# Intro

[Developed by](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+developed+by&stick=H4sIAAAAAAAAAOPgE-LUz9U3MKuML6zQUskot9JPzs_JSU0uyczP0y_OTyspTyxKtUpJLUvNyS9ITVFIqlzEKpySlZiXnq-ALAoAvJ2ulUkAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAZegQIDhAS)**:**[Django Software Foundation](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=Django+Software+Foundation&stick=H4sIAAAAAAAAAOPgE-LUz9U3MKuML6xQ4gIxi0wKzYortFQyyq30k_NzclKTSzLz8_SL89NKyhOLUq1SUstSc_ILUlMUkioXsUq5ZCXmpecrBEOlFdzyS_NSEkFaANTdHm1cAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98QmxMoATAZegQIDhAT)

[Stable release](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+stable+release&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAaegQIDhAW)**:**2.1.7 ([11 February 2019](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=11+February+2019&stick=H4sIAAAAAAAAAONgVhLQL9E3MiwsLzHNTbcorCwoWMQqYGio4JaaVFSaWFSpYGRgaAkAHsoMhycAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98QmxMoATAaegQIDhAX); 29 days ago)

[Preview release](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+preview+release&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAbegQIDhAa)**:**2.1 beta 1 ([18 June 2018](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=18+June+2018&stick=H4sIAAAAAAAAAONgVhLQL9E3Ms_Nrio2SsvLsjQxWcTKY2ih4FWal6pgZGBoAQDw85weIwAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98QmxMoATAbegQIDhAb); 8 months ago)

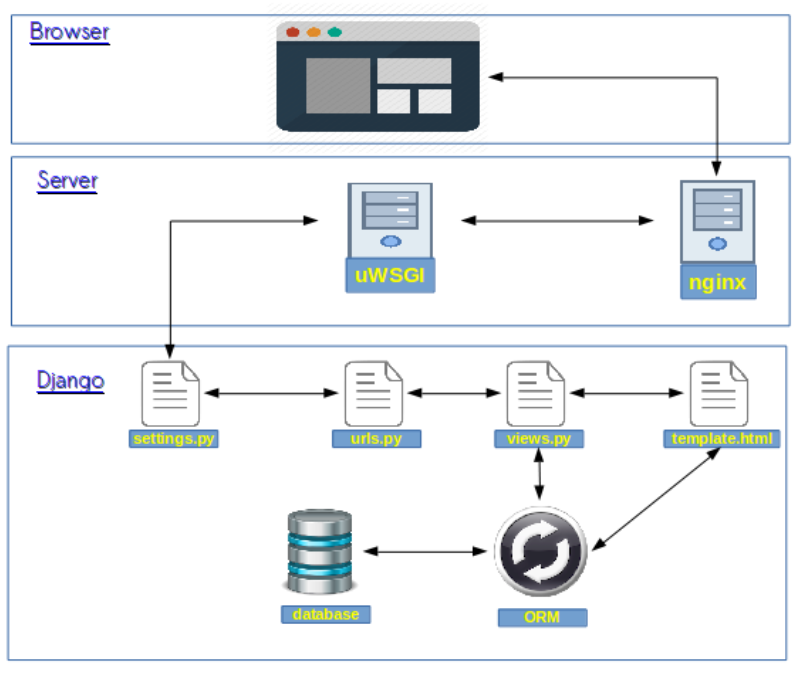
[License](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+license&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAcegQIDhAe)**:**3-clause [BSD](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=BSD&stick=H4sIAAAAAAAAAONgVuLQz9U3MLQoNlnEyuwU7AIA8g-1nRIAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98QmxMoATAcegQIDhAf)

[Size](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+size&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAdegQIDhAi)**:**7.6 MB

[Initial release date](https://www.google.com/search?rlz=1C1CHBF_enQA776QA776&q=django+initial+release+date&stick=H4sIAAAAAAAAAOPgE-LUz9U3MKuML6zQUs4ot9JPzs_JSU0uyczP0y_OTyspTyxKtUosKEhNLFJISSxJXcQqnZKVmJeer5CZl1mSmZijUJSak5pYnAqWBQCgq_PkUAAAAA&sa=X&ved=2ahUKEwiswfaMh5LhAhWR-aQKHfUCD98Q6BMoADAeegQIDhAl)**:**July 15, 2005

Django was created in the fall of 2003, when the web programmers at the **Lawrence Journal-World** newspaper, **Adrian Holovaty** and **Simon Willison**, began using Python to build applications. It was released publicly under a BSD license in July 2005. The framework was named after guitarist Django Reinhardt.

