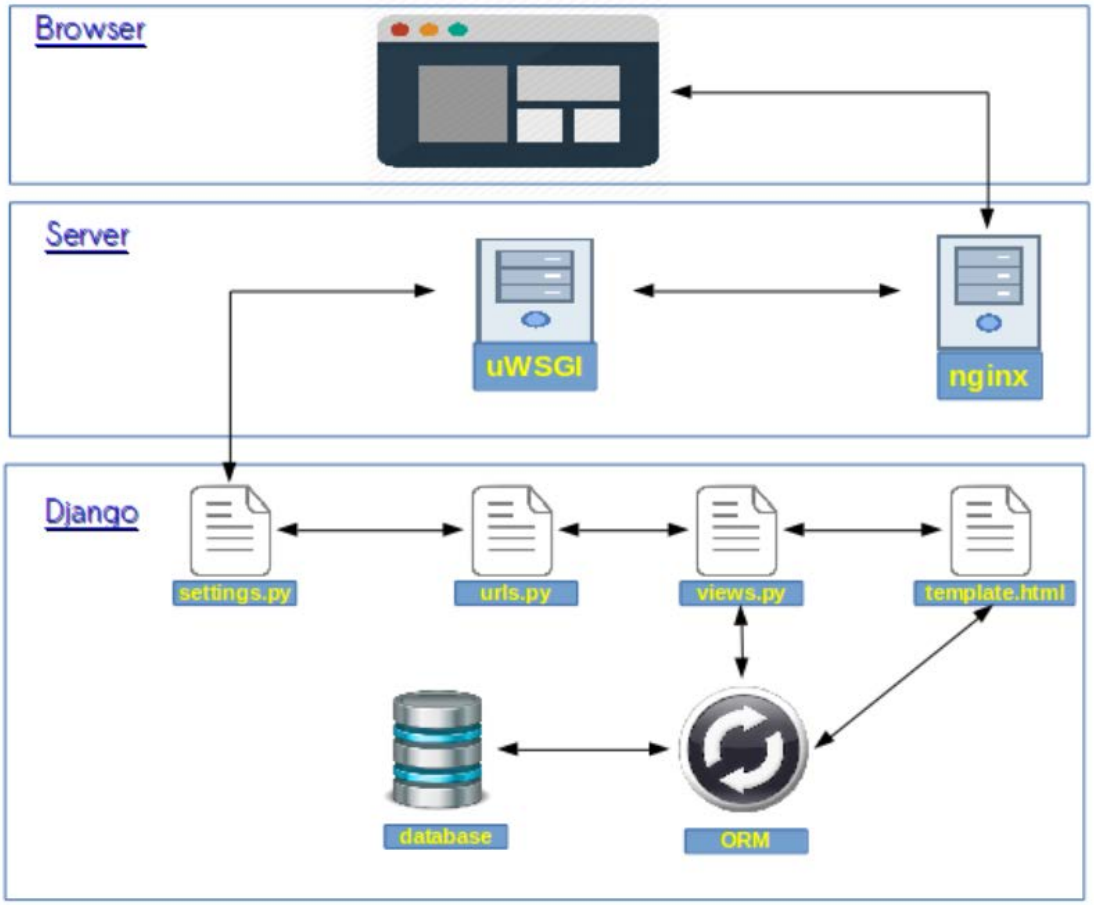
Intro

Developed by**:** Django Software Foundation
Stable release**:** 2.1.7 (11 February 2019; 29 days ago)
Preview release**:** 2.1 beta 1 (18 June 2018; 8 months ago)
License**:** 3-clause BSD https://opensource.org/licenses/BSD-3-Clause
Size**:** 7.6 MB
Initial release date**:** 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.

Popular sites build using Django

1) Instagram
2) Bitbucket
3) Pintrest

The general overview of how Django works:



Django has a simple Request/Response pipeline. When a request comes in Django applies the list of middlewares to the request and applies them again to the response generated by the views.

