**How to set up a REST Service or a Web Application in Django.**

**Introduction:**

Django is a very versatile framework, primarily used for developing web applications, but it is also widely used in creating mobile app backends and REST APIs, among other things. Here we will take a quick look at how to make these things possible using Django.

As you all probably know that REST stands for “Representational State Transfer”. Basically what happens is that a user (or some software agent on one end) provides some input (or performs some activity) and the result of those inputs (or activities) are sent to the server side using a protocol that allows a request to be sent to the server (in most cases HTTP protocol is used, but it doesn't need to be HTTP, as long as it can support a request/response scheme). The server, on receiving the request, makes appropriate changes to the state of the system (hence we call it “State Transfer”).

For an introductory understanding of RESTful web services, you may take a look at the content pointed to by <https://developer.ibm.com/articles/ws-restful/>

**A Brief Discussion on How to Create a Django Project:**

Django runs on HTTP protocol, and hence it is very handy in creating a REST API or a RESTful web service. It allows the user to provode inputs on a web interface (or any interface that is capable of sending a HTTP request object to the Django application). In our examples in this document, we will be considering web application interfaces, but you can easily extend them to a mobile app or any other app that can send a similar HTTP request.

In order to understand how to do these things, we need to know a bit of Django here. This is my first post in a series of 5 posts on the same topic, so in this document, I will have you acquainted with certain django files that are of utmost importance, along with the basic structure of a Django app.

A 'Django project' is composed of a number of 'Django applications' that are stitched together with some logic. The first thing that you do when creating a Django project is to install Django in your python environment. This is usually done by running the command “pip install Django==<version number>”. You may skip the version number part, in which case the latest version of Django that can be handled by your OS and your python distro installed on your system will be installed. Fair enough.

*Note: I would be explaining things with the assumption that we are working on a linux/unix system. In case you have an Apple (mac) or Microsoft (windows) OS, please adjust your commands and paths appropriately.*

Once the installation is successfully complete, you will find a file named “django-admin” in the path of your computer. (Run “which django-admin” to see where it exists, on my system it is in “/usr/bin/django-admin”). To create a project, you need to execute the following command in a directory of your choice. Preferably create a new directory, 'cd' into it and run the following command:

*#> django-admin startproject <your project name>*

Your project name has be to provided to create the django project. For example, let us assume I am creating a project named “testyard”. In order to create my project, I need to execute the following command:

*#> django-admin startproject testyard*

The above command creates the following directory structure in the directory where you executed the above mentioned command.

testyard/

manage.py

testyard/

\_\_init\_\_.py

settings.py

urls.py

wsgi.py

**A Discussion on the Files Created Above:**

The first file, manage.py, is very important. This is the file that will eventually assist you to create applications, run the Django test web server, run Django management tasks (not covered in this document, but we will take look at it in a later article), and a host of other activities. The \_\_init\_\_.py file in the next level ensures that we will be working in an object oriented environment. Next comes the 2 most important files: settings.py and urls.py. We will discuss settings.py file first and urls.py next.

settings.py : This file contains the settings for the Django applications to work. Some of the more important config params in this file are the Database connection parameters (username, password, database name, DB host, port,Engine, etc.), static files location, Template Loaders, Middleware classes (we will be dicussing this in another article, but just keep in mind that these are the programs that interact with the request object before they reach the “view” functions, so they are capable of changing the request before it can be processed), installed apps, root urlconf (we will discuss this in this article later), etc.

You are also free to write your own application specific settings file in the same location. However, you need to put a different name to it and then import settings.py in it. There are a load of other parameters in settings.py and I would request you to go through a settings.py file in the location “https://github.com/supmit13/python-ff-ext/blob/master/urldbapp/settings.py” and figure out the parameters.

urls.py: This file contains the map of the path requested by a client (browser or a bot) to the specific view function that needs to be called when the given path is requested by an HTTP request. You will find a sample of such a urls.py file here: “<https://github.com/supmit13/python-ff-ext/blob/master/urldbapp/urls.py>”. This file is also known as the “urlconf”.