The general overview of how Django works:



# The actual steps

1. Install Python on the machine

* Windows <https://www.ics.uci.edu/~pattis/common/handouts/pythoneclipsejava/python.html>
* Mac and Create and Activate Virtual Environment <https://wsvincent.com/install-python3-mac/>
* Linux

1. Create virtual environment

* All platforms using (venv) if the Python version on machine is >= 3.3 <https://docs.python.org/3/library/venv.html>

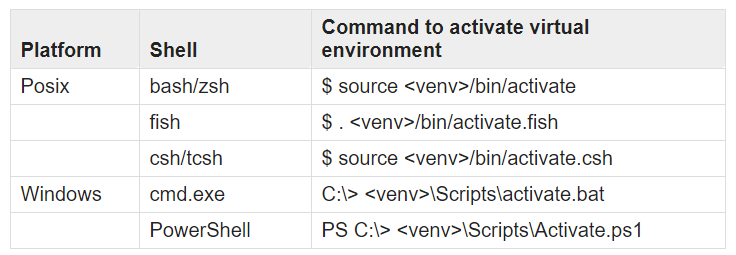
Creation use below command on linux

$ python3 -m venv /path/to/new/virtual/environment

Or on windows

c:\>python -m venv c:\path\to\myenv

Activattion of Env



pip is the preferred installer program. Starting with Python 3.4, it is included by default with the Python binary installers.

A *virtual environment* is a semi-isolated Python environment that allows packages to be installed for use by a particular application, rather than being installed system wide.

venv is the standard tool for creating virtual environments, and has been part of Python since Python 3.3. Starting with Python 3.4, it defaults to installing pip into all created virtual environments.

virtualenv is a third party alternative (and predecessor) to venv. It allows virtual environments to be used on versions of Python prior to 3.4, which either don’t provide venv at all, or aren’t able to automatically install pip into created environments. Install virtual env and

Using [virtualenv](https://packaging.python.org/key_projects/#virtualenv):

virtualenv <DIR>

source <DIR>/bin/activate

or windows

virtualenv <DIR>

<DIR>/bin/activate

1. Install pip. If someone has python version less than 3.3

NB: to update pip use ***$python -m pip install --upgrade pip***

* Linux machines <https://packaging.python.org/guides/installing-using-linux-tools/>
* Windows machines

Securely Download [get-pip.py](https://bootstrap.pypa.io/get-pip.py) [[1]](https://packaging.python.org/tutorials/installing-packages/#id7)

Run python get-pip.py. [[2]](https://packaging.python.org/tutorials/installing-packages/#id8) This will install or upgrade pip. Additionally, it will install [setuptools](https://packaging.python.org/key_projects/#setuptools) and [wheel](https://packaging.python.org/key_projects/#wheel) if they’re not installed already.

**Warning**

Be cautious if you’re using a Python install that’s managed by your operating system or another package manager. get-pip.py does not coordinate with those tools, and may leave your system in an inconsistent state. You can use python get-pip.py --prefix=/usr/local/ to install in /usr/local which is designed for locally-installed software.

While pip alone is sufficient to install from pre-built binary archives, up to date copies of the setuptools and wheel projects are useful to ensure you can also install from source archives:

python -m pip install --upgrade pip setuptools wheel

1. Create a directory where the project code is going to reside and cd there

* projectFolder

1. Use pip to install Django while using the devEnv you just made

(env)$ pip install django

1. Use the installed Django to create a **project**

(env)$ django-admin startproject ourProject

1. Cd to ourProject directory and add an Application in there using

(env)$ cd ourProject

(env)$ python manage.py startapp ourApp

1. Install DjangoREST inside **ourProject** using **pip**

(env)$ pip install djangorestframework



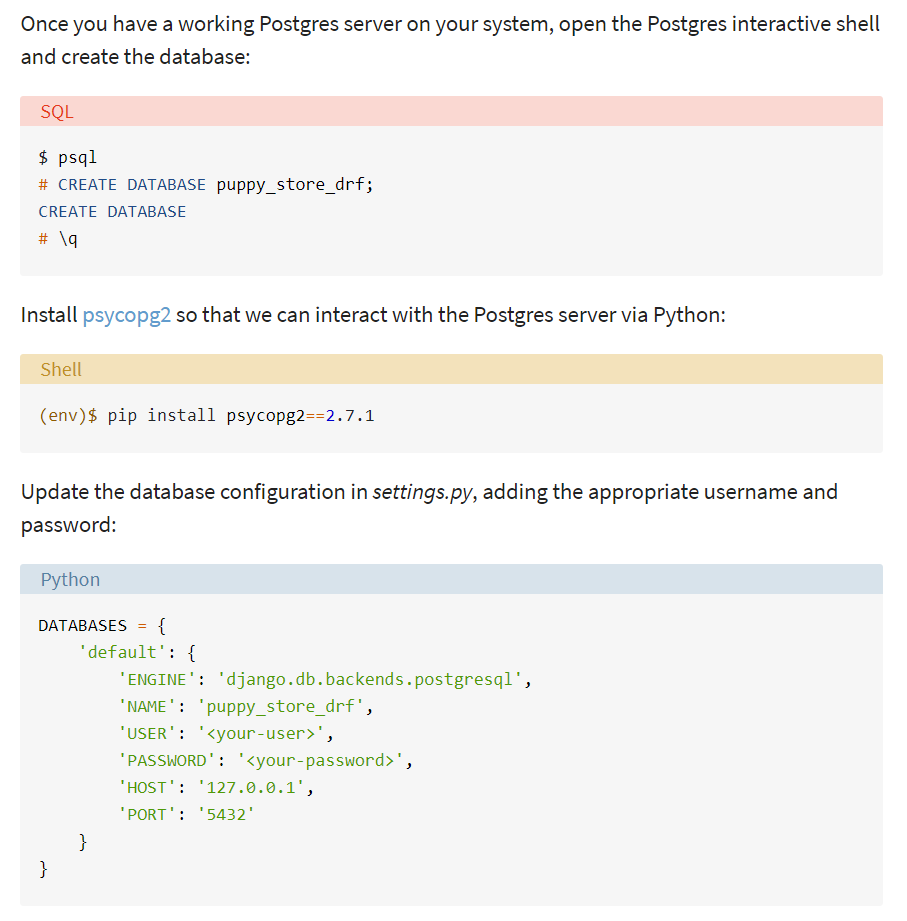
Update settings.py to also look like below

MEDIA\_URL = '/media/'

MEDIA\_ROOT = os.path.join(BASE\_DIR, 'media')

STATIC\_URL = '/static/'

STATIC\_ROOT = os.path.join(BASE\_DIR, 'static')

1. If you want to connect your Django app to an external database server you can do the setup as below: 
2. Now we can define a single model, add it to the admin dashboard and apply migrations

class Store(models.Model):

user = models.OneToOneField(

User,

on\_delete=models.CASCADE,

related\_name="store\_relation"

)

name = models.CharField(max\_length=200, default="Empty Flower Store")

logo = models.ImageField(

null=True,

blank=True,

upload\_to="stores/images/logos/"

)

banner = models.ImageField(

null=True,

blank=True,

upload\_to="stores/images/banners/"

)

is\_active = models.BooleanField(default=False)

date\_created = models.DateTimeField(auto\_now\_add=True)

rating = models.IntegerField(default=5)

class Meta:

ordering = ("date\_created",)

Add to the admin dashboard from admin.py of the app folder.

from django.contrib import admin

from ourApp.models import Consignment 🡨--import the model

# Register your models here.

admin.site.register(Consignment) 🡨-- ad dit using the regiter function

Create a Serializer to Serialize that model’s data so that it can be sent as responses in the http responses.

1. Create a file named “serializers.py” in the app folder
2. Add the **serializers.py** code below

from rest\_framework import serializers

from nonTrivialApp.models import Store

from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = (

'id',

'username',

'email',

'first\_name',

'last\_name',

)

class StoreSerializer(serializers.ModelSerializer):

#user = UserSerializer(read\_only=True)

class Meta:

model = Store

fields = (

'id',

'user',

'name',

'logo',

'banner',

'is\_active',

'date\_created',

'rating',

)

$python manage.py makemigrations

$python manage.py migrate

$python manage.py createsuperuser

Tell them the admin dashboard is available at /admin with the entered credentials

1. We can save all the packages will installed for the application to work by doing:

* $pip freeze > requirements.txt
* When running this project in on a different machine run the command

$pip install -r requirements.txt

Before launching the app.

* To see more details about an installed requirement run the command:

$pip show <packagename>

There are other ways to collect the dependent packages into the requirements file <https://medium.com/python-pandemonium/better-python-dependency-and-package-management-b5d8ea29dff1>

# Setup the Routing to point to the view functions (1)

1. Create url.py in the app folder
2. Define some of the routes of the API endpoints

from django.conf.urls import url

from nonTrivialApp import views

from django.conf import settings

from django.conf.urls.static import static

from django.contrib.auth.decorators import login\_required

from django.views.static import serve

@login\_required

def protected\_serve(request, path, document\_root=None, show\_indexes=False):

return serve(request, path, document\_root, show\_indexes)

urlpatterns = [

url(

r'^api/v1/stores/(?P<pk>[0-9]+)$',

views.get\_delete\_update\_store,

name='get\_delete\_update\_store'

),

url(

r'^api/v1/stores/$',

views.get\_post\_store,

name='get\_post\_store'

)

]

if settings.DEBUG:

urlpatterns += static(settings.MEDIA\_URL,

document\_root=settings.MEDIA\_ROOT)

1. Define the view functions in the view.py file

from django.shortcuts import render

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

# Create your views here.

# dont forget the "id" param that comes from the path.

@api\_view(['GET', 'DELETE', 'PUT'])

def get\_delete\_update\_store(request, id):

return Response({})

@api\_view(['GET', 'POST'])

def get\_post\_store(request):

return Response({})

1. Add the following 2 lines to urls.py of the project folder

from django.conf.urls import include, url

from django.contrib import admin

from django.urls import path

urlpatterns = [

path('admin/', admin.site.urls),

url(r'^', include('nonTrivialApp.urls')),

url(r'^api-auth/', include('rest\_framework.urls', namespace='rest\_framework')),

]

1. Now lets run the server again and visit the urls

* <http://127.0.0.1:9999/api/v1/stores/1>
* <http://127.0.0.1:9999/api/v1/stores/>

To explore and interact with the newly created REST API but who tests there apps manually like this??? What about Auto Tests and TDD ☺

As all can see we are not getting any data from the DB because there is no logic in the view functions to query data and serialize it.

# Before we get too carried away developing the app TESTING TESTING TESTING (2)

Django provides a testing framework for your code via “django.test” module.

So if you want to follow the TDD approach, they have the necessary tools for you. In TDD there are 2 repetitive steps:

* add a unit test, just enough code to fail
* then update the code to make it pass the test.

Once the test passes, start over with the same process for the new test.

**Steps**

1. Create a folder called “test” in the “ourApp” folder
2. Add a file named **test\_views.py** and delete **tests.py** in **ourApp** folder and also add **\_\_int\_\_.py** inside the **tests** folder

Syntax and file structure sample

import json

from rest\_framework import status

from django.test import TestCase, Client

from django.urls import reverse

from nonTrivialApp.models import Store

from nonTrivialApp.serializers import StoreSerializer

from django.contrib.auth.models import User

# initialize the APIClient app, This client will be used to send HTTP requests to the API itself and it returns the Response of the API

client = Client()

class GetAllStoresTest(TestCase):

""" Test module for GET all Stores API view function """

def setUp(self): # this is used to create the preconditions before the test

user1 = User.objects.create\_user(username='a', email='a@a.a')

Store.objects.create(

user=user1,

name='Store1',

rating=0)

user2 = User.objects.create\_user(username='b', email='b@b.b')

Store.objects.create(

user=user2,

name='Store2',

rating=0)

def test\_get\_all\_stores(self):

# get API response

response = client.get(reverse('get\_post\_stores'))

# get data from db

stores = Store.objects.all()

serializer = StoreSerializer(stores, many=True)

self.assertEqual(response.data, serializer.data)

self.assertEqual(response.status\_code, status.HTTP\_200\_OK)

1. Run $**python manage.py test**
2. The test will fail because the view function is returning an empty dictionary as its response. We want it to fail! This is the first step of our TDD cycle. Now we have to develop the view function that we are developing to pass this test. After that rinse and repeat.

Now it is time to develop the view to pass the test

# Developing the view functions and more about the ORM mapper(3)

1. Edit the view function to look like this

@api\_view(['GET', 'POST'])

def get\_post\_stores(request):

# get all stores in the system

if request.method == 'GET':

stores = Store.objects.all()

serializer = StoreSerializer(stores, many=True)

return Response(serializer.data)

# insert a new record for a store

elif request.method == 'POST':

return Response({})

1. Run the test and see that it passes the test
2. Go to <http://127.0.0.1:9999/api/v1/stores/> in your browser to see the browsable API and the data on the server
3. Look at the data that is being brought to represent the User of the store in this response. It is just the id of the user. What if we want ed to bring the data of the user embedded in this response????
4. Easy peasy, just add an UserSerializer and use it to define the Store Serializer.

from rest\_framework import serializers

from nonTrivialApp.models import Store

from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = (

'id',

'username',

'email',

'first\_name',

'last\_name',

)

class StoreSerializer(serializers.ModelSerializer):

user = UserSerializer(read\_only=True) #<<<<<this embeds the data of user

class Meta:

model = Store

fields = (

'id',

'user',

'name',

'logo',

'banner',

'is\_active',

'date\_created',

'rating',

)

* Save and revisit <http://127.0.0.1:9999/api/v1/stores/> and voila the data of the User is there.

We can also write the tests for the other API endpoints

* Let’s test the endpoint for getting a specific store
* Add this class to the test\_views.py file

class GetSingleStoreTest(TestCase):

def setUp(self):

user1 = User.objects.create\_user(username='c', email='c@c.c')

self.jumbo = Store.objects.create(

user=user1,

name='Jumbo',

rating=3)

def test\_get\_valid\_single\_store(self):

response = client.get(

reverse('get\_delete\_update\_store', kwargs={'id': self.jumbo.id})

)

store = Store.objects.get(id=self.jumbo.id)

serializer = StoreSerializer(store)

self.assertEqual(

response.data,

serializer.data,

'The response data does not match the serializer data'

)

self.assertEqual(

response.status\_code,

status.HTTP\_200\_OK,

'The response code was not 200'

)

NB: you can add custom messages to the assert function to be displayed when the assertion has failed.

* Run : $python manage.py test
* The test will fail as we would like it to because the view function for getting a single store is returning an empty dictionary.
* Now add the following code to pass the test

@api\_view(['GET', 'DELETE', 'PUT'])

def get\_delete\_update\_store(request, id):

# dont forget the "id" param that comes from the path.

# get the record that will be processed by this request if it exists

try:

store = Store.objects.get(id=id)

except Store.DoesNotExist:

return Response(status=status.HTTP\_404\_NOT\_FOUND)

# get details of a single store

if request.method == 'GET':

serializer = StoreSerializer(store)

return Response(serializer.data)

NB: you must import the status module using

from rest\_framework import status

inside views.py

Those of you who are concerned with code coverage reports there is something for you too <https://docs.djangoproject.com/en/2.1/topics/testing/advanced/#integration-with-coverage-py>

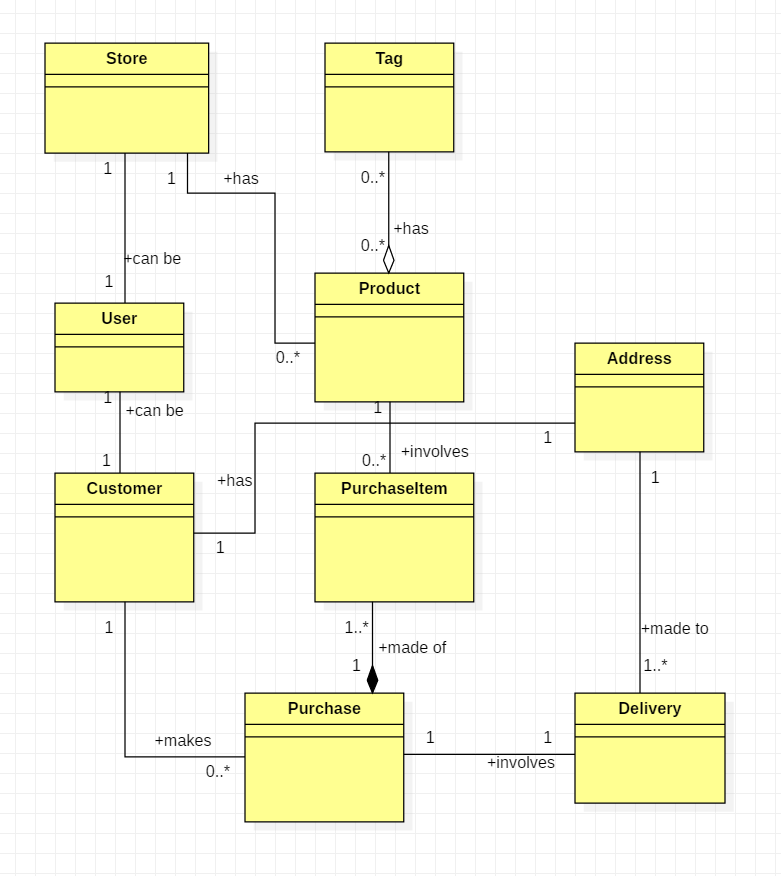
>pip install coverage

> pip freeze>requirements.txt

>coverage run --source='.' manage.py test

> coverage report

# Expanding the Models and Serializers.(4)



from django.db import models

from django.contrib.auth.models import User

# Create your models here.

class Tag(models.Model):

name = models.CharField(max\_length=200)

date\_created = models.DateTimeField(auto\_now\_add=True)

class Address(models.Model):

line\_1 = models.CharField(max\_length=200)

line\_2 = models.CharField(max\_length=200)

city = models.CharField(max\_length=200)

country = models.CharField(max\_length=200)

lat = models.FloatField(

null=True,

blank=True

)

lon = models.FloatField(

null=True,

blank=True

)

class Customer(models.Model):

user = models.OneToOneField(

User,

on\_delete=models.CASCADE,

related\_name="customer\_relation"

)

gender = models.CharField(max\_length=20)

address = models.ForeignKey(Address, on\_delete=models.CASCADE)

class Store(models.Model):

user = models.OneToOneField(

User,

on\_delete=models.CASCADE,

related\_name="store\_relation"

)

name = models.CharField(max\_length=200, default="Empty Flower Store")

logo = models.ImageField(

null=True,

blank=True,

upload\_to="stores/images/logos/"

)

banner = models.ImageField(

null=True,

blank=True,

upload\_to="stores/images/banners/"

)

is\_active = models.BooleanField(default=False)

date\_created = models.DateTimeField(auto\_now\_add=True)

rating = models.IntegerField(default=5)

class Meta:

ordering = ("date\_created",)

class Product(models.Model):

name = models.CharField(max\_length=200)

description = models.CharField(max\_length=200)

price = models.FloatField(default=0)

# for demonstrating auto many to many field using OODB modeling https://docs.djangoproject.com/en/2.1/ref/models/fields/#django.db.models.ForeignKey

tags = models.ManyToManyField(Tag, blank=True)

image = models.ImageField(

null=True,

blank=True,

upload\_to="products/images/"

)

class Purchase(models.Model):

customer = models.ForeignKey(Customer, on\_delete=models.CASCADE)

date = models.DateTimeField(auto\_now\_add=True)

class PurchaseItem(models.Model):

purchase = models.ForeignKey(Purchase, on\_delete=models.CASCADE)

product = models.ForeignKey(Product, on\_delete=models.CASCADE)

quantity = models.IntegerField()

sub\_total = models.FloatField()

class Delivery(models.Model):

purchase = models.ForeignKey(Purchase, on\_delete=models.CASCADE)

destination = models.ForeignKey(Address, on\_delete=models.CASCADE)

status = models.CharField(max\_length=200)

delivered\_on = models.DateTimeField(

null=True,

blank=True

)

Don’t forget to register them.

* Run

>python manage.py makemigrations

>python manage.py migrate

>python manage.py runserver 9999

And we have a fresh new app with the Data model that we want.

Create Serializers for them

from rest\_framework import serializers

from nonTrivialApp.models import Store, Tag, Address, Customer, Product, Purchase, PurchaseItem, Delivery

from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):

class Meta:

model = User

fields = (

'id',

'username',

'email',

'first\_name',

'last\_name',

)

class StoreSerializer(serializers.ModelSerializer):

user = UserSerializer(read\_only=True)

class Meta:

model = Store

fields = (

'id',

'user',

'name',

'logo',

'banner',

'is\_active',

'date\_created',

'rating',

)

class TagSerializer(serializers.ModelSerializer):

class Meta:

model = Tag

fields = (

'id',

'name',

'date\_created',

)

class AddressSerializer(serializers.ModelSerializer):

class Meta:

model = Address

fields = (

'id',

'line\_1',

'line\_2',

'city',

'country',

'lat',

'lon',

)

class CustomerSerializer(serializers.ModelSerializer):

class Meta:

model = Customer

fields = (

'id',

'user',

'gender',

'address',

)

class ProductSerializer(serializers.ModelSerializer):

# to get the collection of related objects in many to many relationship

tags = TagSerializer(many=True)

class Meta:

model = Product

fields = (

'id',

'name',

'description',

'price',

'tags',

'image',

)

class PurchaseSerializer(serializers.ModelSerializer):

# to get the collection of related objects in many to many relationship

customer = CustomerSerializer(read\_only=True)

class Meta:

model = Purchase

fields = (

'id',

'customer',

'date',

)

class PurchaseItemSerializer(serializers.ModelSerializer):

purchase = PurchaseSerializer(read\_only=True)

product = ProductSerializer(read\_only=True)

class Meta:

model = Purchase

fields = (

'id',

'purchase',

'product',

'quantity',

'sub\_total',

)

class DeliverySerializer(serializers.ModelSerializer):

purchase = PurchaseSerializer(read\_only=True)

class Meta:

model = Purchase

fields = (

'id',

'purchase',

'destination',

'status',

'delivered\_on',

)

# Dealing with Auto Creation of Database entries (Using Generic Class Based Views) (5)

You can use generic class based views instead of the function views we used in the beginning

We will use this method to create the endpoint dealing with Tags

* Add these 2 urls to the urls.py

url(

r'^api/v1/tags/(?P<pk>[0-9]+)/$',

views.TagDetail.as\_view(),

name='get\_post\_stores'

),

url(

r'^api/v1/tags/$',

views.TagList.as\_view(),

name='get\_post\_stores'

)

* Add these imports into views.py

from rest\_framework import mixins

from rest\_framework import generics

from nonTrivialApp.models import Store, Tag

from nonTrivialApp.serializers import StoreSerializer, TagSerializer

* Add these 2 classes to views.py too

class TagList(mixins.ListModelMixin, mixins.CreateModelMixin, generics.GenericAPIView):

queryset = Tag.objects.all()

serializer\_class = TagSerializer

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

return self.list(request, \*args, \*\*kwargs)

def post(self, request, \*args, \*\*kwargs):

return self.create(request, \*args, \*\*kwargs)

class TagDetail(mixins.RetrieveModelMixin, mixins.UpdateModelMixin, mixins.DestroyModelMixin, generics.GenericAPIView):

queryset = Tag.objects.all()

serializer\_class = TagSerializer

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

return self.retrieve(request, \*args, \*\*kwargs)

def put(self, request, \*args, \*\*kwargs):

return self.update(request, \*args, \*\*kwargs)

def delete(self, request, \*args, \*\*kwargs):

return self.destroy(request, \*args, \*\*kwargs)

* Why they may not be suitable for every use case??
* Because they are difficult to test, difficult to read and understand, not highly customizable but good for large scale code that has too many repeated non highly customized behavior. It is up to you to decide what is good for you

# Securing the endpoints and Instantiating Related objects manually, Pagination (6)

Update REST\_FRAMEWORK setting in setting.py to look like below

REST\_FRAMEWORK = {

'DEFAULT\_AUTHENTICATION\_CLASSES': (

'rest\_framework.authentication.BasicAuthentication',

'rest\_framework.authentication.SessionAuthentication',

'rest\_framework.authentication.TokenAuthentication',

),

'DEFAULT\_PERMISSION\_CLASSES': [

'rest\_framework.permissions.AllowAny'

],

'TEST\_REQUEST\_DEFAULT\_FORMAT': 'json'

}

I am going to demonstrate the TokenAuthentication system here

In the same file update the INSTALLED\_APPS to look like the one below

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'nonTrivialApp',

'rest\_framework',

'rest\_framework.authtoken',

]

Save this and and run:

>python manage.py makemigrations

>python manage.py migrate

In views.py import

from rest\_framework.decorators import api\_view, permission\_classes

from rest\_framework.permissions import IsAuthenticated

and use the decorator as below to make a view function restricted to Authenticated users only

