DJANGO

When Python Bites the Web

Here's how to start using Django for Web application development.

Quoting Wikipedia, "software frameworks aim to facilitate software development by allowing designers and programmers to devote their time to meeting software requirements rather than dealing with the more standard low-level details of providing a working system, thereby reducing overall development time."

Rails is one of the most famous Web application development frameworks on top of the Ruby programming language. Django, on the other hand, is an advanced Web programming framework built on top of Python. Django is a strong competitor of Ruby on Rails.

Django abstracts lots of background details, like the SQL database, by providing higher-level interfaces. It is a very handy framework and a quick solution to build any complex Web application. If you are already familiar with the Python programming language, you would find it easy to hack with Django.

Let us get started

We will now look at how to get started with Django. You need the Django development server to work with it. Download the development server from *www.djangoproject.com* and install it as follows:

```
$ wget http://www.djangoproject.com/download/1.1.1/tarball/
$ tar xzvf Django-1.1.1.tar.gz
$ cd Django-1.1.1
$ sudo python setup.py install
```

...or on Ubuntu:

```
$ sudo apt-get install python-django
```

On installation, certain Django utilities like *django-admin* and *manage.py* will be available. We will use these tools throughout the development process.

Now let us dig into the basics. A Django project is an environment where we deploy the Django server. A Django project may contain more than one application. Here, there are Django Web applications that are sub-grouped as different modules. You will understand the distinction between a Django project and application in detail when we look at the code.

You can create a new Django project using the *django-admin* command:

```
slynux@hackbox:~/LFY$ django-admin startproject helloweb
slynux@hackbox:~/LFY$ cd helloweb/
```

```
slynux@hackbox:~/LFY/helloweb$ ls
__init__.py  manage.py  settings.py  urls.py
```

A directory named *helloweb/*, which is the name given to the project in the above snippet, will be created. The directory will contain the basic skeleton for a Django project. Figure 1 shows the directory skeleton structure.

You will find the *__init__.py* file in most of the directories in a Django project. Its purpose is to mark the directories on a disk as Python package directories, to be imported by Python in the program.

For example, if we have a directory *foo/* and it contains the files *__init__.py* and *linux.py*, we can access its contents from a Python shell as follows:

```
>>> from foo import linux
```

The *manage.py* file is the project management file. We will execute this file to do further management tasks related to the current project after the basic skeleton is created. We will use *manage.py* to start the development server, create new applications, synchronise database tables and do many other things inside the project directory in which *manage.py* exists.

The *settings.py* file holds the different settings variables for the current Django project. It includes the time zone used, database authentication and connectivity, templates, additional Django extensions support, etc. It manages the overall settings for the project.

The *urls.py* file is the URL map file for the current Django project. In *urls.py*, we will add different URL mapping schemes. When we request some URL, the function of this file is to specify what should be displayed as the output. It has a pretty good regex support, and many customisation options. It directs each URL to a view function, which returns the output HTTP response.

If some non-existent URL is requested, we can direct it to a custom '404 Error' page. Or if we request a URL in the format of *http://example.com/page/(digits)/* (for example, *http://example.com/page/11/*), we can write custom regular expression patterns such that this Web page will always be handled by a custom-written page function and the *(digits)* will be taken as the parameter.

Now, let's look at how to set up a basic Django HelloWorld application.

We'll use the Django development server to test our Web application. After development, we will deploy it on an Apache server with Django extension support.

Navigate to the project skeleton directory that we have created and issue the following command:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py runserver
Validating models...
0 errors found
Django version 1.1.1, using settings 'helloweb.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Figure 1: Directory structure of the 'helloweb' Django project

The above snippet informs us that a development server has started and we can access the output pages from the local host machine by pointing a Web browser to the following URL: *http://127.0.0.1:8000/*. By default, the development server runs at port 8000. We can change it by using the desired port number as an additional argument along with the *manage.py* command as follows:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py runserver port_no
```

Shell

There is an option provided by *manage.py*—the Django *shell*. We can use it to access the Django Python interpreter (which runs the current project's Python interpreter with its environment set-up). It is very helpful during the debugging process:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py shell
Python 2.6.4rc2 (r264rc2:75497, Oct 20 2009, 02:55:11)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

We need to write a sample Web page using Django now. Create a file *views.py* and append the following lines in it:

```
from django.http import HttpResponse
def index(request):
      html = "<h1>Hello Web !</h1>"
    return HttpResponse(html)
```

The above snippet of code is known as a 'view'. Views generate the HTML data for the Web page:

We need to set up the *urls.py* next. Add the following line to *urls.py*:

```
(r'^helloweb/$', 'helloweb.views.index'),
```

...so that it looks like the following code:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^helloweb/$', 'helloweb.views.index'),
)
```

Now, using a browser, check out the following Web page: *http://localhost:8000/helloweb*. You can see the output returned by the *index()* function.