The file named wsgi.py exists to allow the applications you are creating to run on uwsgi and nginx (the production environment stuff). We will be taking a look at this file later in this article.

Once these files have been created, you start creating applications inside the project. But first, you need to figure out if the previous procedure is working fine with a server. So for that purpose, you need to run the development server of the application

To do this, you need to run the following command on the command prompt. Note that in order to run this, you need to be in the same directory where “manage.py” exists.

*#>python manage.py runserver 0.0.0.0:8080*

This command will start the development server. Now, you should be able to go to a browser, type in localhost:8080/login/ , and you should be able to see the login screen if the urls.py has an entry for that url path. Alternatively, you may just type localhost:8080/ to see if your django server is running.

So that makes you capable enough to start creating a REST API using python/Django. In the subsequent sections we will demonstrate how to write code to create the REST API.

The views.py file: Django follows a variation of the the MVC design pattern, and it is normally referred to as the MVT pattern. The 'M' stands for the model (the DB schema), V stands for views.py, which in an MVC framework is the controller part, while the 'T' stands for templates, whiich in MVC framework would be the view component.

Since we will be looking at the REST API first, we will concentrate on the views.py. First, have a look at the handlers.py at the following link: “<https://github.com/supmit13/python-ff-ext/blob/master/urldbapp/savewhatyoulookedat/handlers.py>”. In this program, the code that is to be in views.py has been put into handlers.py, but for all practical purposes, they work in the same manner. You may consider the code in handlers.py to be that of views.py in this example. (Actually, this was some code I wrote long back when I was just starting to dabbling in django. Hence I made something to see if django is flexible enough, and found that it sure was flexible).

**The Structure and Behaviour of Django:**

The “views.py” file will contain one or more functions (note that it may also contain classes, and we will take a look at classes in views.py in another post), and the functions will contain only one argument: “request”. The “request” is basically an instance of the HttpRequest class (defined by Django). All view functions in a Django App consume an instance of this HttpRequest class and return an instance of the HttpResponse class.

Why do we need to have multiple functions in the views.py file? This is because, you might want to serve multiple pages or responses, each of which has a different url path. For example, you might want to serve a login page, a registration page, an activity to check the login credentials of the user who is trying to login into your app, and a dashboard page. Each of these pages will have a different URL path, say, <http://mywebsite.com/login> for login page, <http://mywebsite.com/registration> for registration page, and so on. So each of these URLs will need a separate function to handle the request. Hence, we need one function for each of these actions in the views.py file.

How do we associate each of the activities mentioned above with a specific views.py function? This is where the urls.py file comes into play. The urls.py has a map of each url path to a specific function in views.py of a particular Django app (Remember we mentioned in the begining that a Django project is composed of one or more Django apps. We will get to the apps part in a moment). A urls.py looks something like the following:

*urlpatterns = patterns('',*

*(r'^time/$', current\_datetime),*

*(r'^savewhatyoulookedat/login/$', userLogin),*

*(r'^savewhatyoulookedat/register/$', userRegister),*

*(r'^savewhatyoulookedat/$', saveURL),*

*(r'^savewhatyoulookedat/logout/$', userLogout),*

*(r'^savewhatyoulookedat/downloadPlugin/$', firefoxPluginDownload),*

*(r'^savewhatyoulookedat/managedata/$', manageData),*

*(r'^savewhatyoulookedat/commandHandler/$', executeCommand),*

*(r'^savewhatyoulookedat/showimage/$', showImage),*

*(r'^savewhatyoulookedat/search/$', searchURL),*

*# (r'^admin/doc/', include('django.contrib.admindocs.urls')),*

*# Uncomment the next line to enable the admin:*

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

*)*

Basically, as you can see, the mapping is actually between a regular expression that should match the url path and a function in the views.py file. As you create more activity handlers in your views.py file, you keep adding an entry for each of them in the urls.py file. For a Django project with multiple applications, the urls.py file might also look like the following:

*urlpatterns += patterns('',*

*url(r'^%s$'%mysettings.REGISTER\_URL, 'skillstest.Auth.views.register', name='newuser'),*