1. The actual steps.

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 [1]

     Run python get-pip.py. [2] This will install or upgrade pip. Additionally, it will install setuptools and 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
```

2) Create a directory where the project code is going to reside and cd there
   - projectFolder

3) Use the installed Django to create a **project**

```
(myEnv)$ django-admin startproject nonTrivialProject
```

4) Cd to `nonTrivialProject` directory and add an Application in there using the following commands:

```
(myEnv)$ cd nonTrivialProject
(myEnv)$ python manage.py startapp nonTrivialApp
```

5) Now we need to configure our Django project to make use of the REST Framework.

   First add the **nonTrivialApp**, the **rest_framework** and the **rest_framework.authtoken** to the INSTALLED_APPS section in the **nonTrivialApp/settings.py** file of our project.

Snippet 1

```
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',
]
```

6) Next define a global settings for the Res Framework in a single dictionary, again in the setting.py file

Snippet 2

```
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'
}
```

This allows unrestricted access to the API and sets the default format to json for all requests

7) Paste the following code under the line "STATIC_URL = '/static/' in the settings.py to configure where files will be stored and served from in your project

Snippet 3

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')

MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

8) If you want to connect your Django app to an external database server you can do the setup as below:
9) Once you have a working Postgres server on your system, open the Postgres interactive shell and create the database:

Snippet 4

```
$ psql

# CREATE DATABASE flower_store;
CREATE DATABASE
# \q
```

Install psycopg2 so that we can communicate with the Postgres server via Python:

- (myEnv)$ pip install psycopg2

Update the database configuration in the **settings.py** file,adding the appropriate username and password as shown below

Snippet 5

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'flower_store',
        'USER': 'theUsernameHere',
        'PASSWORD': '<theBatabasePasswordHere>',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

10) Now we can define a single model, add it to the admin dashboard and apply migrations

Snippet 6

```
from django.db import models

from django.contrib.auth.models import User


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",)
```

11) Add the model to the admin dashboard from **nonTrivialApp/admin.py** of the app folder.

Snippet 7

```
from django.contrib import admin
from nonTrivialApp.models import Store  # Import your model here.
# Register your models here.
admin.site.register(Store) # This make your model accessible from admin portal
```

12) Create a Serializer to serialize the models' data so that it can be sent in the http responses.
    i) Create a file named "**serializers.py**" in the **nonTrivialApp** folder.
    ii) Add the code below to your **serializers.py** file.

Snippet 8

```
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',
        )
```

iii) Run the following commands

$ python manage.py makemigrations

$ python manage.py migrate

$ python manage.py createsuperuser

The above commands make the framework create the tables corresponding to our models in the connected database. The last command creates admin user who can access the admin portal at **/admin** with the entered credentials.

13) 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

14) We can now launch the app by typing the command:
- $ python manage.py runserver 9999
- Visit the app at http://localhost:9999
- Visit the admin portal at http://localhost:9999/admin

2. Setup the Routing to point to the view functions.

1) Create **urls.py** in the **nonTrivialApp** folder
2) Define some of the routes of the REST API endpoints by pasting the code below into the new file.

Snippet 9

```
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<id>[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)
```

3) Define the view functions in the **views.py** file in the **nonTrivialApp** folder

Snippet 10

```
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({})
```

4) Add the following 2 lines to **urls.py** file in the **nonTrivialProject** folder.

Snippet 11

```
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')),
]
```

5) Now let's run the server again and visit the urls of our REST API
   - http://127.0.0.1:9999/api/v1/stores/1
   - http://127.0.0.1:9999/api/v1/stores/

     You can explore and interact with the newly created REST API with the browser.
     The Django REST framework has a nie GUI for this.
     But who tests there APIs 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.
6) You can use Postman or curl to interact with your API.
   - Open Postman and test :
   - http://127.0.0.1:9999/api/v1/stores/1
   - http://127.0.0.1:9999/api/v1/stores/

3. Before we get too carried away developing the app,TESTING TESTING and TESTING

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

If you want to follow the TDD approach, the necessary tools are available 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 **nonTrivialApp** folder
2) Add a file named **test_views.py** and delete **tests.py** in **nonTrivialApp** folder and also add **__int__.py** inside the **tests** folder
   Syntax and file structure sample

Snippet 12

