Table of Contents

Django Blog

Creating Web sites is fun, it's so great that we do whatever we can to learn how to do it. We developers love doing it, no matter what.

Python is a language known for being easy to learn and ridiculously powerful. If you're reading this book, it means you have already made your decision and did your research about this wonderful language.

Django is complete full stack [mega] Web framework. It provides what so ever needed for developing almost any kind of Web sites and Web Applications. In some time and not much of hassle you can have a Web application up and ready without needing to spend lots of time on reinventing the wheels.

For this book I have chosen Django Web Framework to build a Blog with common features. We assume you know basic Syntax of Python and have read Django Official Tutorial.

Provided code and snippets are simple and self-explanatory and come with comment in the code as well as some `DocString` for code documentation that can be used for future code document generation via available tools such as Sphinx and others. The way of coding would be describing *what* and *why* we're doing that implementation in plain English.

Chapter 1: Installing Python & Django

Before going further and implement our beloved Blog in Django, we need to install Python and Django. Installing both is simple and requires not much of hassle.

In this book we'll be writing our web site in **Python 2** and **Django 1.9**.

Necessary tools to get up and running are:

- Python 2
- PIP
- Django 1.9

GNU/Linux & Mac OSX

Both GNU/Linux and Mac OSX come with Python installed as default, be sure to have Python 2 installed.

To know which version of Python you have installed on your machine, you can pass `-V` parameter to python.

On my machine which is ArchLinux 64 bit, running `python -V` would print:

```
$ python -V
Python 3.5.1
```

That indicates I'm using Python 3 by default on my machine, if you're in the same situation you might have Python 2 installed as well.

You can can find out how many python version you have on your machine by looking it with `whereis` command:

```
$ whereis python
python: /usr/bin/python /usr/bin/python3.5m /usr/bin/python3.5m-config /usr/bin/python2.7
/usr/bin/python3.5 /usr/bin/python3.5-config /usr/bin/python2.7 /usr/lib/python3.5
/usr/lib/python2.7 /usr/include/python3.5m /usr/include/python2.7 /usr/share/man/man1/pyt
```

There, I have `python2` as well. Let's pass `-V` to python again and see:

```
$ python2 -V
Python 2.7.11
```

Yep, `python2` is the binary that I'm looking for.

If you don't have required Python version you can install it by using your OS package manager.

Next would be installing PIP.

PIP

Installing PIP is easy, its official documentation have done a great job to tell you how to install it.

Head over to:
https://pip.pypa.io/en/stable/installing/#do-i-need-to-install-pip

While running:

```
$ python get-pip.py
```

Be sure to use the binary of Python 2, if your `python` command is linked to Python 3, then PIP will be installed for that Python version and will download and install the packages against it.

Django

We'll be using *pip* to install Django package from PyPI.

```
$ pip install Django
```

To verify Django installation, let's check its version from command line:

```
$ django-admin.py --version
1.9.5
```

If you get anything else or an error, you might want to repeat the steps to make sure you didn't miss anything.

Summary

In this chapter we have installed:

- Python: A great general purpose language that makes every one life easier.
- PIP: Python Package Manager that install.
- Django: The heavy lifting Python Web Framework.

That made our tools for developing our Django Blog ready.

Chapter 2: Starting the Project

In this chapter we're going to star our Django Blog project and do the basic configuration so we can get to the development in next chapters.

Using `django-admin.py` , we can start our project directory and its default settings right away. Let's go ahead and fire it up and see the result:

```
$ django-admin.py startproject django_blog && cd django_blog
```

If you have **tree** installed on your machine, it should output the following result when running it inside of `django_blog` directory:

```
$ tree
.
├── django_blog
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py

1 directory, 5 files
```

Awesome!

At the moment there's nothing to touch or modify, we can just keep the current files and their structure as they are.

What we have now, is a **Django Project** that is responsible for our *settings* and holding the main *urls* and a script called `manage.py` that is required to run Django server and lots of other things.