*url(r'^%s$'%mysettings.DASHBOARD\_URL, 'skillstest.views.dashboard', name='dashboard'),*

*url("%s$"%mysettings.LOGIN\_URL, 'skillstest.Auth.views.login', name='login'),*

*url(r'%s$'%mysettings.MANAGE\_TEST\_URL, 'skillstest.Tests.views.manage', name='managetests'),*

*url(r'%s$'%mysettings.CREATE\_TEST\_URL, 'skillstest.Tests.views.create', name='createtests'),*

*url(r'%s$'%mysettings.EDIT\_TEST\_URL, 'skillstest.Tests.views.edit', name='edittests'),*

*url(r'%s$'%mysettings.ABOUTUS\_URL, 'skillstest.views.aboutus', name='aboutus'),*

*url(r'%s$'%mysettings.HELP\_URL, 'skillstest.views.helpndocs', name='helpndocs'), url(r'%s$'%mysettings.CAREER\_URL, 'skillstest.views.careers', name='careers'),*

... ... ...

'Auth' and 'Tests' are the names of the applications in the “skillstest” project. Don't worry about the variables in uppercases – they are defined elsewhere and are of no consequence to our example here.

So, now let us see how to create an application inside a project. We do that by executing the following command:

python manage.py startapp <our-app-name>

For example, if our app name is “Letsplay”, then we would run

*python manage.py startapp Letsplay*

The above command creates a directory structure like the following:

*Letsplay/*

*\_\_init\_\_.py*

*admin.py*

*apps.py*

*migrations/*

*\_\_init\_\_.py*

*models.py*

*tests.py*

*views.py*

In the above structure, we will focus mostly on the views.py and the models.py files. However, we will also touch upon the others first.

The “admin.py” file is required if you want to customize your admin panel in django. Normally, if you try to access the URL “<http://localhost:8080/admin>”, you would see an admin panel. This will display you all the models you have (we will discuss models in just a bit), the config settings of your django project (in readonly mode, of course), etc.

The “apps.py” file allows the creator of the django app to put in some application specific parameters. Each app in a Django project has its own apps.py file.

The “tests.py” file allows the app creator to write tests for the app. This file needs to conform to a certain structure. It needs to define a class for a specific test case. This class needs to have a method named “setUp”, and then it should have the tests defined as methods in the class itself. Unless you are a Django purist, you won't use this file to define your tests. Normally, in the real life scenarios, we have an application created using Django, another component created using some other technology and several other components fitted together to work as a service. In such cases, we need to write tests to check the functionality of the entire scheme of things rather than just the Django part. Hence, it is almost customary to create tests as a different suite using python or some other language like perl or ruby (or whatever the tester prefers).

By and large, any application you write (in python using Django or any other language and framework), you eventually end up interacting with a database somewhere down the line. All Django apps also tend to do the same. This is where the models.py file steps in. The “models.py” file basically provides you with an ORM (Object-Relational-Mapping) scheme in the Django app. Hence for every table in your prefered database, you have a class defined for it in the models.py file. It looks something like this:

*from django.db import models*

*import os, sys, re, time, datetime*

*import inspect*

*"""*

*'Topic' is basically category or domain.*

*"""*

*class Topic(models.Model):*

*topicname = models.CharField(max\_length=150)*

*user = models.ForeignKey(User, null=False)*

*createdate = models.DateField(auto\_now=True)*

*isactive = models.BooleanField(default=True)*

*class Meta:*

*verbose\_name = "Topics Table"*

*db\_table = 'Tests\_topic'*

*def \_\_unicode\_\_(self):*

*return "%s"%(self.topicname)*

*class Subtopic(models.Model):*

*subtopicname = models.CharField(max\_length=150)*

*subtopicshortname = models.CharField(max\_length=50)*

*topic = models.ForeignKey(Topic, blank=False, null=False)*

*createdate = models.DateField(auto\_now=True)*

*isactive = models.BooleanField(default=True)*

*class Meta:*

*verbose\_name = "Subtopics Table"*

*db\_table = 'Tests\_subtopic' # Name of the table in the database*

*def \_\_unicode\_\_(self):*

*return "%s (child of %s)"%(self.subtopicname, self.topic.topicname)*

*class Session(models.Model):*

*sessioncode = models.CharField(max\_length=50, unique=True)*

*status = models.BooleanField(default=True) # Will be 'True' as soon as the user logs in, and will be 'False' when user logs out.*

