

<https://www.udemy.com/django-python/>

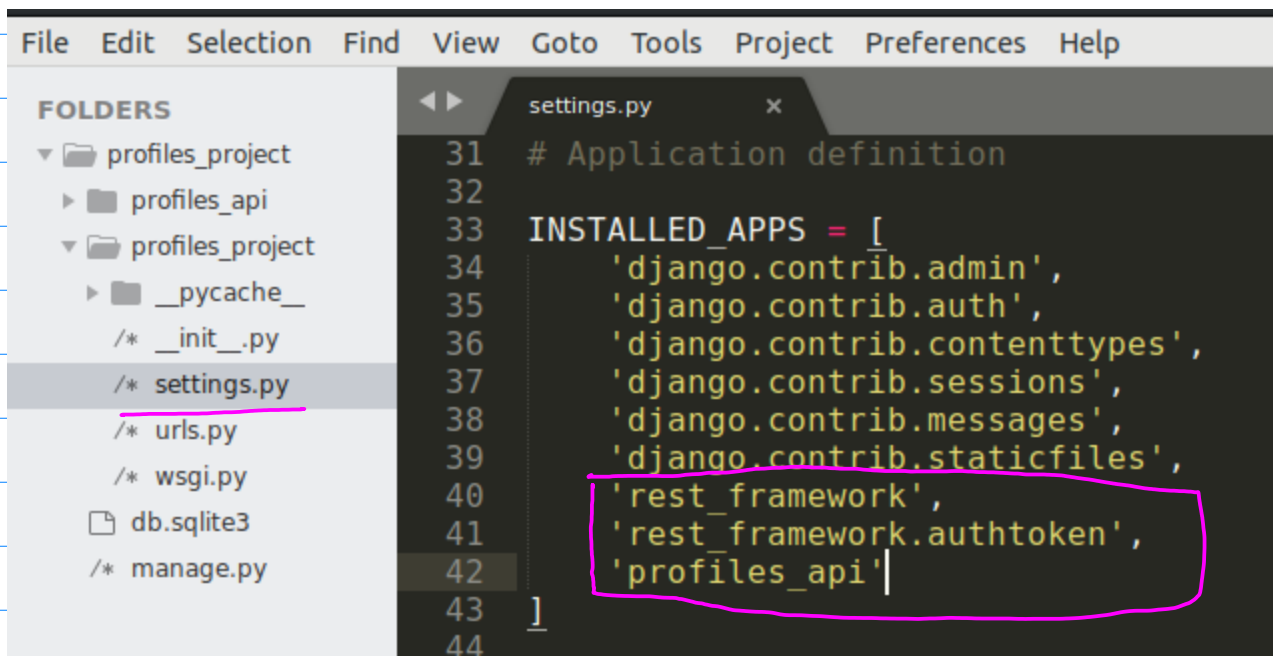
sudo apt-get install python3-venv

```
amiya@amiya:~/.../rest-api$ python3 -m venv apienv
amiya@amiya:~/.../rest-api$ source apienv/bin/activate
(apienv) amiya@amiya:~/.../rest-api$
```

pip3 install django

pip3 install djangorestframework

```
(apienv) amiya@amiya:~/.../rest-api$ mkdir src
(apienv) amiya@amiya:~/.../rest-api$ cd src
(apienv) amiya@amiya:~/.../src$ django-admin.py startproject profiles_project
(apienv) amiya@amiya:~/.../src$
(apienv) amiya@amiya:~/.../src$
(apienv) amiya@amiya:~/.../profiles_project$ python3 manage.py startapp profiles_api
(apienv) amiya@amiya:~/.../profiles_project$
(apienv) amiya@amiya:~/.../profiles_project$
(apienv) amiya@amiya:~/.../profiles_project$ ls
manage.py  profiles_api  profiles_project
```



The screenshot shows an IDE window with a menu bar (File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, Help) and a sidebar showing a folder tree. The main editor displays the contents of settings.py, with the following code highlighted by a pink box:

```
31 # Application definition
32
33 INSTALLED_APPS = [
34     'django.contrib.admin',
35     'django.contrib.auth',
36     'django.contrib.contenttypes',
37     'django.contrib.sessions',
38     'django.contrib.messages',
39     'django.contrib.staticfiles',
40     'rest_framework',
41     'rest_framework.authtoken',
42     'profiles_api'
43 ]
44
```

```
(apienv) amiya@amiya:~/.../rest-api$ pip3 freeze > requirements.txt
```

What are APIViews?

Uses standar HTTP Methods for functions

GET, POST, PUT, PATCH, DELETE

Gives you the most control over the logic:

Perfect for implementing complex logic

Calling other APIs

When to use APIViews?