Project settings

The *settings.py file* handles the overall settings for the current Django project. Let's take a look at the basic file structure. Here, I will only explain the important sections in this file. It is much easier to comprehend other options since they are self-explanatory, thanks to the excellent commenting:

```
slynux@hackbox:~/LFY/helloweb$ less settings.py:
DATABASE_ENGINE = ''   # 'postgresql_psycopg2', 'postgresql', 'mysql',
                       # 'sqlite3' or 'oracle'.
DATABASE_NAME = ''     # Or path to database file if using sqlite3.
DATABASE_USER = ''     # Not used with sqlite3.
DATABASE_PASSWORD = '' # Not used with sqlite3.
DATABASE_HOST = '' # Set to empty string for localhost.
                    # Not used with sqlite3.
DATABASE_PORT = '' # Set to empty string for default.
                    # Not used with sqlite3.
```

This section is used to provide the database settings. In Django, we use the models concept to deal with the database and datamodels. It supports all the various database servers that are widely used. To use a MySQL server, we can configure the settings as follows:

```
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'database_name'
DATABASE_USER = 'database_username'
DATABASE_PASSWORD = 'password'
```

We may use CSS and JS files as resource files for our Web page. In order to reference them we need to provide the absolute path and the media URL path. For example:

```
MEDIA_ROOT = '/home/slynux/LFY/helloweb/media'
MEDIA_URL = ''http://localhost:8000/helloweb/media'
```

We'll need to specify the *MEDIA_URL* parameter in the URL format that points to the absolute path. We will use the *MEDIA_URL* to access media files like CSS, JS and other files accessible throughout the Web URL.

To facilitate template support, we use template settings in the *settings.py* file. We specify the templates directory path using the *TEMPLATE_DIRS* tuple as follows:

```
TEMPLATE_DIRS = (
"/home/slynux/LFY/helloweb/templates_dir1",
"/home/slynux/LFY/helloweb/templates_dir2"
)
```

The *INSTALLED_APPS* section facilitates using third-party Django modules along with our project. For example, if we need a user registration application, we need not write it from scratch. We can download it and attach that application to our project by keeping the application in our project directory and adding a line in the *INSTALLED_APPS* tuple. When we add some applications to the current project, we have to add its reference path to this tuple:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
)
```

Views and templates

Writing dynamic pages in Django is very easy. Django puts different concepts into practice—like views and templates.

Views are routines that actually return an output as a Web page, i.e., a dynamic page. We can keep several view routines for different URLs. When a URL is requested by a browser, the Django server checks its mapping from the *urls.py* file, and the corresponding function specified in it is displayed as the output.

Templates in Django are similar to what we mean by templates, in general. The basic idea is to keep the HTML user interface code separate from the logic code and data processing side. Templates provide a higher abstraction over the logic. We can use the variables and data that are returned by the core functions as required by the user interface part. Thus we can simply write the user interface code without bothering about the core logic.

We will now look at how to write view functions in practice. We can place our views in any custom *.py* file. But we have to specify its path in the *urls.py* URL configuration file.

Consider the following code fragment:

```
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello World")
```

This is the simplest form of the view function.

To map a URL to a view function, we need to append the following line to it:

```
(r'^helloweb/$', 'helloweb.views.index'),
```

Here, *r'^helloweb$'* is the URL regular expression; the ^ symbol specifies that the page URL starts with *helloweb*, and $ specifies that the URL ends with helloweb. Hence, the URL is matched to the *index()* function only when a correct 'helloweb' occurs in the URL.

If the regular expression was just *r'helloweb'*, it can match any URL, like *something_helloweb_something*.

The *'request'* parameter in a view function is the *HttpRequest* object. We can obtain data related to a session, *GET* request, and *POST* request from the *'request'* object.

To obtain *POST* request variables, we can use *post_var = request.POST*. It will return a dictionary that contains all post variables and their values. We can use *get_var = request.GET* to obtain all GET variables and values. Similarly, for a Session, use *session_var = request.session*.

To check whether a variable exists in a dictionary, we can make use of the *has_key()* method. Have a look at the modified index view function that makes use of GET variables:

```
def index(request):
      html = "Hello world, No page requested"

  if request.GET.has_key('page_no')
      html = "Hello World, Page requested is no: %s" %request.GET['page_no']

    return HttpResponse(html)
```

Now the page responds according to the GET request variable, page_no. Try different URLs—for example, *http://127.0.0.1:8000/helloweb/?page_id=100* and *http://127.0.0.1:8000/helloweb/*

We will now look at how to handle *page/15/*-type URL mapping to views—for example, *http://127.0.0.1:8000/helloweb/page/44/*. Peeking inside the *urls.py* file, we have:

```
urlpatterns = patterns('',
       (r'^helloweb/page/(?P<page_id>\d+)/$','helloweb.views.page'),
       )
View function:
def page(request, page_id):
      return HttpResponse("This is Page: %s" %page_id)
```

Here, *(?P<page_id>\d+)* is the regular expression technique used: *\d+* defines the type of data, i.e., the digits; and *<page_id>* defines the name of the variable, so that we can use it as an argument for the view function page.

The view function can parse the page number from the URL and set *page_id=number*, so that we can manipulate the output *HttpResponse* according to that page number.

Here we have discussed a simple HttpResponse, which always returns HTML code given as an argument. Now we will look at how to deal with templates.

From now, we will place the HTML code as separate *.html* files, instead of inside the view function—and we call them as templates. Set the templates directory in *settings.py*:

```
TEMPLATE_DIRS = (
"/home/slynux/LFY/helloweb/html_templates"
)
```

Create an HTML file in the *html_templates/* directory called *template1.html*, with the following content:

```
<html>
<head><title> Django Hello Web project</title></head>
<body>
<p> This is a helloweb template</p>
<table>
<tr><td>Student </td><td>Roll No</td></tr>
<tr><td>S1</td><td>1</td></tr>
<tr><td>S2 </td><td>2</td></tr>
<tr><td>S3 </td><td>3</td></tr>
</table>
</body>
</html>
```

Now we can modify the view function to support the templates. The HTML interface code is taken through a *get_template* function. Here, *Context()* has a dictionary argument. It takes the variables and objects to be passed to the template to generate dynamic code. However, no objects are passed to the template—the template is just a static HTML page:

```
def index(request):
      from django.template.loader import get_template
      from django.template import Context
      site_template  = get_template('template1.html')
      html = site_template.render(Context({}))
      return HttpResponse(html)
```

Or we can write the same template rendering in one line using the *render_to_response* shortcut as follows:

```
def index(request):
      from django.shortcuts import render_to_response
      return render_to_response('template1.html',{})
```

We will receive an output like what's shown below:

This is a helloweb template

Student	Roll No
S1	1
S2	2
S3	3

This is purely static. Now we will write a dynamic page utilising the facilities of the Django template system. Change the table section of *template1.html* to the following:

```
<table>
<tr><td>Student </td><td>Roll No</td></tr>
{% for name,rollno in student_list.items %}
<tr><td>{{name}}</td><td>{{rollno}}</td></tr>
{% endfor %}
</table>
also add the following code just above the <table>
<p>{% if flag %}
      Flag is on.
{% else %}
      Flag is off.
{% endif %}
</p>
```

...and the new view function to:

```
def index(request):
   from django.shortcuts import render_to_response
   flag=False
   if request.GET.has_key('flag'):
      flag=True
   student_list = {'S1':1, 'S2':2, 'S3':3,'S4':4}
   return render_to_response('template1.html',{'student_list':student_list, 'flag':flag})
```

...where:

- *{'student_list':student_list, 'flag':flag}* is the argument we have passed to the template file.
- The *flag* variable becomes true for URL *http://127.0.0.1:8000/helloweb/?flag* and false for *http://127.0.0.1:8000/helloweb/*.
- *{% if flag %}* is a conditional block that returns the code part by checking the Boolean value of the flag variable passed to the template.
- A *{% for name,rollno in student_list.items %}* statement is used to derive rows for the table from the data passed as the dictionary to the template file. This is the same as the Python statement: for key,val in dictionary.item()

Check out the following URLs, *http://127.0.0.1:8000/helloweb/?flag* and *http://127.0.0.1:8000/helloweb/* and see how they work.

Like the above template techniques, there are lots of options facilitated by the Django template system. The best place to look for more about Django templates is the code example from the Django documentation website at *http://docs.djangoproject.com/en/dev/ref/templates/builtins/*

Databases and models

Django is designed to work with different database management systems. It provides a nice interface to interact with the database- and storage-related tasks. Django does not use traditional SQL queries to interact with the database, like most of the other Web programming frameworks. It introduces a new concept called Models.

In Django, we do not need to work with any SQL queries; the data model concept introduced by Django makes it possible to handle data in terms of objects and groups. Therefore, we have the option of object-oriented database manipulation and management through Django. We manipulate every piece of data in terms of objects.

Let's go through a simple example on how to write a Web page that deals with the MySQL database. Create a database *helloweb_db*, using MySQL or phpmyadmin, with the user name *helloweb* and password *hellowebpass*.

To run applications involving MySQL access, make sure that the *mysql-server* and *python-mysqldb* packages are already installed. Modify *settings.py* with the MySQL database details as explained in the earlier part of this article.

Django can only handle one project at a time. But it can execute many applications. Applications are sub-modules of the Django project. While deploying the Web server, we set the Web root directory to a Django project. We access all other Django applications related to that project by postfixing a path to the Web root URL.

Now we will create a Django application to learn how to code database interactive pages.

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py startapp student
slynux@hackbox:~/LFY/helloweb$ cd student
```

