Django is a web development framework that assists in building and maintaining quality web applications. Django helps eliminate repetitive tasks making the development process an easy and time saving experience. This tutorial gives a complete understanding of Django.

MVC Pattern

When talking about applications that provides UI (web or desktop), we usually talk about MVC architecture. And as the name suggests, MVC pattern is based on three components: Model, View, and Controller. [Check our MVC tutorial here](http://www.tutorialspoint.com/struts_2/basic_mvc_architecture.htm) to know more.

DJANGO MVC - MVT Pattern

The Model-View-Template (MVT) is slightly different from MVC. In fact the main difference between the two patterns is that Django itself takes care of the Controller part (Software Code that controls the interactions between the Model and View), leaving us with the template. The template is a HTML file mixed with Django Template Language (DTL).

The following diagram illustrates how each of the components of the MVT pattern interacts with each other to serve a user request −



The developer provides the Model, the view and the template then just maps it to a URL and Django does the magic to serve it to the user.

# Apps Life Cycle

## Create an Application

We assume you are in your project folder. In our main “myproject” folder, the same folder then manage.py −

$ python manage.py startapp myapp

You just created myapp application and like project, Django create a “myapp” folder with the application structure −

myapp/

\_\_init\_\_.py

admin.py

models.py

tests.py

views.py

* **\_\_init\_\_.py** − Just to make sure python handles this folder as a package.
* **admin.py** − This file helps you make the app modifiable in the admin interface.
* **models.py** − This is where all the application models are stored.
* **tests.py** − This is where your unit tests are.
* **views.py** − This is where your application views are.

## Get the Project to Know About Your Application

At this stage we have our "myapp" application, now we need to register it with our Django project "myproject". To do so, update INSTALLED\_APPS tuple in the settings.py file of your project (add your app name) −

INSTALLED\_APPS = (

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'myapp',

)

# Admin Interface

## Starting the Admin Interface

The Admin interface depends on the django.countrib module. To have it working you need to make sure some modules are imported in the INSTALLED\_APPS and MIDDLEWARE\_CLASSES tuples of the myproject/settings.py file.

For INSTALLED\_APPS make sure you have −

INSTALLED\_APPS = (

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'myapp',

)

For MIDDLEWARE\_CLASSES −

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',

)

Before launching your server, to access your Admin Interface, you need to initiate the database −

$ python manage.py migrate

syncdb will create necessary tables or collections depending on your db type, necessary for the admin interface to run. Even if you don't have a superuser, you will be prompted to create one.

If you already have a superuser or have forgotten it, you can always create one using the following code −

$ python manage.py createsuperuser

Now to start the Admin Interface, we need to make sure we have configured a URL for our admin interface. Open the myproject/url.py and you should have something like −

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

# Examples:

# url(r'^$', 'myproject.views.home', name = 'home'),

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

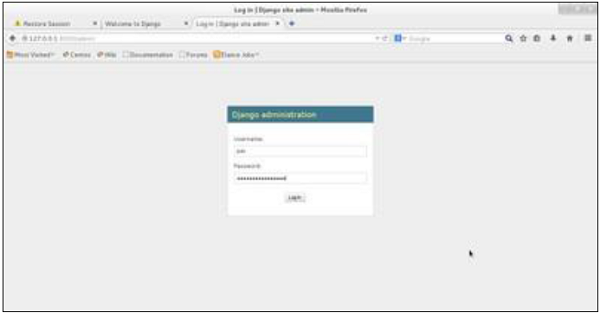
url(r'^admin/', include(admin.site.urls)),

)

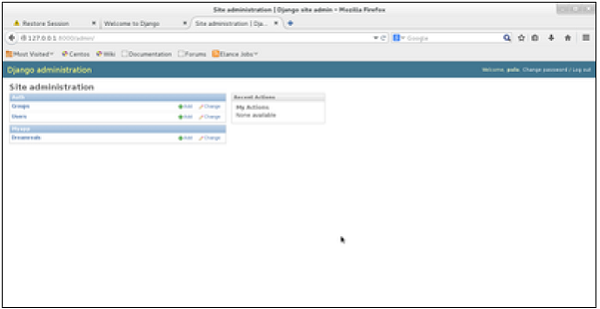
Now just run the server.

$ python manage.py runserver

And your admin interface is accessible at: http://127.0.0.1:8000/admin/



Once connected with your superuser account, you will see the following screen −



That interface will let you administrate Django groups and users, and all registered models in your app. The interface gives you the ability to do at least the "CRUD" (Create, Read, Update, Delete) operations on your models.