While you're in `django_blog` directory, you can see how many useful commands are available through `manage.py` , let's try and see what kind of options we have:

```
$ ./manage.py

Type 'manage.py help <subcommand>' for help on a specific subcommand.

Available subcommands:

[auth]
    changepassword
    createsuperuser

[django]
    check
    compilemessages
    createcachetable
    dbshell
    diffsettings
    dumpdata
    flush
    inspectdb
    loaddata
    makemessages
    makemigrations
    migrate
    sendtestemail
    shell
    showmigrations
    sqlflush
    sqlmigrate
    sqlsequencereset
    squashmigrations
    startapp
    startproject
    test
    testserver

[sessions]
    clearsessions

[staticfiles]
    collectstatic
    findstatic
    runserver
```

Lots of helpers, lovely. For now we need the command `startapp` to create our blog app. Let's create an app called `blog` which we will work on this app to create our blog logic and implementation:

```
./manage.py startapp blog
```

Run `tree` to see what we have:

```
$ tree blog/
blog/
├── admin.py
├── apps.py
├── __init__.py
├── migrations
│   └── __init__.py
├── models.py
├── tests.py
└── views.py

1 directory, 7 files
```

Well, done!

Summary

In this chapter we have started our project and created our single `blog` app as well. Now, we have all the things we need to start developing our Django Blog.

Chapter 3: Make Blogging Happens

This chapter comes to be longer, since the core of Django Blog Application will be developed and it gonna be operational.

Design

Our Blog application will have a minimal features, not fancy and of course not overloaded. At its core it should comes with below features:

- Submitting Posts.
  - Drafting & Publishing.
  - Post as different Authors (users).
- An Index page to show all the post sorted by the newest.
- A Detail page to show individual Post content.
- User Friendly URLs.

Yes, that's a good way to design an application when working with Django or any kind of Web Framework.

An application should do one job and only one job and do that completely. That's why we don't have **comments** or **Visit Counts** or other features included in our Blog Application design.

Any extra features that can work independently from the blog app should get developed as an external app or package.

Models

Let's define the database structure, in Django it should live in `models.py` file.

`blog/models.py`:

```
from __future__ import unicode_literals
from django.contrib.auth.models import User
from django.utils.encoding import python_2_unicode_compatible
from django.utils.translation import gettext as _
from django.db import models


@python_2_unicode_compatible
```

```
class Post(models.Model):
    STATUS_DRAFTED = 1
    STATUS_PUBLISHED = 2

    STATUS_CHOICES = (
        (STATUS_DRAFTED, _('Drafted')),
        (STATUS_PUBLISHED, _('Published')),
    )

    title = models.CharField(verbose_name=_('title'), max_length=155)
    slug = models.SlugField(verbose_name=_('slug'), unique=True)
    content = models.TextField(verbose_name=_('content'), max_length=10000)
    user = models.ForeignKey(verbose_name=_('user'), to=User)
    status = models.CharField(
        verbose_name=_('status'),
        choices=STATUS_CHOICES,
        default=STATUS_DRAFTED
    )
    created = models.DateTimeField(
        verbose_name=_("created"),
        auto_now_add=True,
        editable=False
    )
    updated = models.DateTimeField(
        verbose_name=_('updated'),
        auto_now=True,
        editable=False,
    )

    def __str__(self):
        """
        Post title as `Post` object representative.

        :returns: Post's title
        :rtype: str
        """
        return self.title

    def is_drafted(self):
        """
        :rtype: bool
        """
        return self.status == self.STATUS_DRAFTED

    def is_published(self):
        """
        :rtype: bool
        """
        return self.status == self.STATUS_PUBLISHED

    class Meta:
        verbose_name = _('post')
        verbose_name_plural = _('posts')
```

```
        get_latest_by = "id"
        ordering = ['-id', ]
```

Model `Post` is easy to read and follow. Let's look at some good practices in this code that help you write a better Django application.