Now the skeleton files for the application book appear in the *book/* directory. There will be a *models.py* file. Write the following code in it:

```
from django.db import models

class StudentRegister(models.Model):
    name= models.CharField(max_length=30)
    guardian_name= models.CharField(max_length=30)
    rollno = models.IntegerField()
    admission_date = models.DateTimeField()
```

We now need to add the reference to the application we created to the *INSTALLED_APPS* tuple of the *settings.py* file—i.e., add the entry *helloweb.student* to the tuple.

To create the corresponding tables automatically in the database, issue the following command:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py syncdb
Creating table auth_permission
Creating table auth_group
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table student_studentregister

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'slynux'):
E-mail address: slynux@slynux.com
Password:
Password (again):
Superuser created successfully.
Installing index for auth.Permission model
Installing index for auth.Message model
```

You can see that besides the *student_studentregister* tables, so many other tables are also created. These are actually for the user administration option provided by Django. For every standard Web application, there will be an administration page. If you have used any content management system like Drupal, you would have seen some kind of administration page that can handle many users, the database contents, etc. Django has a nice feature — by default, it comes with basic data model administration. We can customise this to handle many users and add additional features. You can learn more about the administration interface at *http://docs.djangoproject.com/en/dev/intro/tutorial02/#activate-the-admin-site*.

You can view the SQL code used by Django internally to create tables by issuing the following command:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py sql student
BEGIN;
CREATE TABLE `student_studentregister` (
```

```
    `id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY,
    `name` varchar(30) NOT NULL,
    `guardian_name` varchar(30) NOT NULL,
    `rollno` integer NOT NULL,
    `admission_date` datetime NOT NULL
)
;
COMMIT;
```

Since the tables are already created by *syncdb*, we can work on the data storage using data model objects. For debugging and learning purposes, we can make use of the Django shell interpreter:

```
slynux@hackbox:~/LFY/helloweb$ ./manage.py shell
Python 2.6.4rc2 (r264rc2:75497, Oct 20 2009, 02:55:11)
[GCC 4.4.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from helloweb.student.models import StudentRegister
>>> student_object = StudentRegister()
>>> student_object.name = "S1"
>>> student_object.guardian_name = "G1"
>>> student_object.rollno = 1
>>> student_object.admission_date = '2009-11-01'
>>> student_object.save()
Now the object instance is saved in the table.
You can retrive the object instances for StudentRegister data
model as the following.
>>> StudentRegister.objects.all()
[<StudentRegister: StudentRegister object>]
It returns a list of all instances of StudentRegister class.

If you need to get the object instance for a particular data
entry such that rollno=4,
 >>> s =StudentRegister .objects.get(rollno=4)
If you need to remove it from database table,
>>> s.delete()
```

We will now modify the templates and views to support database interaction.

Add some more entries to the table by using *student_object = StudentRegister()* and save it using the *student_object.save()* method.

Replace the *student_list = {'S1':1, 'S2':2, 'S3':3,'S4':4} line* in the view function with the following code that grabs data from the database.

```
from helloweb.student.models import StudentRegister
slist = StudentRegister.objects.all()
student_list = {}
for s in slist:
      student_list[s.name] = s.rollno
```

This will make it populate the *student_list* dictionary with entries from the database/*StudentRegister* instances.

Run the development Web server and see the output. You can see that the table is populated with data from the database. It would be the same data you have fed through the Django shell interface.

Finally, here is a task for you: Modify the template file and the view function *index()* to view all the data from the *StudentRegister* instances—i.e., the guardian's name, admission date, etc. Also write a page *http://127.0.0.1:8000/helloweb/post_data* and a *post()* view function so that we are able to insert data into the database from a Web interface. Use the same statements we used in the Django shell interface to implement it.

This article has covered most of the bits and bytes at the basic level of Django Web application development. Once you get started coding in Django, you will definitely fall in love with it. The Django project website has excellent documentation available. You should always use *http://docs.djangoproject.com/en/1.1/* as the primary reference. END

By: Sarath Lakshman

The author is a Hacktivist of Free and Open Source Software from Kerala. He loves working on the GNU/Linux environment and contributes to the PiTiVi video editor project. He is also the developer of SLYNUX, a distro for newbies. He blogs at www.sarathlakshman.info