# Django - URL Mapping

Now that we have a working view as explained in the previous chapters. We want to access that view via a URL. Django has his own way for URL mapping and it's done by editing your project url.py file **(myproject/url.py)**. The url.py file looks like −

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

#Examples

#url(r'^$', 'myproject.view.home', name = 'home'),

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

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

)

When a user makes a request for a page on your web app, Django controller takes over to look for the corresponding view via the url.py file, and then return the HTML response or a 404 not found error, if not found. In url.py, the most important thing is the **"urlpatterns"** tuple. It’s where you define the mapping between URLs and views. A mapping is a tuple in URL patterns like −

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

#Examples

#url(r'^$', 'myproject.view.home', name = 'home'),

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

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

url(r'^hello/', 'myapp.views.hello', name = 'hello'),

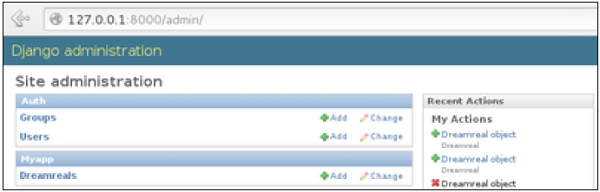
)

The marked line maps the URL "/home" to the hello view created in myapp/view.py file. As you can see above a mapping is composed of three elements −

* **The pattern** − A regexp matching the URL you want to be resolved and map. Everything that can work with the python 're' module is eligible for the pattern (useful when you want to pass parameters via url).
* **The python path to the view** − Same as when you are importing a module.
* **The name** − In order to perform URL reversing, you’ll need to use named URL patterns as done in the examples above. Once done, just start the server to access your view via :http://127.0.0.1/hello

Organizing Your URLs

So far, we have created the URLs in “myprojects/url.py” file, however as stated earlier about Django and creating an app, the best point was to be able to reuse applications in different projects. You can easily see what the problem is, if you are saving all your URLs in the “projecturl.py” file. So best practice is to create an “url.py” per application and to include it in our main projects url.py file (we included admin URLs for admin interface before).



How is it Done?

We need to create an url.py file in myapp using the following code −

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

urlpatterns = patterns('', url(r'^hello/', 'myapp.views.hello', name = 'hello'),)

Then myproject/url.py will change to the following −

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

from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',

#Examples

#url(r'^$', 'myproject.view.home', name = 'home'),

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

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

url(r'^myapp/', include('myapp.urls')),

)

We have included all URLs from myapp application. The home.html that was accessed through “/hello” is now “/myapp/hello” which is a better and more understandable structure for the web app.



Now let's imagine we have another view in myapp “morning” and we want to map it in myapp/url.py, we will then change our myapp/url.py to −

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

urlpatterns = patterns('',

url(r'^hello/', 'myapp.views.hello', name = 'hello'),

url(r'^morning/', 'myapp.views.morning', name = 'morning'),

)

This can be re-factored to −

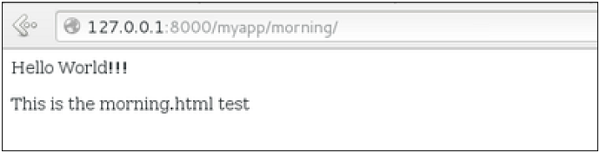
from django.conf.urls import patterns, include, url

urlpatterns = patterns('myapp.views',

url(r'^hello/', 'hello', name = 'hello'),

url(r'^morning/', 'morning', name = 'morning'),)

As you can see, we now use the first element of our **urlpatterns** tuple. This can be useful when you want to change your app name.



Sending Parameters to Views

We now know how to map URL, how to organize them, now let us see how to send parameters to views. A classic sample is the article example (you want to access an article via “/articles/article\_id”).

Passing parameters is done by capturing them with the **regexp** in the URL pattern. If we have a view like the following one in “myapp/view.py”

from django.shortcuts import render

from django.http import HttpResponse

def hello(request):

return render(request, "hello.html", {})

def viewArticle(request, articleId):

text = "Displaying article Number : %s"%articleId

return HttpResponse(text)

We want to map it in myapp/url.py so we can access it via “/myapp/article/articleId”, we need the following in “myapp/url.py” −

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

urlpatterns = patterns('myapp.views',

url(r'^hello/', 'hello', name = 'hello'),

url(r'^morning/', 'morning', name = 'morning'),

url(r'^article/(\d+)/', 'viewArticle', name = 'article'),)

When Django will see the url: “/myapp/article/42” it will pass the parameters '42' to the viewArticle view, and in your browser you should get the following result −



Note that the order of parameters is important here. Suppose we want the list of articles of a month of a year, let's add a viewArticles view. Our view.py becomes −

from django.shortcuts import render

from django.http import HttpResponse

def hello(request):

return render(request, "hello.html", {})

def viewArticle(request, articleId):

text = "Displaying article Number : %s"%articleId

return HttpResponse(text)

def viewArticle(request, month, year):

text = "Displaying articles of : %s/%s"%(year, month)

return HttpResponse(text)

The corresponding **url.py** file will look like −

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

urlpatterns = patterns('myapp.views',

url(r'^hello/', 'hello', name = 'hello'),

url(r'^morning/', 'morning', name = 'morning'),

url(r'^article/(\d+)/', 'viewArticle', name = 'article'),

url(r'^articles/(\d{2})/(\d{4})', 'viewArticles', name = 'articles'),)

Now when you go to “/myapp/articles/12/2006/” you will get 'Displaying articles of: 2006/12' but if you reverse the parameters you won’t get the same result.