Since we're writing our code in Python 2 while using Django it's better to use:

```
from django.utils.encoding import python_2_unicode_compatible
```

Decorator `python_2_unicode_compatible` makes models with `__str__` method compatible to work with `unicode` data as well, in earlier versions of Django we would write make a method called `__unicode__` that would work with `unicode` strings as well.

Look how we started our models to be translation ready for Django:

```
from django.utils.translation import gettext as _
```

I always make my project translatable, even if I don't have any plans to add any translation in future. It's a good practice to follow while developing a software, applying it won't hurt anyone and most importantly not all the world speaks or write English or your mother language.

Aliasing `gettext` with `_` should be familiar to you, most of the other projects or languages use *underscore* `_` to mark translatable strings.

Later on, you can run user `django-admin.py` to collect and compile translation files and make the internationalization and localization of your program much easier.

As keeping creation and modification of the posts, we have created two fields called:

- `created` : Will have a default value of creation `DateTime` by passing `auto_now_add` argument. So any time a post gets created, `created` default value would be set, after saving the `Post` instance it will never get updated.
- `updated` : Get the same default `DateTime` value as `created` field but will change to current date time whenever `Post` instance get modified and saved.

Both `created` and `updated` fields cannot not be modified from admin panel, because we have passed `editable` as argument with value of `False` .

After we have defined our Blog's Post required fields, we have created two helpers as well.

- `is_published`
- `is_drafted`

These two helpers will help us later in our code base to have our conditions in the code much easier. Our view templates are the ones that would benefit a lot as well as Post Admin change list.

Admin

Now that we have our Post model ready, we need an interface to **C**reate/**R**ead/**U**pdate/**D**elete (CRUD) as an administrator.

Django comes with a powerful built-in Admin Framework to build an awesome admin panel right away with creating a simple file called `admin.py`, where we define how our Post model should be available to admin area. When starting `blog` app with `./manage.py startapp blog` it should have created an empty file for you.

Let's define blog `Post` model admin, in `admin.py`.

`blog/admin.py`:

```
from django.contrib import admin

from blog.models import Post


class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'user', 'status', 'created', )
    list_filter = ('status', 'created', 'updated', )
    search_fields = ('title', 'content', )


admin.site.register(Post, PostAdmin)
```

That's all it takes for our Post admin model. Let's go ahead and see what we have defined here.

- `list_display`: On the change list of the admin, where we see a list of all the posts, we'll be displaying some of the Post attributes such as:
  - `title`
  - `user`
  - `status`
  - `created`
- `list_filter`: Filtering and Sorting the post list is another thing that Django Admin provides. Blog Post model has some fields that our suitable for filtering our data:
  - `status`
  - `created`

    - `updated`
- `search_fields`: Now a search field will appear on the change list that we can search in our posts. So if we look for a keyword or a text, it will search our defined model fields:
    - `title`
    - `content`

Views

The public face of blog has an logic behind it, Django calls it `Views` and has a comprehensive set of tools provided to make crafting HTTP requests pretty much easy.

`blog/views.py`:

```
from django.views.generic import ListView, DetailView

from blog.models import Post


class PostListView(ListView):
    """
    Listing of all the posts.
    Paginating by 10 items per page.

    :class: PostListView
    """
    model = Post
    paginate_by = 10
    context_object_name = 'posts'


class PostDetailView(DetailView):
    """
    Getting one Post based on its `slug`.

    :class: PostDetailView
    """
    model = Post
    context_object_name = 'post'
```

There, that's our views.

Glossary

CRUD

Create, read, update and delete: Wikipedia: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

3. Chapter 3: Make Blogging Happens

PIP

The PyPA recommended tool for installing Python packages. https://pip.pypa.io

1. Chapter 1: Installing Python & Django

tree

tree - list contents of directories in a tree-like format.

2. Chapter 2: Starting the Project

Unicode

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems.