**Django Views and Models**

**Introduction:**

The views and the models form the Controller and the Model components of the MVC architecture that Django implements. In this article, we will be taking a look at these 2 components. First, we will take a look at the way these are connected in a Django project to form a cohesive system. Next, we will see these components in more detail. And finally, we will take a look at how we can create object oriented views (or class based views) in a project.

**Views and Models – The Connection:**

In order to understand the relationship between Django Views and Models, we need to have a clear understanding of the Model-View-Controller (MVC) design pattern. The MVC design pattern works by separating the presentation logic from the data and the controller component logic. The user initiates some activity through the presentation component, and the Controller makes the necessary computations or actions using the data from the Models component. Finally, the change in the state of the system due to the user's actions on the presentation component is displayed to the user on the presentation component itself. The presentation component forms the View of the MVC design pattern. This should not be confused with the Django views.py files, as the views.py file functions actually form the controller component of the MVC design pattern.

The model component of the MVC design pattern is implemented using the “models.py” file in Django. The models.py file contains the object representation of the database tables used by the Django app. Basically, each table associated with an app is represented in models.py as a class with the fields in the table represented by class attributes. The views.py functions (controller components) create objects of these classes (each object representing a row in the actual table) and manipulate the associated data. Thus, Django provides an ORM (Object Relational Mapping) of each table using the classes in the models.py file. An example will illustrate this feature quite nicely.

Let us suppose we need to create an application named “Auth”, and the purpose of this app would be to authenticate a user (Actually Django, by default, provides an authentication mechanism for every app you create, so you probably won't be doing this practically, unless you have a very good reason to do so. For the purpose of our example, this app is fine). So, we create a class in the models.py file like so:

*class User(models.Model):*

*username = models.CharField(max\_length=20, blank=False)*

*password = models.CharField(max\_length=20, blank=False)*

*realname = models.CharField(max\_length=30, blank=True)*

*class Meta:*

*verbose\_name = "User Table"*

*db\_table = 'Auth\_user'*

For now, please don't pay attention to the syntactic details. We will go over those in the next section. But do please note that we have defined 3 attributes in the class. The attributes are 'username', 'password' and 'realname'. When you do “python manage.py makemigrations Auth” (in the appropriate directory, that is the directory in which manage.py exists, and hoping that you have already run “python manage.py migrate” to create the database and the tables that Django internally uses), Django will look at your models.py class and create the necessary table for you in the database that has already been created for your project. Please note that Django creates a single database for your entire project, and creates tables for all your apps in that project in that database. For the above case, you will get a table named “Auth\_user”, and Django will create the following fields in it: 'id' (int), 'username' (character field), 'password' (character field), and 'realname' (character field). We haven't specified the 'id' field in our class, but Django automatically includes that in any table it creates. Normally, it serves the purpose of the primary key in the table.

Normally, a Django views.py function (working as a “controller” in the MVC scheme of things) will import this class from your models.py file and use this to create an object of it using the following syntax:

*userobj = User()*

It would then refer to the class' attributes in the following way:

*userobj.username = “some name here”*

*if not userobj.password:*

*userobj.password = “defaultpassword”*

*...*

As you can see, Django does all the hard work for you behind the scenes to provide you with the interface to modify values of fields in the table in the same way by which you would change a variable in a python script. Nice and easy.

However, there is a downside to it. Most of the time, it makes the job of a programmer easier, and the programmer does not need to write down complex SQL statements to query the database. But there are times when the programmer would want to write an optimized SQL statement rather than using the SQL statement Django uses behind the scenes for her. As it happens, there is a way to do this in Django, and it is done using what is called a “raw() manager”, and it lets you write raw SQL statements like this:

*for usr in User.objects.raw(“select username from Auth\_user”):*

*# Do something here...*

You can check out the documentation for this at “<https://docs.djangoproject.com/en/2.2/topics/db/sql/>”. However, please note that you should do this in rare occasions and only when it is absolutely necessary. Normally, Django will provide you with appropriate solutions for your normal data query-ing needs.

So that is how Django models.py and views.py function in tandem. Now let us take a brief look at the entries in the models.py file. After we are done with it, we will also take a look at the contents of the views.py file and then we will sum up the information we covered here.

**A Peek into models.py**

The models.py file, as shown above, contains classes for every table in your application. Each app in a Django project contains its own models.py file, and each project would normally have a separate directory for each app. This happens by default, if you create your apps using manage.py commands. You can force Django to have a single models.py for all apps, but this is not done normally, and we will continue to follow the conventions used by Django, as it makes the life of the programmer a lot easier. After all, one would be using Django for that among other needs.

Now, let us take a look at the contents of the models.py as listed in the previous code listing. Specifically, we will be taking a look at the syntax of it. Normally, you would have a few “import” statements at the begining of the file. We have not shown that in the above listing, so we will list them here. Note that there may be other import statements apart from the set shown below, and that will depend on your requirements.

*from django.db import models*

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

Having done that, let us now focus on the class we defined in the previous listing. First, we note that there are 3 attributes – username, password, and realname. These three attributes are the fields in the table. Django provides us with a specific data type for each data type used in a database table. For example, for a character or varchar data type in a table, Django provides the CharField data type. The 3 fields mentioned above are all of this datatype. Some of the other important data types Django provides us with are “DecimalField”, “FloatField”, “IntegerField”, “BooleanField”, “DateField”, “TimeField”, “DurationField”, “TextField”, etc. You can find the exhaustive list of supported field types at the following page: [https://docs.djangoproject.com/en/2.2/ref/models/fields/#field-types](https://docs.djangoproject.com/en/2.2/ref/models/fields/" \l "field-types). The parameters in the brackets in the listing are specific to CharField type. In our case, they specify the maximum length of the string that the field can hold, and “blank=True” and “blank=False” specifies whether or not the field can have an empty (“”) value.