```
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_store'))
        # 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)
```

3) Run $ **python manage.py test**
4) The test **will fail** because the view function we are testing is returning an empty dictionary as its response. We wanted it to fail! This is the first step of our TDD cycle. Now we have to further develop the view function to pass this test. After it passes then rinse and repeat.
5) You can also see the test coverage of your project using the coverage package (https://docs.djangoproject.com/en/2.1/topics/testing/advanced/#integration-with-coverage-py) we installed during the setup by running the following commands:

```
coverage run --source='.' manage.py test
coverage report
```

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

4. Developing the view functions and more about the ORM mapper.

1) Edit the **views.py** to look like the one below. Notice the new imports and the code added to the **get_post_store()** function.

Snippet 13

```
from django.shortcuts import render

from rest_framework.response import Response
from rest_framework.decorators import api_view
from nonTrivialApp.models import Store
from nonTrivialApp.serializers import StoreSerializer
# Create your views here.


@api_view(['GET', 'DELETE', 'PUT'])
# dont forget the "id" param that comes from the path.
def get_delete_update_store(request, id):
    return Response({})


@api_view(['GET', 'POST'])
def get_post_store(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({})
```

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 wanted to bring the data of the user embedded in this response????
4) Easy peasy, just add a **UserSerializer** as a property of the **StoreSerializer** class. (Uncomment that attribute coz I had already put it there to save time ☺)

Snippet 14

```
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.
- Let's run the tests again and see them pass because the view function is now returning all the Stores the GET method.
  python manage.py test

We can also write the tests for the other API endpoints

- Let's test the endpoint for getting a specific store
- Add the class below to the **test_views.py** file in the **tests** folder

Snippet 15

```
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 edit the view function **get_delete_update_store()** for it to pass the test.

Snippet 16

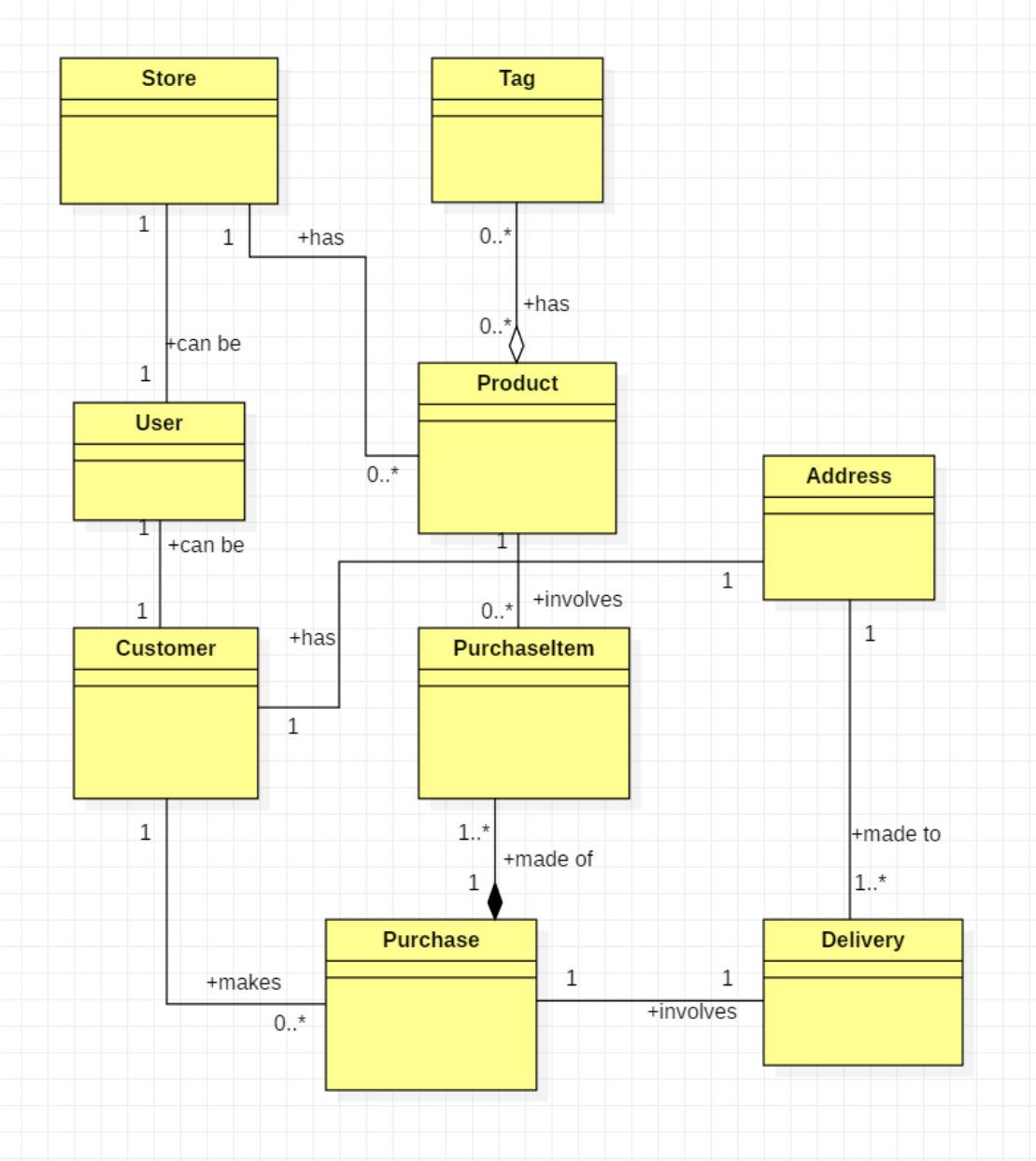
```
from django.shortcuts import render

from rest_framework.response import Response
from rest_framework.decorators import api_view
from nonTrivialApp.models import Store
from nonTrivialApp.serializers import StoreSerializer
from rest_framework import status
# Create your views here.


@api_view(['GET', 'DELETE', 'PUT'])
# dont forget the "id" param that comes from the path.
def get_delete_update_store(request, id):
    # 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)
    elif request.method == 'DELETE':
        return Response({})
    elif request.method == 'PUT':
        return Response({})


@api_view(['GET', 'POST'])
def get_post_store(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({})
```

Run the tests to see that the code passes the tests

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

coverage report

5. Expanding the Models and Serializers.



Snippet 17

```
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.mode
ls.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 these Models so that they can appear in the admin portal. Add the code into admin.py

Snippet 18

```
from django.contrib import admin
# Import your models here.
from nonTrivialApp.models import Store, Address, Customer, Delivery,
Product, Purchase, PurchaseItem, Tag
# Register your models here.
admin.site.register(Store)  # This make your model accessible from
admin portal
admin.site.register(Address)
admin.site.register(Customer)
admin.site.register(Delivery)
admin.site.register(Product)
admin.site.register(Purchase)
admin.site.register(PurchaseItem)
admin.site.register(Tag)
```

Commit these changes to the Database bu running the commands:

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 the new models

Snippet 19

```
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',
        )
```

6. Dealing with Auto Creation of Database entries (Using Generic Class Based Views)

You can use Generic Class Based views instead of the function based views we have used up to now. Using GCBVs will reduce the number of line of code you have to write upfront.