@api\_view(['GET', 'POST'])

@permission\_classes((IsAuthenticated, ))

def get\_post\_stores(request):

# get all stores in the system

if request.method == 'GET':

stores = Store.objects.all()

serializer = StoreSerializer(stores, many=True)

return Response(serializer.data)

# insert a new record for a store

elif request.method == 'POST':

return Response({})

Now try to visit the endpoint using GET method at <http://127.0.0.1:9999/api/v1/stores/>

You will get a

**HTTP 403 Forbidden**

NB: you can use:

if request.user.is\_authenticated:

inside a function to restrict access

Now let us give a registered user a token on sign up and each time they want to interact with a restricted endpoint, they must send that token in the header of their http request.

Add below code to urls.py of the app

,

url(

r'^api/v1/create-customer/$',

views.create\_customer,

name='create\_customer'

),

url(

r'^api/v1/authenticate-customer/$',

views.authenticate\_customer,

name='authenticate\_user'

)

And add these imports and functions in views.py

from rest\_framework.decorators import api\_view, permission\_classes

from rest\_framework.permissions import IsAuthenticated

from django.utils.html import escape

from django.contrib.auth.models import User

from rest\_framework.authtoken.models import Token

from django.contrib.auth import authenticate

@api\_view(['POST'])

def create\_customer(request):

try:

username = escape(request.POST["username"])

email = escape(request.POST["email"])

password = escape(request.POST["password"])

except:

return Response({

'error': 'missing form data'

})

gender = escape(request.POST.get("gender", "-"))

line\_1 = escape(request.POST.get("line\_1", "-"))

line\_2 = escape(request.POST.get("line\_2", "-"))

city = escape(request.POST.get("city", "-"))

country = escape(request.POST.get("country", "-"))

lat = escape(request.POST.get("lat", '0'))

lon = escape(request.POST.get("lon", '0'))

# check username taken

user = User.objects.filter(username=username)

if len(user) > 0:

return Response({"error": "username taken"})

# check email taken

user = User.objects.filter(email=email)

if len(user) > 0:

return Response({"error": "email taken"})

# create user

user = User.objects.create\_user(

username=username,

password=password,

email=email

)

token = Token.objects.create(user=user)

# create the Address of the customer

customer\_address = Address(

line\_1=line\_1,

line\_2=line\_2,

city=city,

country=country,

lat=lat,

lon=lon

)

customer\_address.save()

# adding the the user object and the address object to build customer object

customer = Customer(

user=user,

gender=gender,

address=customer\_address

)

customer.save()

return Response({

'token': str(token)

})

@api\_view(['POST'])

def authenticate\_customer(request):

username = request.POST.get('username', False)

password = request.POST.get('password', False)

print(username)

user = User.objects.filter(username=username)

if len(user) > 0: # username correct

user = authenticate(username=username, password=password)

if user is not None:

token, created = Token.objects.get\_or\_create(user=user)

return Response({

'token': str(token)

})

else:

return Response({

'error': 'wrong password'

})

else:

return Response({

'error': 'wrong username'

})

Posting data using json in request body (application/json)

@api\_view(['GET', 'POST'])

def get\_post\_addresses(request):

# get all addresses in the system

if request.method == 'GET':

# paging code

page\_number = request.GET.get("page\_number", 1)

items\_per\_page = request.GET.get("items\_per\_page", 5)

paginator = Paginator(address\_list, items\_per\_page)

try:

addresses = paginator.page(

int(page\_number))

except PageNotAnInteger:

addresses = paginator.page(1)

except EmptyPage:

addresses = paginator.page(

paginator.num\_pages)

serializer = AddressSerializer(addresses, many=True)

return Response(serializer.data)

# insert a new record for a store

elif request.method == 'POST':

data = {

'line\_1': request.data.get('line\_1'),

'line\_2': request.data.get('line\_2'),

'city': request.data.get('city'),

'country': request.data.get('country'),

'lat': request.data.get('lon'),

'lon': request.data.get('lat')

}

serializer = AddressSerializer(data=data)

if serializer.is\_valid():

serializer.save()

return Response(serializer.data, status=status.HTTP\_201\_CREATED)

return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

Use post man to test and post the json string in post and see the results using get