*# The 'status' will automatically be set to 'False' after a predefined period. So users will need to login again after that period.*

*# The predefined value will be set in the settings file skills\_settings.py. (skills\_settings.SESSION\_EXPIRY\_LIMIT)*

*user = models.ForeignKey(User, null=False, blank=False, db\_column='userid\_id')*

*starttime = models.DateTimeField(auto\_now\_add=True) # Should be automatically set when the object is created.*

*endtime = models.DateTimeField(default=None)*

*sourceip = models.GenericIPAddressField(protocol='both', help\_text="IP of the client's/user's host")*

*istest = models.BooleanField(default=False) # Set it to True during testing the app.*

*useragent = models.CharField(max\_length=255, default="", help\_text="Signature of the browser of the client/user") # Signature of the user-agent to guess the device used by the user.*

*# This info may later be used for analytics.*

*class Meta:*

*verbose\_name = "Session Information Table"*

*db\_table = 'Auth\_session'*

*def \_\_unicode\_\_(self):*

*return self.sessioncode*

*def isauthenticated(self):*

*if self.status and self.user.active:*

*return self.user*

*else:*

*return None*

*def save(self, \*\*kwargs):*

*super(Session, self).save(kwargs)*

The attributes in the classes are the fields in the respective tables in the database. The name of the DB table is defined in the “class Meta” of each of the Topic and Subtopic classes with the attribute named “db\_table”. The database associated with these tables is defined in the settings.py file (remember when we discussed settings.py file attributes?). For the datatypes used in the models.py file, you need to look up the Django documentation as there are quite a few datatypes and relationships and they cannot be dealt with here. In fact, the documentation for them is quite substantial. However, we have used only a few of them above and they are quite self-explanatory.

Actually, Django is quite popular because of 2 reasons.

#1. It provides the developer with all the boiler plate code, so the coder doesn't need to write all the boring stuff.

#2. It provides the coder with the ORM, so retrieving or setting a value in a certain row of a specific table in the DB is quite easy. That is the “up side” of it. There is a “down side” too. When you use ORM, you do not use SQL statements, and hence if the operation is a little complex, the ORM can become quite inefficient. With SQL statements, you can do some optimization to make the statement run faster, but with ORM, there is no such possiblity. For this reason, Django offers a way out. You can create “raw” SQL statements to query your DB, but this is rarely used by most developers. You should use “raw” SQL statements only when you see that the ORM way of manipulating the DB is distinctively inefficient.

Anyway, let us now move on to the final stages of this document. This happens to be the most important stage in the creation of a REST application. We will now take a look at the views.py file. Please refer to the example code below:

*# User login handler*

*def login(request):*

*if request.method == "GET":*

*msg = None*

*if request.META.has\_key('QUERY\_STRING'):*

*msg = request.META.get('QUERY\_STRING', '')*

*if msg is not None and msg != '':*

*msg\_color = 'FF0000'*

*msg = skillutils.formatmessage(msg, msg\_color)*

*else:*

*msg = ""*

*# Display login form*

*curdate = datetime.datetime.now()*

*tmpl = get\_template("authentication/login.html")*

*c = {'curdate' : curdate, 'msg' : msg, 'register\_url' : skillutils.gethosturl(request) + "/" + mysettings.REGISTER\_URL }*

*c.update(csrf(request))*

*cxt = Context(c)*

*loginhtml = tmpl.render(cxt)*

*for htmlkey in mysettings.HTML\_ENTITIES\_CHAR\_MAP.keys():*

*loginhtml = loginhtml.replace(htmlkey, mysettings.HTML\_ENTITIES\_CHAR\_MAP[htmlkey])*