To avoid that, it is possible to link a URL parameter to the view parameter. For that, our **url.py** will become −

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

urlpatterns = patterns('myapp.views',

url(r'^hello/', 'hello', name = 'hello'),

url(r'^morning/', 'morning', name = 'morning'),

url(r'^article/(\d+)/', 'viewArticle', name = 'article'),

url(r'^articles/(?P\d{2})/(?P\d{4})', 'viewArticles', name = 'articles'),)

# Template System

## The Render Function

This function takes three parameters −

* **Request** − The initial request.
* **The path to the template** − This is the path relative to the TEMPLATE\_DIRS option in the project settings.py variables.
* **Dictionary of parameters** − A dictionary that contains all variables needed in the template. This variable can be created or you can use locals() to pass all local variable declared in the view.

## Django Template Language (DTL)

Django’s template engine offers a mini-language to define the user-facing layer of the application.

### Displaying Variables

A variable looks like this: {{variable}}. The template replaces the variable by the variable sent by the view in the third parameter of the render function. Let's change our hello.html to display today’s date −

**hello.html**

<html>

<body>

Hello World!!!<p>Today is {{today}}</p>

</body>

</html>

Then our view will change to −

def hello(request):

today = datetime.datetime.now().date()

return render(request, "hello.html", {"today" : today})

We will now get the following output after accessing the URL/myapp/hello −

Hello World!!!

Today is Sept. 11, 2015

As you have probably noticed, if the variable is not a string, Django will use the \_\_str\_\_ method to display it; and with the same principle you can access an object attribute just like you do it in Python. For example: if we wanted to display the date year, my variable would be: {{today.year}}.

## Filters

They help you modify variables at display time. Filters structure looks like the following: {{var|filters}}.

**Some examples** −

* **{{string|truncatewords:80}}** − This filter will truncate the string, so you will see only the first 80 words.
* **{{string|lower}}** − Converts the string to lowercase.
* **{{string|escape|linebreaks}}** − Escapes string contents, then converts line breaks to tags.

You can also set the default for a variable.

## Tags

Tags lets you perform the following operations: if condition, for loop, template inheritance and more.

### Tag if

Just like in Python you can use if, else and elif in your template −

<html>

<body>

Hello World!!!<p>Today is {{today}}</p>

We are

{% if today.day == 1 %}

the first day of month.

{% elif today.day == 30 %}

the last day of month.

{% else %}

I don't know.

{%endif%}

</body>

</html>

In this new template, depending on the date of the day, the template will render a certain value.

### Tag for

Just like 'if', we have the 'for' tag, that works exactly like in Python. Let's change our hello view to transmit a list to our template −

def hello(request):

today = datetime.datetime.now().date()

daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

return render(request, "hello.html", {"today" : today, "days\_of\_week" : daysOfWeek})

The template to display that list using {{ for }} −

<html>

<body>

Hello World!!!<p>Today is {{today}}</p>

We are

{% if today.day == 1 %}

the first day of month.

{% elif today.day == 30 %}

the last day of month.

{% else %}

I don't know.

{%endif%}

<p>

{% for day in days\_of\_week %}

{{day}}

</p>

{% endfor %}

</body>

</html>

And we should get something like −

Hello World!!!

Today is Sept. 11, 2015

We are I don't know.

Mon

Tue

Wed

Thu

Fri

Sat

Sun

### Block and Extend Tags

A template system cannot be complete without template inheritance. Meaning when you are designing your templates, you should have a main template with holes that the child's template will fill according to his own need, like a page might need a special css for the selected tab.

Let’s change the hello.html template to inherit from a main\_template.html.

**main\_template.html**

<html>

<head>

<title>

{% block title %}Page Title{% endblock %}

</title>

</head>

<body>

{% block content %}

Body content

{% endblock %}

</body>

</html>

**hello.html**

{% extends "main\_template.html" %}

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

{% block content %}

Hello World!!!<p>Today is {{today}}</p>

We are

{% if today.day == 1 %}

the first day of month.

{% elif today.day == 30 %}

the last day of month.

{% else %}

I don't know.

{%endif%}

<p>

{% for day in days\_of\_week %}

{{day}}

</p>

{% endfor %}

{% endblock %}

In the above example, on calling /myapp/hello we will still get the same result as before but now we rely on extends and block to refactor our code −

In the main\_template.html we define blocks using the tag block. The title block will contain the page title and the content block will have the page main content. In home.html we use extends to inherit from the main\_template.html then we fill the block define above (content and title).

