, which I will call Twit. To do so, run from inside the *mytwit* directory:

```
python manage.py startapp Twit
```

In *mytwit/settings.py*, set *DATABASE_ENGINE = 'sqlite3'* and *DATABASE_NAME* to the full path to *mytwit/twits.db*, the SQLite file in which your database will be stored. The full path depends on the directory in which you ran the initial *startproject*. So that you don't have to revisit *settings.py* again, you must add two items to the *INSTALLED_APPS* list: *django.contrib.admin* for the admin interface and the *mytwit.Twit* app. If you add to the end of the list, make sure you add the trailing commas.

After defining the database you will use, you must build your *Model*, which is a Python object that defines the SQL tables and columns and their relationships. Because your simple application has only one table, just define the one class. Thus, *mytwit/Twit/models.py* should be as shown in Listing 1.

First, import the Django model helpers and then define your *Twit* class, which will contain a date column and a text column for your actual entry. Then define the special *__unicode__* method, which tells the *Model* how to display an instance of the object in string form (in this case, just print the date and full entry). This information is used by the admin when displaying listings of entries from the database. The empty class *Admin* tells Django that you want it to provide the admin interface for you.

To check what you've done, validate your models by running:

```
python manage.py validate
```

If everything is okay, it should return *0 errors found*. Now Django can build the database tables. To do so, enter

```
python manage.py syncdb
```

which outputs several *Creating table* lines, some of which are for *user/group* permissions, others for the admin, and the final for the Twit table. At this point, Django also prompts you to create a superuser for the admin interface, but remember the username and password, which you will need later.

After successfully creating the Model and database tables, we need to turn on the admin interface. This is done by uncommenting the three lines in the mytwit/urls.py file that was created when we ran *startproject*. The three lines are labeled telling you to uncomment them to turn on the admin. The urls.py file is how Django maps different URLs

Listing 2: Reverse Entries

```
01 from django.shortcuts import render_to_response
02 from models import Twit
03
04 def alltwits(request):
05     all_entries = Twit.objects.all().order_by("date").reverse()
06     return render_to_response('all_twits.html', { 'entries': all_entries })
```

Listing 3: Building the Template

```
01 <html>
02 <body>
03 <table>
04   <tr>
05     <th>Date</th>
06     <th>Entry</th>
07   </tr>
08   {% for t in entries %}
09       <tr>
10         <td>{{ t.date }}</td>
11         <td>{{ t.entry}}</td>
12       </tr>
13   {% endfor %}
14 </table>
15 </body>
16 </html>
```

to different parts of your application with regular expressions.

Additionally, we need to create an admin.py file. In this example, we're just using the defaults, but this is where you could customize various aspects of the admin interface. For this short example, *mytwit/Twit/admin.py* needs to read:

```
from django.contrib import admin
from mytwit.Twit.models ➥
import Twit

class TwitAdmin(admin.➥
ModelAdmin):
      pass

admin.site.register➥
(Twit, TwitAdmin)
```

To see it in action, run

```
python manage.py runserver
```

which runs a test server on 127.0.0.1:8000. If you need to run it on a different IP address or port, you can append that to the command with:

```
python manage.py ➥
runserver 127.0.0.1:5555
```

Assuming you're using the defaults, go to *http://127.0.0.1:8000/admin* and you will be prompted to log in to your app's admin interface. After logging in, you will see the screen shown in Figure 1.

Because you are building a personal application, you can ignore the *Sites* and admin sections for now and just click on the *Add* icon in the Twit box. Then you will see something like Figure 2.

Now you can insert data for your entry. Clicking *Today* automatically fills in today's date, or you can use the calendar widget to pick another date. Next, input text in *Entry* and click *Save*, after which you return to a page that lists all the Twits in your database. If you click on the entry you just made, you go to an edit/delete interface to make changes or remove the entry.

Playing is fun, but you should share these entries on the web with friends. To do so, you must add a view (a module that performs the logic) and a template (how data are presented to the user). If you are used to other MVC frameworks, in which the view typically refers to the template itself, this can be confusing.

To begin, edit your *mytwit/Twit/views.py* to contain a simple method that returns all of your entries in reverse chronological format (Listing 2). This defines the method *alltwits*, which grabs all of your Twit objects ordered by the date field and reverses them. Then it calls *render_to_response()* with the name of the template for the view and a dictionary that contains the data you want passed on to the template.

After you're done with the view, you need to build the template. To make things fit better on the page, see my simple markup example (Listing 3) to get a general idea. To keep your templates separate from everything else, save this file in *mytwit/templates/all_twits.html*.

As you can see, the Django template language has advanced features and is easy to use. Here you use a simple *for* loop to go through each Twit object you passed in from the *alltwits()* method and to display the data via the *date* and *entry* methods on each Twit object in *entries*.

Now configure Django to find your template on the file system and set up a URL that maps to the view. To set up template directories in *mytwit/settings.py*, you must add the full path to the *TEMPLATE_DIRS* list, which depends on where you ran *startproject* (be sure to use the full path). Now edit *mytwit/urls.py* to map the URL (Listing 4).

Here, you have imported your app-specific views, added a */twits/* URL, and left the default Django admin mapping alone. Now if you go to *http://127.0.0.1:8000/twits*, you should see all of the Twits you entered in the admin interface.

Use of the standalone server and SQLite is great for quick development, but if you want to build a production app, you should switch to Apache, mod_python, and a more robust database such as PostgreSQL (see the Django site). Thanks go to Jacob Kaplan-Moss and Adrian Holovaty for contributing to this article. Code listings can be downloaded from [7]. ■

Listing 4: Map to the URL

```
01 from django.conf.urls.defaults import *
02 from django.contrib import admin
03 from mytwit.Twit import views
04
05 admin.autodiscover()
06
07 urlpatterns = patterns('',
08     (r'^twits/', 'mytwit.Twit.views.alltwits'),
09     (r'^admin/(.*)', admin.site.root ),
10  )
```

THE AUTHOR

Frank Wiles is the owner of Revolution Systems (*http://www.revsys.com*), an Internet infrastructure and web-development consultancy specializing in scaling and performance-tuning open source software.

INFO

[1] Django Project Homepage: *http://www.djangoproject.com*

[2] The Django Software Foundation: *http://www.djangoproject.com/foundation*

[3] Django-powered sites: *http://www.djangosites.org*

[4] Download Django: *djangoproject.com/download/*

[5] Pysqlite2: *http://initd.org/pub/software/pysqlite/*

[6] The Django Book: *http://www.djangobook.com*

[7] Article Code: *http://www.linux-magazine.com/resources/article_code*