Next, we have a class named “meta” inside the class for the specific table. This class (meta) specifies some utility information for use with Django. It tells Django what the name of the table corresponding to the User model would be would be (using the “ *db\_table”* attribute) and what the verbose name of our model would be (using the “ *verbose\_name*” attribute). There can be other attributes in the “meta” class - “*abstract*”, meaning the model will be an abstract base class, “*get\_latest\_by”* for naming one or more fields that would be used by the model manager's “latest()” and “earliest” methods, “*ordering*” for naming one or more fields, for use when obtaining a list of objects, etc. You can see the list of attributes available for use “[here](https://docs.djangoproject.com/en/2.2/ref/models/options/)”.

**A Peek into views.py**

The functions of the Django views.py file of your app are the controller part of the MVC pattern. Every views.py function that serves a HTTP request consumes an instance of the “HttpRequest” class defined by Django. This class is defined in the “django.http” module of the Django framework along with other classes like “HttpResponse”, “HttpResponseBadRequest” and “HttpRedirect”. A views.py function will take the HttpRequest object as parameter, do whatever computation it needs to do using the attributes of that object and finally returns a HttpResponse class instance. Normally, it would load a template from the “templates” directory (or from the directory where you store your Django HTML templates that you have created before), create a “Context” object using the values computed by you in the function, render the template using the “render” function and create the HttpResponse object using the rendered content. I know this will sound like a lot of cryptic stuff, so here is an example of a views.py function to explain what I have said above.

*from django.conf import settings*

*from django.views.generic import View*

*from django.http import HttpResponseBadRequest, HttpResponse, HttpResponseRedirect*

*from django.template import Template, Context*

*from django.template.loader import get\_template*

*# More import statements here... Add your own imports too.*

*def authenticate\_user(request):*

*if request.method != "POST":*

*message = “Incorrect HTTP request method used” # or whatever message you want to send back*

*response = HttpResponseBadRequest(“/appname/badmethodpage/”)*

*return response*

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

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

*userobj = User.objects.get(username=usrname)*

*passwd = userobj.password*

*if password != passwd:*

*message = “Incorrect password”*

*response = HttpResponseRedirect(“/appname/login/?” + message)*

*return response*

*values\_dict = {'somekey1' : 'some computed value #1', 'somekey2' : 'some computed value #2'}*

*tmpl = get\_template(“auth/profile\_template.html”)*

*cxt = Context(values\_dict)*

*auth\_html = tmpl.render(cxt)*

*return HttpResponse(auth\_html)*

Let us take a closer look at the above function. First, we check out if the method used to send us this HTTP request is “POST” or not. In our case above, we are explicitly looking for “POST” requests, and we want to stop if we do not get a POST request. We can find this out by looking at the “method” attribute of the “request” object. Once we are ok with the request method, we carry on by extracting the form parameters “username” and “password”. We assume that the form that sends us this request had a field named “username” and “password”. “username” would possibly be a text input field (<input type='text' name='username'>) and “password” would possibly be a password field (<input type='password' name='password'>). Having retrieved those values, we continue by comparing the password received from the HTTP request (variable named 'password') with the password value retrieved from the database (variable 'passwd') using the username as the query parameter. If they do not match, we redirect to the login page using the following statements:

*message = “Incorrect password”*

*response = HttpResponseRedirect(“**/appname/login/?” + message)*

*return response*

However, if they do match, we take the computation forward by the process described above the function listing. The line *values\_dict = {'somekey1' : 'some computed value #1', 'somekey2' : 'some computed value #2'}* creates a dictionary, then the line *tmpl = get\_template(“auth/profile\_template.html”)* gets the template file in the variable 'tmpl', the line *cxt = Context(values\_dict)* creates the “Context” object using the dictionary of values we created above, the line *auth\_html = tmpl.render(cxt)* renders the page in a variable “auth\_html”, and finally we create a HttpResponse object using the rendered variable and return that HttpResponse object using the line *return HttpResponse(auth\_html).*

Here we have assumed that the directory containing the templates for the “Auth” app reside in a directory named “auth” under the directory specified using “TEMPLATE\_DIRS” variable in settings.py file. The “TEMPLATE\_DIRS” variable is defined in the settings.py file and the declaration looks as follows:

*TEMPLATE\_DIRS = (*

*os.path.join(PROJECT\_ROOT, "templates"),*

*)*

You might have a different value, however.

**Conclusion:**

So that should give you an idea as to what models.py and views.py files are and how they work. To sum it up, these 2 files form the heart of a Django app. The other important files that work with these two files to serve dynamic pages using Django are the “urls.py”, “settings.py” and the appropriate template file. The “urls.py” file contain a mapping of the URL path to a specific views.py function. For example, we can configure the URL path “/appname/login\_process/” to the views.py function listed above. The “settings.py” file contains a lot of variables that are global in nature and the views.py functions use them as and when necessary. The template file serves the HTML content and is populated and rendered by values from the views.py functions.

I think the above discussion would be suitable to get you started with writing models and views.py functions. For details, you must check out the official Django documentation. Good luck with Django.