# Django - Models

## Creating a Model

Following is a Dreamreal model created as an example −

from django.db import models

class Dreamreal(models.Model):

website = models.CharField(max\_length = 50)

mail = models.CharField(max\_length = 50)

name = models.CharField(max\_length = 50)

phonenumber = models.IntegerField()

class Meta:

db\_table = "dreamreal"

Every model inherits from django.db.models.Model.

Our class has 4 attributes (3 CharField and 1 Integer), those will be the table fields.

The Meta class with the db\_table attribute lets us define the actual table or collection name. Django names the table or collection automatically: myapp\_modelName. This class will let you force the name of the table to what you like.

There is more field's type in django.db.models, you can learn more about them on <https://docs.djangoproject.com/en/1.5/ref/models/fields/#field-types>

After creating your model, you will need Django to generate the actual database −

$python manage.py syncdb

### Manipulating Data (CRUD)

Let's create a "crudops" view to see how we can do CRUD operations on models. Our myapp/views.py will then look like −

**myapp/views.py**

from myapp.models import Dreamreal

from django.http import HttpResponse

def crudops(request):

#Creating an entry

dreamreal = Dreamreal(

website = "www.polo.com", mail = "sorex@polo.com",

name = "sorex", phonenumber = "002376970"

)

dreamreal.save()

#Read ALL entries

objects = Dreamreal.objects.all()

res ='Printing all Dreamreal entries in the DB : <br>'

for elt in objects:

res += elt.name+"<br>"

#Read a specific entry:

sorex = Dreamreal.objects.get(name = "sorex")

res += 'Printing One entry <br>'

res += sorex.name

#Delete an entry

res += '<br> Deleting an entry <br>'

sorex.delete()

#Update

dreamreal = Dreamreal(

website = "www.polo.com", mail = "sorex@polo.com",

name = "sorex", phonenumber = "002376970"

)

dreamreal.save()

res += 'Updating entry<br>'

dreamreal = Dreamreal.objects.get(name = 'sorex')

dreamreal.name = 'thierry'

dreamreal.save()

return HttpResponse(res)

### Other Data Manipulation

Let's explore other manipulations we can do on Models. Note that the CRUD operations were done on instances of our model, now we will be working directly with the class representing our model.

Let's create a 'datamanipulation' view in **myapp/views.py**

from myapp.models import Dreamreal

from django.http import HttpResponse

def datamanipulation(request):

res = ''

#Filtering data:

qs = Dreamreal.objects.filter(name = "paul")

res += "Found : %s results<br>"%len(qs)

#Ordering results

qs = Dreamreal.objects.order\_by("name")

for elt in qs:

res += elt.name + '<br>'

return HttpResponse(res)

## Linking Models

Django ORM offers 3 ways to link models −

One of the first case we will see here is the one-to-many relationships. As you can see in the above example, Dreamreal company can have multiple online websites. Defining that relation is done by using django.db.models.ForeignKey −

**myapp/models.py**

from django.db import models

class Dreamreal(models.Model):

website = models.CharField(max\_length = 50)

mail = models.CharField(max\_length = 50)

name = models.CharField(max\_length = 50)

phonenumber = models.IntegerField()

online = models.ForeignKey('Online', default = 1)

class Meta:

db\_table = "dreamreal"

class Online(models.Model):

domain = models.CharField(max\_length = 30)

class Meta:

db\_table = "online"

As you can see in our updated myapp/models.py, we added the online model and linked it to our Dreamreal model.

Let's check how all of this is working via manage.py shell −

First let’s create some companies (Dreamreal entries) for testing in our Django shell −

$python manage.py shell

>>> from myapp.models import Dreamreal, Online

>>> dr1 = Dreamreal()

>>> dr1.website = 'company1.com'

>>> dr1.name = 'company1'

>>> dr1.mail = 'contact@company1'

>>> dr1.phonenumber = '12345'

>>> dr1.save()

>>> dr2 = Dreamreal()

>>> dr1.website = 'company2.com'

>>> dr2.website = 'company2.com'

>>> dr2.name = 'company2'

>>> dr2.mail = 'contact@company2'

>>> dr2.phonenumber = '56789'

>>> dr2.save()

Now some hosted domains −

>>> on1 = Online()

>>> on1.company = dr1

>>> on1.domain = "site1.com"

>>> on2 = Online()

>>> on2.company = dr1

>>> on2.domain = "site2.com"

>>> on3 = Online()

>>> on3.domain = "site3.com"

>>> dr2 = Dreamreal.objects.all()[2]

>>> on3.company = dr2

>>> on1.save()

>>> on2.save()

>>> on3.save()

Accessing attribute of the hosting company (Dreamreal entry) from an online domain is simple −

>>> on1.company.name

And if we want to know all the online domain hosted by a Company in Dreamreal we will use the code −

>>> dr1.online\_set.all()

To get a QuerySet, note that all manipulating method we have seen before (filter, all, exclude, order\_by....)