*return HttpResponse(loginhtml)*

*elif request.method == "POST":*

*username = request.POST.get('username') or ""*

*password = request.POST.get('password') or ""*

*keeploggedin = request.POST.get('keepmeloggedin') or 0*

*csrfmiddlewaretoken = request.POST.get('csrfmiddlewaretoken', "")*

*userobj = authenticate(username, password)*

*if not userobj: # Incorrect password - return user to login screen with an appropriate message.*

*message = error\_msg('1002')*

*return HttpResponseRedirect(skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_URL + "?msg=" + message)*

*else: # user will be logged in after checking the 'active' field*

*if userobj.active:*

*sessobj = Session()*

*clientip = request.META['REMOTE\_ADDR']*

*timestamp = int(time.time())*

*# timestamp will be a 10 digit string.*

*sesscode = generatesessionid(username, csrfmiddlewaretoken, clientip, timestamp.\_\_str\_\_())*

*sessobj.sessioncode = sesscode*

*sessobj.user = userobj*

*# sessobj.starttime should get populated on its own when we save this session object.*

*sessobj.endtime = None*

*sessobj.sourceip = clientip*

*if userobj.istest: # This session is being performed by a test user, so this must be a test session.*

*sessobj.istest = True*

*elif mysettings.TEST\_RUN: # This is a test run as mysettings.TEST\_RUN is set to True*

*sessobj.istest = True*

*else:*

*sessobj.istest = False*

*sessobj.useragent = request.META['HTTP\_USER\_AGENT']*

*# Now save the session...*

*sessobj.save()*

*# ... and redirect to landing page (which happens to be the profile page).*

*response = HttpResponseRedirect(skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_REDIRECT\_URL)*

*response.set\_cookie('sessioncode', sesscode)*

*response.set\_cookie('usertype', userobj.usertype)*

*return response*

*else:*

*message = error\_msg('1003')*

*return HttpResponseRedirect(skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_URL + "?msg=" + message)*

*else:*

*message = error\_msg('1001')*

*return HttpResponseRedirect(skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_URL + "?msg=" + message)*

*-------------------------------------------------------------------------------------------------------------------*

*# User registration handler*

*def register(request):*

*privs = Privilege.objects.all()*

*privileges = {}*

*for p in privs:*

*privileges[p.privname] = p.privdesc*

*if request.method == "GET": # display the registration form*

*msg = ''*

*if request.META.has\_key('QUERY\_STRING'):*

*msg = request.META.get('QUERY\_STRING', '')*

*if msg is not None and msg != '':*

*var, msg = msg.split("=")*

*for hexkey in mysettings.HEXCODE\_CHAR\_MAP.keys():*

*msg = msg.replace(hexkey, mysettings.HEXCODE\_CHAR\_MAP[hexkey])*

*msg = "<p style=\"color:#FF0000;font-size:14;font-face:'helvetica neue';font-style:bold;\">%s</p>"%msg*

*else:*

*msg = ""*

*curdate = datetime.datetime.now()*

*(username, password, password2, email, firstname, middlename, lastname, mobilenum) = ("", "", "", "", "", "", "", "")*

*tmpl = get\_template("authentication/newuser.html")*

*#c = {'curdate' : curdate, 'msg' : msg, 'login\_url' : skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_URL, 'register\_url' : skillutils.gethosturl(request) + "/" + mysettings.REGISTER\_URL, 'privileges' : privileges, 'min\_passwd\_strength' : mysettings.MIN\_ALLOWABLE\_PASSWD\_STRENGTH, }*

*c = {'curdate' : curdate, 'msg' : msg, 'login\_url' : skillutils.gethosturl(request) + "/" + mysettings.LOGIN\_URL, 'hosturl' : skillutils.gethosturl(request),\*

*'register\_url' : skillutils.gethosturl(request) + "/" + mysettings.REGISTER\_URL,\*