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

- Add 2 urls dealing with Tags to the **urls.py** in **nonTrivialApp** folder by making the **urlpatterns** list look like the one below.

Snippet 20

```
urlpatterns = [

    url(
        r'^api/v1/stores/(?P<id>[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'
    ),
    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

Snippet 21

```
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

Snippet 22

```
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)
```

- Let's test the end points using Postman with a POST request with a body containing {"name":"Lilly" } and Content-Type application/json in the header
- Let's test the end points using Postman with a PUT request with a body containing {"name":"Tulip" } and Content-Type application/json in the header and pk =1 in the url (http://localhost:9999/api/v1/tags/1/)
- Revisit the get request to see the updated record in the list
- Let's test the end points using Postman with a DELETE request @ http://localhost:9999/api/v1/tags/1/
- Revisit the get request to see the deleted record missing from the list.

NB: if you don't like typing the browsable api has forms to interact with your api made for you.

- 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

7. Securing the endpoints, Instantiating Related objects

In our settings.py file we have the following configuration for the Django REST Framework. The 3 classes in the value of **'DEFAULT_AUTHENTICATION_CLASSES'** give our API the ability to handle different types of Authentication mechanisms. Choose what is best for your use case because there are many options.

Snippet 23

```
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 in this project.

TokenAuthentication is dependent on us declaring **'rest_framework.authtoken'** the INSTALLED_APPS list like we have already done before.

Snippet 24

```
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',
]
```

If you add this value to the list you have to save and enter the commands to make migrations to your DB. This creates a table that willkeep track of the tokens issued to our users.

```
python manage.py makemigrations
python manage.py migrate
```

Securing your endpoints involves:

1) Adding the imports below to your views.py file

Snippet 25

```
from rest_framework.decorators import permission_classes
from rest_framework.permissions import IsAuthenticated
```

2) Add the decorator **@permission_classes((IsAuthenticated,))** to a view function to make it restricted to Authenticated users only like below.

Snippet 26

```
@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 401 Unauthorized
```

NB: you can use: **if request.user.is_authenticated:** inside a function to restrict parts of the function body.

For requests to gain access to a restricted endpoint they must provide a Token in the header section. A token is generated when a user is created. The user sending a request has to identify themselves using their corresponding token.

Add below code to **urls.py** of the **nonTrivialApp**

Snippet 27

```
    ,
    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 the following imports to **views.py.**

Snippet 28

```
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
from nonTrivialApp.models import Customer, Address
```

Add the following functions to **views.py**

Snippet 29

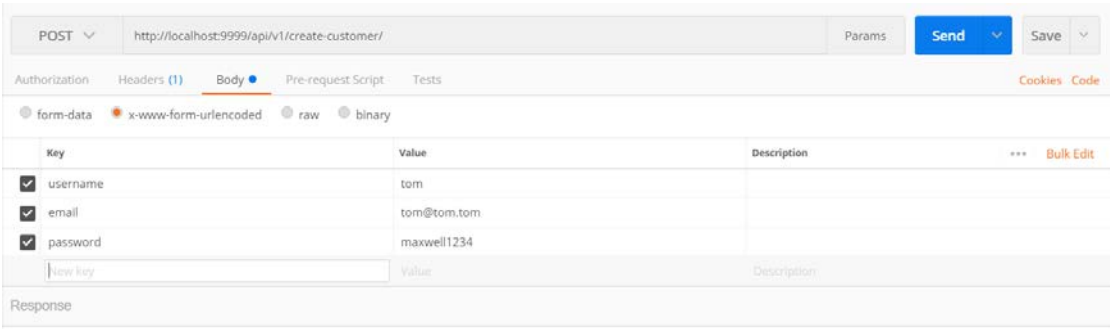
```
@api_view(['POST'])
def create_customer(request):
    try:
        username = escape(request.POST["username"])
        email = escape(request.POST["email"])
        password = escape(request.POST["password"])
    except:
        return Response({
            'error': 'missing form data'
        })

    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 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)
    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({
```

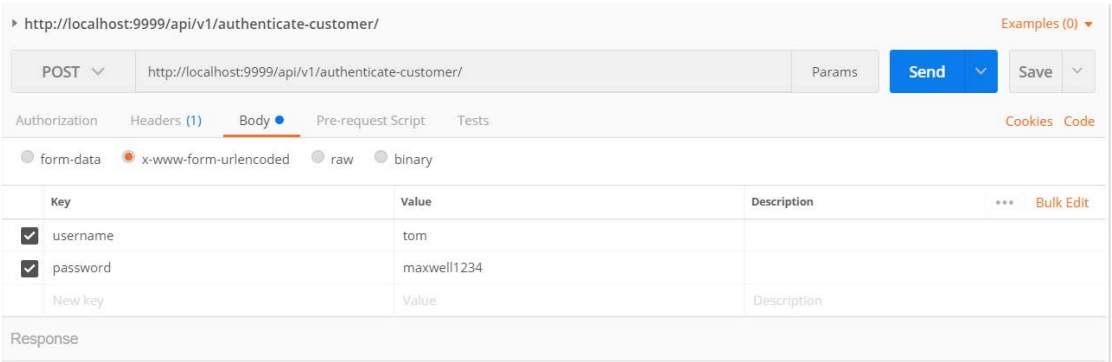
```
                'token': str(token)
            })
        else:
            return Response({
                'error': 'wrong password'
            })
    else:
        return Response({
            'error': 'wrong username'
        })
```

In Postman let's visit http://localhost:9999/api/v1/stores/ using the GET request. It will be restricted because we do not have an access token.

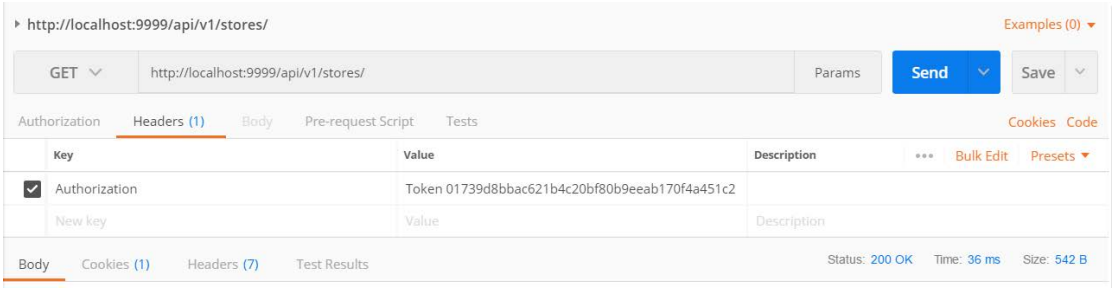
- Use Postman to POST a username,email,password to http://localhost:9999/api/v1/create-customer/ with the x-www-form-urlencoded option selected.



- Use Postman to authenticate the customer we just created by using a POST request with **username** and **password** as request data to http://localhost:9999/api/v1/authenticate-customer/ .



- You will get the token in the response and we are going to use that token to access the restricted endpoint at http://localhost:9999/api/v1/stores/ using Postman.



NB: if you don't send the token you won't get in.

8. Pagination.

Posting data using json in request body (application/json) and Paginating the results. First at the following imports to the views.py

Snippet 30

```
from nonTrivialApp.serializers import AddressSerializer
from django.core.paginator import Paginator, EmptyPage,
PageNotAnInteger
```

After that add the following view function in the views.py

Snippet 31

```
@api_view(['GET', 'POST'])
def get_post_addresses(request):
    # get all addresses in the system
    if request.method == 'GET':
        address_list = Address.objects.all()

        # 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)
```

Add a route for the point to this new view function by adding the following code to urls.py in nonTrivialApp folder.