You can also access the linked model attributes for filtering operations, let's say you want to get all online domains where the Dreamreal name contains 'company' −

>>> Online.objects.filter(company\_\_name\_\_contains = 'company'

**Note** − That kind of query is just supported for SQL DB. It won’t work for non-relational DB where joins doesn’t exist and there are two '\_'.

But that's not the only way to link models, you also have OneToOneField, a link that guarantees that the relation between two objects is unique. If we used the OneToOneField in our example above, that would mean for every Dreamreal entry only one Online entry is possible and in the other way to.

# Page Redirection

Page redirection is needed for many reasons in web application. You might want to redirect a user to another page when a specific action occurs, or basically in case of error. For example, when a user logs in to your website, he is often redirected either to the main home page or to his personal dashboard. In Django, redirection is accomplished using the 'redirect' method.

The 'redirect' method takes as argument: The URL you want to be redirected to as string A view's name.

The myapp/views looks like the following so far −

def hello(request):

today = datetime.datetime.now().date()

daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

return render(request, "hello.html", {"today" : today, "days\_of\_week" : daysOfWeek})

def viewArticle(request, articleId):

""" A view that display an article based on his ID"""

text = "Displaying article Number : %s" %articleId

return HttpResponse(text)

def viewArticles(request, year, month):

text = "Displaying articles of : %s/%s"%(year, month)

return HttpResponse(text)

Let's change the hello view to redirect to djangoproject.com and our viewArticle to redirect to our internal '/myapp/articles'. To do so the myapp/view.py will change to −

from django.shortcuts import render, redirect

from django.http import HttpResponse

import datetime

# Create your views here.

def hello(request):

today = datetime.datetime.now().date()

daysOfWeek = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

return redirect("https://www.djangoproject.com")

def viewArticle(request, articleId):

""" A view that display an article based on his ID"""

text = "Displaying article Number : %s" %articleId

return redirect(viewArticles, year = "2045", month = "02")

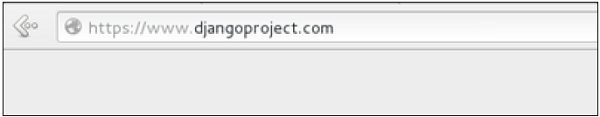
def viewArticles(request, year, month):

text = "Displaying articles of : %s/%s"%(year, month)

return HttpResponse(text)

In the above example, first we imported redirect from django.shortcuts and for redirection to the Django official website we just pass the full URL to the 'redirect' method as string, and for the second example (the viewArticle view) the 'redirect' method takes the view name and his parameters as arguments.

Accessing /myapp/hello, will give you the following screen −



And accessing /myapp/article/42, will give you the following screen −



It is also possible to specify whether the 'redirect' is temporary or permanent by adding permanent = True parameter. The user will see no difference, but these are details that search engines take into account when ranking of your website.

Also remember that 'name' parameter we defined in our url.py while mapping the URLs −

url(r'^articles/(?P\d{2})/(?P\d{4})/', 'viewArticles', name = 'articles'),

That name (here article) can be used as argument for the 'redirect' method, then our viewArticle redirection can be changed from −

def viewArticle(request, articleId):

""" A view that display an article based on his ID"""

text = "Displaying article Number : %s" %articleId

return redirect(viewArticles, year = "2045", month = "02")

**To** −

def viewArticle(request, articleId):

""" A view that display an article based on his ID"""

text = "Displaying article Number : %s" %articleId

return redirect(articles, year = "2045", month = "02")

State Management

# Cookies Handling

from django.template import RequestContext

def login(request):

username = "not logged in"

if request.method == "POST":

#Get the posted form

MyLoginForm = LoginForm(request.POST)

if MyLoginForm.is\_valid():

username = MyLoginForm.cleaned\_data['username']

else:

MyLoginForm = 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

Let’s now create a formView for the login form, where we won’t display the form if cookie is set and is not older than 10 second −

def formView(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 < 10:

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

else:

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

else:

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

url

from django.conf.urls import patterns, url

from django.views.generic import TemplateView

urlpatterns = patterns('myapp.views',

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

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

# Sessions

## Setting Up Sessions

In Django, enabling session is done in your project **settings.py**, by adding some lines to the **MIDDLEWARE\_CLASSES** and the **INSTALLED\_APPS**options. This should be done while creating the project, but it's always good to know, so **MIDDLEWARE\_CLASSES** should have −

'django.contrib.sessions.middleware.SessionMiddleware'

And **INSTALLED\_APPS** should have −

'django.contrib.sessions'

By default, Django saves session information in database (django\_session table or collection), but you can configure the engine to store information using other ways like: in **file** or in **cache**.

When session is enabled, every request (first argument of any view in Django) has a session (dict) attribute.

Let's create a simple sample to see how to create and save sessions. We have built a simple login system before (see Django form processing chapter and Django Cookies Handling chapter). Let us save the username in a cookie so, if not signed out, when accessing our login page you won’t see the login form. Basically, let's make our login system we used in Django Cookies handling more secure, by saving cookies server side.

For this, first lets change our login view to save our username cookie server side −

def login(request):

username = 'not logged in'

if request.method == 'POST':

MyLoginForm = LoginForm(request.POST)

if MyLoginForm.is\_valid():

username = MyLoginForm.cleaned\_data['username']

request.session['username'] = username

else:

MyLoginForm = LoginForm()

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

Then let us create formView view for the login form, where we won’t display the form if cookie is set −

def formView(request):

if request.session.has\_key('username'):

username = request.session['username']

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

else:

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

Now let us change the url.py file to change the url so it pairs with our new view −

from django.conf.urls import patterns, url

from django.views.generic import TemplateView

urlpatterns = patterns('myapp.views',

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

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

Let's create a simple logout view that erases our cookie.

def logout(request):

try:

del request.session['username']

except:

pass

return HttpResponse("<strong>You are logged out.</strong>")

### Some More Possible Actions Using Sessions

We have seen how to store and access a session, but it's good to know that the session attribute of the request have some other useful actions like −

* **set\_expiry (*value*)** − Sets the expiration time for the session.
* **get\_expiry\_age()** − Returns the number of seconds until this session expires.
* **get\_expiry\_date()** − Returns the date this session will expire.
* **clear\_expired()** − Removes expired sessions from the session store.
* **get\_expire\_at\_browser\_close()** − Returns either True or False, depending on whether the user’s session cookies have expired when the user’s web browser is closed.

# Caching

To cache something is to save the result of an expensive calculation, so that you don’t perform it the next time you need it. Following is a pseudo code that explains how caching works −

given a URL, try finding that page in the cache

if the page is in the cache:

return the cached page

else:

generate the page

save the generated page in the cache (for next time)

return the generated page

Django comes with its own caching system that lets you save your dynamic pages, to avoid calculating them again when needed. The good point in Django Cache framework is that you can cache −

* The output of a specific view.
* A part of a template.
* Your entire site.

To use cache in Django, first thing to do is to set up where the cache will stay. The cache framework offers different possibilities - cache can be saved in database, on file system or directly in memory. Setting is done in the **settings.py** file of your project.

## Setting Up Cache in Database

Just add the following in the project settings.py file −

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.db.DatabaseCache',

'LOCATION': 'my\_table\_name',

}

}

For this to work and to complete the setting, we need to create the cache table 'my\_table\_name'. For this, you need to do the following −

python manage.py createcachetable

## Setting Up Cache in File System

Just add the following in the project settings.py file −

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.filebased.FileBasedCache',

'LOCATION': '/var/tmp/django\_cache',

}

}

## Setting Up Cache in Memory

This is the most efficient way of caching, to use it you can use one of the following options depending on the Python binding library you choose for the memory cache −

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',

'LOCATION': '127.0.0.1:11211',

}

}

**Or**

CACHES = {

'default': {

'BACKEND': 'django.core.cache.backends.memcached.MemcachedCache',

'LOCATION': 'unix:/tmp/memcached.sock',

}

}

## Caching the Entire Site

The simplest way of using cache in Django is to cache the entire site. This is done by editing the MIDDLEWARE\_CLASSES option in the project settings.py. The following need to be added to the option −

MIDDLEWARE\_CLASSES += (

'django.middleware.cache.UpdateCacheMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.cache.FetchFromCacheMiddleware',

)

Note that the order is important here, Update should come before Fetch middleware.

Then in the same file, you need to set −

CACHE\_MIDDLEWARE\_ALIAS – The cache alias to use for storage.

CACHE\_MIDDLEWARE\_SECONDS – The number of seconds each page should be cached.

## Caching a View

If you don’t want to cache the entire site you can cache a specific view. This is done by using the **cache\_page** decorator that comes with Django. Let us say we want to cache the result of the **viewArticles** view −

from django.views.decorators.cache import cache\_page

@cache\_page(60 \* 15)

def viewArticles(request, year, month):

text = "Displaying articles of : %s/%s"%(year, month)

return HttpResponse(text)

As you can see **cache\_page** takes the number of seconds you want the view result to be cached as parameter. In our example above, the result will be cached for 15 minutes.

**Note** − As we have seen before the above view was map to −

urlpatterns = patterns('myapp.views',

url(r'^articles/(?P<month>\d{2})/(?P<year>\d{4})/', 'viewArticles', name = 'articles'),)

Since the URL is taking parameters, each different call will be cached separately. For example, request to /myapp/articles/02/2007 will be cached separately to /myapp/articles/03/2008.

Caching a view can also directly be done in the url.py file. Then the following has the same result as the above. Just edit your myapp/url.py file and change the related mapped URL (above) to be −

urlpatterns = patterns('myapp.views',

url(r'^articles/(?P<month>\d{2})/(?P<year>\d{4})/',

cache\_page(60 \* 15)('viewArticles'), name = 'articles'),)

And, of course, it's no longer needed in myapp/views.py.