*'min\_passwd\_strength' : mysettings.MIN\_ALLOWABLE\_PASSWD\_STRENGTH, 'username' : username, 'password' : password, 'password2' : password2,\*

*'email' : email, 'firstname' : firstname, 'middlename' : middlename, 'lastname' : lastname, 'mobilenum' : mobilenum, \*

*'availabilityURL' : mysettings.availabilityURL, 'hosturl' : skillutils.gethosturl(request), 'profpicheight' : mysettings.PROFILE\_PHOTO\_HEIGHT, 'profpicwidth' : mysettings.PROFILE\_PHOTO\_WIDTH }*

*c.update(csrf(request))*

*cxt = Context(c)*

*registerhtml = tmpl.render(cxt)*

*for htmlkey in mysettings.HTML\_ENTITIES\_CHAR\_MAP.keys():*

*registerhtml = registerhtml.replace(htmlkey, mysettings.HTML\_ENTITIES\_CHAR\_MAP[htmlkey])*

*return HttpResponse(registerhtml)*

*elif request.method == "POST": # Process registration form data*

*username = request.POST['username']*

*password = request.POST['password']*

*password2 = request.POST['password2']*

*email = request.POST['email']*

*firstname = request.POST['firstname']*

*middlename = request.POST['middlename']*

*lastname = request.POST['lastname']*

*sex = request.POST['sex']*

*usertype = request.POST['usertype']*

*mobilenum = request.POST['mobilenum']*

*profpic = ""*

*#userprivilege = request.POST['userprivilege']*

*csrftoken = request.POST['csrfmiddlewaretoken']*

*message = ""*

*# Validate the collected data...*

*if password != password2:*

*message = error\_msg('1011')*

*elif mysettings.MULTIPLE\_WS\_PATTERN.search(username):*

*message = error\_msg('1012')*

*elif not mysettings.EMAIL\_PATTERN.search(email):*

*message = error\_msg('1013')*

*elif mobilenum != "" and not mysettings.PHONENUM\_PATTERN.search(mobilenum):*

*message = error\_msg('1014')*

*elif sex not in ('m', 'f', 'u'):*

*message = error\_msg('1015')*

*elif usertype not in ('CORP', 'CONS', 'ACAD', 'CERT'):*

*message = error\_msg('1016')*

*elif not mysettings.REALNAME\_PATTERN.search(firstname) or not mysettings.REALNAME\_PATTERN.search(lastname) or not mysettings.REALNAME\_PATTERN.search(middlename):*

*message = error\_msg('1017')*

*....*

*....*

*return HttpResponse(html)*

The above code has 2 functions, and we will discuss them shortly. But before that, please take a look at the corresponding urls.py file for these 2 above functions:

*urlpatterns = patterns('',*

*(r'^savewhatyoulookedat/login/$', login),*

*(r'^savewhatyoulookedat/register/$', register)*

*)*

As you can see above, the 'login' function will be called when you try to access the following URL from your browser (or any other web client):

[**http://localhost:8080/savewhatyoulookedat/login/**](http://localhost:8080/savewhatyoulookedat/login/)

The 'register' function will be called when you try to access the following URL:

[**http://localhost:8080/savewhatyoulookedat/*register*/**](http://localhost:8080/savewhatyoulookedat/register/)

Note how the 'request' object has been used along with some other objects that are the product of Django's ORM. For example, in the login function, there is an instance of the Session model. The DB table behind the Session model is named “Auth\_session” and it is specified in the models.py file above. Thus, whenever a user hits one of the URLs mentioned above, the view runs some DB queries and figures out what response to send to the client. This is how a RESTful application should work, and as you can see, Django really makes it easy to develop one.

**Conclusion:**

Django is a very extensive framework, and since we were discussing REST applications, I deliberately left out Django templates. I will be explaining templates in another post, but since REST apps do not always need an HTML interface, I am skipping it for now. Also, we have just touched on some of the concepts of Django, and there is not enough room to discuss all of them in detail here. I would suggest that you go through this post, try and understand as best as you can, and then take a look at the official Django documentation for more details on the topic.

Hope you find this useful.