Snippet 32

```
    ,
    url(
        r'^api/v1/addresses/$',
        views.get_post_addresses,
        name='get_post_addresses'
    )
```

Run the server: python manage.py runserver 9999

Use Postman to test the **creation** (POST) of objects from json and the **pagination** (GET) of the created objects. http://localhost:9999/api/v1/addresses/?page_number=1

9. Querying the objects from the Database.

Django has a comprehensive query system

We need to import the following into **views.py**

Snippet 33

```
from django.db.models import F
from django.db.models import Q
from nonTrivialApp.models import Product, Tag
from nonTrivialApp.serializers import ProductSerializer
```

We then create a view function that will create dummy products and we use the get method to demonstrate parts of the query system of Django. Add the code below to **views.py**

Snippet 34

```python
@api_view(['GET', 'POST'])
def get_post_products(request):

    if request.method == 'POST':
        # create products
        product1 = Product(
            name="Product1",
            description="Product 1 description",
            price=100,
        )
        product1.save()

        product2 = Product(
            name="Product2",
            description="Product 2 description",
            price=100,
        )
        product2.save()

        # create tags
        tag1 = Tag(name='Tag 1')
        tag1.save()
        tag2 = Tag(name='Tag 2')
        tag2.save()
        tag3 = Tag(name='Tag 3')
        tag3.save()
        tag4 = Tag(name='Tag 4')
        tag4.save()

        product1.tags.add(tag1)
        product1.tags.add(tag2)

        product2.tags.add(tag2)
        product2.tags.add(tag3)
        product2.tags.add(tag4)

        # create tags
        return Response({'message': 'Data created successfully'})
    elif request.method == 'GET':
        products = Product.objects.all()
        # products = Product.objects.filter(tags__name="Tag 1")
        # products = Product.objects.filter(tags__name__in=["Tag 1"])
        # products = Product.objects.all().order_by("name")
        # products = Product.objects.filter(price__gt=500)
        # products = Product.objects.filter(name__icontains="2")
        product_serializer = ProductSerializer(products, many=True)
```

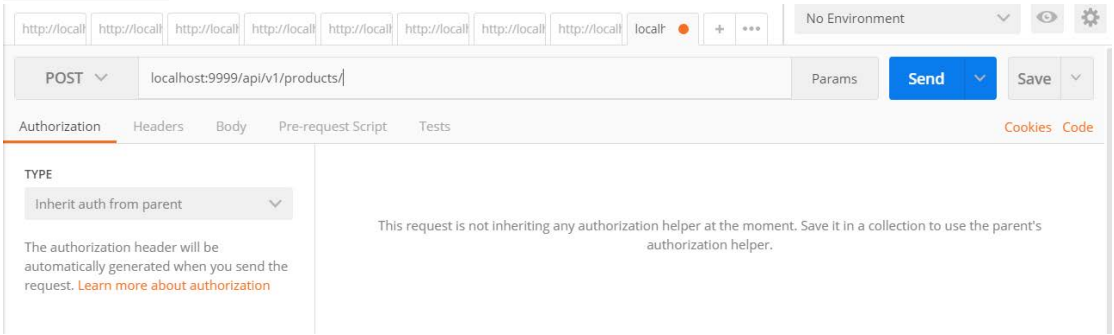
```
        tags = Tag.objects.all()
        #tags = Tag.objects.filter(product__name="Product2")
        #tags = Tag.objects.filter(product__name="Product1")
        tag_serializer = TagSerializer(tags, many=True)
        return Response({
            'products': product_serializer.data,
            'tags': tag_serializer.data
        })
```

And add the code below to **urls.py**

Snippet 35

```
    ,
    url(
        r'^api/v1/products/$',
        views.get_post_products,
        name='get_post_products'
    ),
```