## Caching a Template Fragment

You can also cache parts of a template, this is done by using the **cache** tag. Let's take our **hello.html** template −

{% extends "main\_template.html" %}

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

{% block content %}

Hello World!!!<p>Today is {{today}}</p>

We are

{% if today.day == 1 %}

the first day of month.

{% elif today == 30 %}

the last day of month.

{% else %}

I don't know.

{%endif%}

<p>

{% for day in days\_of\_week %}

{{day}}

</p>

{% endfor %}

{% endblock %}

And to cache the content block, our template will become −

{% load cache %}

{% extends "main\_template.html" %}

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

{% cache 500 content %}

{% block content %}

Hello World!!!<p>Today is {{today}}</p>

We are

{% if today.day == 1 %}

the first day of month.

{% elif today == 30 %}

the last day of month.

{% else %}

I don't know.

{%endif%}

<p>

{% for day in days\_of\_week %}

{{day}}

</p>

{% endfor %}

{% endblock %}

{% endcache %}

# Ajax

Ajax essentially is a combination of technologies that are integrated together to reduce the number of page loads. We generally use Ajax to ease end-user experience. Using Ajax in Django can be done by directly using an Ajax library like JQuery or others. Let's say you want to use JQuery, then you need to download and serve the library on your server through Apache or others. Then use it in your template, just like you might do while developing any Ajax-based application.

Another way of using Ajax in Django is to use the Django Ajax framework. The most commonly used is django-dajax which is a powerful tool to easily and super-quickly develop asynchronous presentation logic in web applications, using Python and almost no JavaScript source code. It supports four of the most popular Ajax frameworks: Prototype, jQuery, Dojo and MooTools.

Using Django-dajax

First thing to do is to install django-dajax. This can be done using easy\_install or pip −

$ pip install django\_dajax

$ easy\_install django\_dajax

This will automatically install django-dajaxice, required by django-dajax. We then need to configure both dajax and dajaxice.

Add dajax and dajaxice in your project settings.py in INSTALLED\_APPS option −

INSTALLED\_APPS += (

'dajaxice',

'dajax'

)

Make sure in the same settings.py file, you have the following −

TEMPLATE\_LOADERS = (

'django.template.loaders.filesystem.Loader',

'django.template.loaders.app\_directories.Loader',

'django.template.loaders.eggs.Loader',

)

TEMPLATE\_CONTEXT\_PROCESSORS = (

'django.contrib.auth.context\_processors.auth',

'django.core.context\_processors.debug',

'django.core.context\_processors.i18n',

'django.core.context\_processors.media',

'django.core.context\_processors.static',

'django.core.context\_processors.request',

'django.contrib.messages.context\_processors.messages'

)

STATICFILES\_FINDERS = (

'django.contrib.staticfiles.finders.FileSystemFinder',

'django.contrib.staticfiles.finders.AppDirectoriesFinder',

'dajaxice.finders.DajaxiceFinder',

)

DAJAXICE\_MEDIA\_PREFIX = 'dajaxice'

Now go to the myapp/url.py file and make sure you have the following to set dajax URLs and to load dajax statics js files −

from dajaxice.core import dajaxice\_autodiscover, dajaxice\_config

from django.contrib.staticfiles.urls import staticfiles\_urlpatterns

from django.conf import settings

Then dajax urls:

urlpatterns += patterns('',

url(r'^%s/' % settings.DAJAXICE\_MEDIA\_PREFIX, include('dajaxice.urls')),)

urlpatterns += staticfiles\_urlpatterns()

Let us create a simple form based on our Dreamreal model to store it, using Ajax (means no refresh).

At first, we need our Dreamreal form in myapp/form.py.

class DreamrealForm(forms.Form):

website = forms.CharField(max\_length = 100)

name = forms.CharField(max\_length = 100)

phonenumber = forms.CharField(max\_length = 50)

email = forms.CharField(max\_length = 100)

Then we need an ajax.py file in our application: myapp/ajax.py. That's where is our logic, that's where we put the function that will be saving our form then return the popup −

from dajaxice.utils import deserialize\_form

from myapp.form import DreamrealForm

from dajax.core import Dajax

from myapp.models import Dreamreal

@dajaxice\_register

def send\_form(request, form):

dajax = Dajax()

form = DreamrealForm(deserialize\_form(form))

if form.is\_valid():

dajax.remove\_css\_class('#my\_form input', 'error')

dr = Dreamreal()

dr.website = form.cleaned\_data.get('website')

dr.name = form.cleaned\_data.get('name')

dr.phonenumber = form.cleaned\_data.get('phonenumber')

dr.save()

dajax.alert("Dreamreal Entry %s was successfully saved." %

form.cleaned\_data.get('name'))

else:

dajax.remove\_css\_class('#my\_form input', 'error')

for error in form.errors:

dajax.add\_css\_class('#id\_%s' % error, 'error')

return dajax.json()