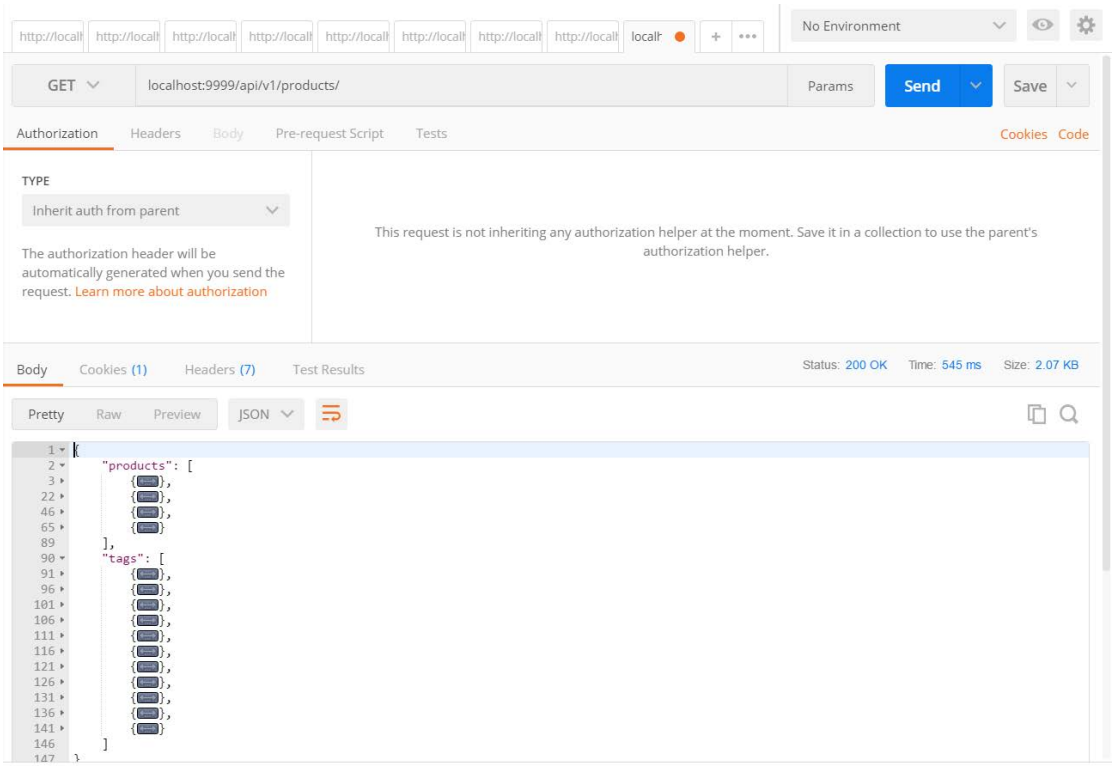
Make a single POST request to http://localhost:9999/api/v1/products/ to create the dummy data using Postman.



We will uncomment some parts of the code in the view function that is handling this request and inspect the response data to see how the queries can be filtered using some of the keywords below.

- filter(), __gt, __gte, __lt, __lte, __count, __in,__startswith, __icontains
- exclude()
- order_by()
- or using Q()
- using F()
- forward join
- reverse join

Send GET requests after uncommenting the view code to see the results using Postman.



10. Securing the Statics.

You can secure your media or static files using the code below in your **urls.py** in **nonTrivialApp** folder.

The code below already exists in the urls.py and we do not have to change it.

```
from django.conf.urls import patterns, include, url
from django.contrib.auth.decorators import login_required
from django.views.static import serve
from django.conf import settings


@login_required
def protected_serve(request, path, document_root=None, show_indexes=False):
    return serve(request, path, document_root, show_indexes)
```

We only need to add the snippet below to the urlspatterns list in **urls.py** of the **nonTrivialApp** folder.

Snippet 36

```
    ,
    url(
        r'^%s(?P<path>.*)$' % settings.MEDIA_URL[1:],
        protected_serve,
        {'document_root': settings.MEDIA_ROOT}
    ),
```