Now let's create the dreamreal.html template, which has our form −

<html>

<head></head>

<body>

<form action = "" method = "post" id = "my\_form" accept-charset = "utf-8">

{{ form.as\_p }}

<p><input type = "button" value = "Send" onclick = "send\_form();"></p>

</form>

</body>

</html>

Add the view that goes with the template in myapp/views.py −

def dreamreal(request):

form = DreamrealForm()

return render(request, 'dreamreal.html', locals())

Add the corresponding URL in myapp/urls.py −

url(r'^dreamreal/', 'dreamreal', name = 'dreamreal'),

Now let's add the necessary in our template to make the Ajax work −

At the top of the file add −

{% load static %}

{% load dajaxice\_templatetags %}

And in the <head> section of our dreamreal.html template add −

We are using the JQuery library for this example, so add −

<script src = "{% static '/static/jquery-1.11.3.min.js' %}"

type = "text/javascript" charset = "utf-8"></script>

<script src = "{% static '/static/dajax/jquery.dajax.core.js' %}"></script>

The Ajax function that will be called on click −

<script>

function send\_form(){

Dajaxice.myapp.send\_form(Dajax.process,{'form':$('#my\_form').serialize(true)});

}

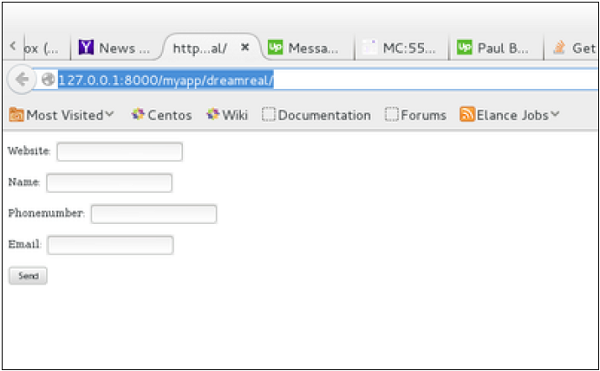
</script>

Note that you need the “jquery-1.11.3.min.js” in your static files directory, and also the jquery.dajax.core.js. To make sure all dajax static files are served under your static directory, run −

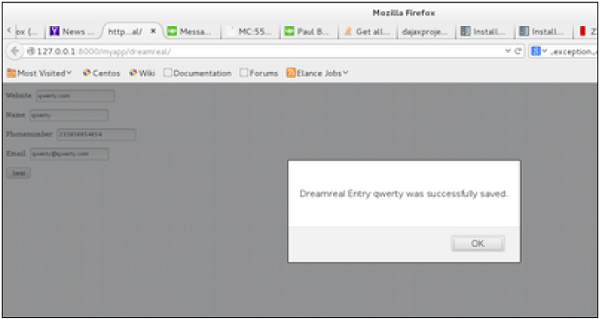
$python manage.py collectstatic

**Note** − Sometimes the jquery.dajax.core.js can be missing, if that happens, just download the source and take that file and put it under your static folder.

You will get to see the following screen, upon accessing /myapp/dreamreal/ −



On submit, you will get the following screen −



Ajax Call :

def likePost(request):

if request.method == 'GET':

post\_id = request.GET['post\_id']

likedpost = Post.obejcts.get(pk=post\_id) #getting the liked posts

m = Like(post=likedpost) # Creating Like Object

m.save()  # saving it to store in database

return HttpResponse("Success!") # Sending an success response

else:

return HttpResponse("Request method is not a GET")

**Create URLs:**

To create urls, open django\_example/urls.py. Your django\_example/urls.py should look something like this:

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

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

url(r'^', include('post.urls')),   # To make post app available at /

]

To create urls, create file post/urls.py. Your post/urls.py should look something like this:

from django.conf.urls import url

from . import views

urlpatterns = [

url(r'^/$', views.index, name='index'),  # index view at /

url(r'^likepost/$', views.likePost, name='likepost'),   # likepost view at /likepost

]

**Making templates and carrying out ajax request:**

* Create a file post/templates/post/index.html. Code sample:

<!DOCTYPE html>

<html>

<head>

<title>Like Post App</title>

</head>

<body>

<p id="message"></p>

{% for post in posts %}

<h3>{{ forloop.counter }}) {{ post.post\_heading }}</h3>

<p>{{ post.post\_text }} </p>

<a class="likebutton" id="like{{post.id}}" href="#" data-catid="{{ post.id }}">Like</a>

{% endfor %}

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.0/jquery.min.js"></script>

<script type="text/javascript">

$('.likebutton').click(function(){

var catid;

catid = $(this).attr("data-catid");

$.ajax(

{

type:"GET",

url: "/likepost",

data:{

post\_id: catid

},

success: function( data )

{

$( '#like'+ catid ).remove();

$( '#message' ).text(data);

}

})

});

</script>

</body>

</html>

Basically, what we are doing here is - we are making an ajax get request -> /likepost?post\_id=<id\_of\_liked\_post>