Some examples of when to use an APIView:

- You need the full control over the logic.
- Processing files and rendering a synchronous response.
- You are calling other APIs/Services.
- Accessing local files or data.

models.py

```
class UserProfileManager(BaseUserManager):
    """Helps Django work with our custom user model."""
    def create_user(self, email, name, password=None):
        """Creates a new user profile object."""
        if not email:
            raise ValueError("Users must have an email address.")
        email = self.normalize_email(email)
        user = self.model(email=email, name=name)
        user.set_password(password)
        user.save(using=self._db)
        return user
    def create_superuser(self, email, name, password):
        """Creates and saves a new superuser with given details."""
        user = self.create_user(email, name, password)
        user.is_superuser = True
        user.is_staff = True
        user.save(using=self._db)
        return user
```

models.py

```
class UserProfile(AbstractBaseUser, PermissionsMixin):
    """Represents a user profile inside our system"""
    email = models.EmailField(max_length=255, unique=True)
    name = models.CharField(max_length=255)
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)

    # Object manager is a class to manage the userprofile, giving it extra functionality
    objects = UserProfileManager()

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['name']

    def get_full_name(self):
        """Used to get a users full name."""
        return self.name

    def get_short_name(self):
        """Used to get a users short name."""
        return self.name
```

```
/* views.py
▼ profiles_project
  ► __pycache__
  /* __init__.py
  /* settings.py
  /* urls.py
  /* wsgi.py
  db.sqlite3
+124
+125 AUTH_USER_MODEL = 'profiles_api.UserProfile'
126
```

```
/* views.py
▼ profiles_project
  ► __pycache__
  /* __init__.py
  /* settings.py
  /* urls.py
  /* wsgi.py
+16
17 from django.contrib import admin
+18 from django.urls import re_path, path, include
19
+20 urlpatterns = [
21     path('admin/', admin.site.urls),
+22     re_path(r'^api-auth/', include('rest_framework.urls')),
+23     path('api/', include('profiles_api.urls'))
24 ]
```

The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'rest-api', 'apienv', 'profiles_project', and 'profiles_api'. The code editor shows the following Python code in 'urls.py':

```
1 from django.urls import re_path, path, include
2 from . import views
3
4 urlpatterns = [
5     path('hello-view/', views.HelloAPIView.as_view()),
6 ]
```

profiles_api/serializers.py

The screenshot shows a code editor with the following Python code in 'serializers.py':

```
from rest_framework import serializers

class HelloSerializer(serializers.Serializer):
    """Serializes a name field for testing our APIView."""

    name = serializers.CharField(max_length=10)
```

What are APIViews?

Uses standard HTTP Methods for functions

GET, POST, PUT, PATCH, DELETE

profiles_api/views.py

GET

```
from . import serializers
from rest_framework import status

# Create your views here.

class HelloAPIView(APIView):
    """Test API View."""
    serializer_class = serializers.HelloSerializer

    def get(self, request, format=None):
        """Returns a list of APIView features."""
        an_apiview = [
            'User HTTP methods as function (get, post, patch, put delete)',
            'It is similar to a traditional Django view',
            'Gives you the most control over your logic',
            'Is mapped manually to URLs',
        ]

        return Response({'message': 'Hello!', 'an_apiview': an_apiview})
```

POST

```
    def post(self, request):
        """Create a Hello Message with our name."""
        serializer = serializers.HelloSerializer(data=request.data)

        if serializer.is_valid():
            name = serializer.data.get('name')
            message = 'Hello {}'.format(name)
            return Response({'message': message})
        else:
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

PUT

```
    def put(self, request, pk=None):
        """Handles updating an object."""
        return Response({'method': 'put'})
```

PATCH

```
    def patch(self, request, pk=None):
        """Patch request, only updates fields provided in the request."""

        return Response({'method': 'patch'})
```

DELETE

```
    def delete(self, request, pk=None):
        """Deletes an object."""

        return Response({'method': 'delete'})
```

Hello Api - Django | x
localhost:8000/api/hello-view/

Django REST framework

Hello Api

Test API View.

DELETE OPTIONS GET

GET /api/hello-view/

HTTP 200 OK
Allow: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{  
  "message": "Hello!",  
  "url": "/api/hello-view/",  
  "methods": "Use HTTP methods as functions (get, post, patch, put, delete)".  
  "is_similar_to_a_traditional_django_view": "It is similar to a traditional Django view".  
  "gives_you_the_most_control_over_your_login": "Gives you the most control over your login".  
  "is_mapped_manually_to_DSL": "Is mapped manually to DSL".  
}
```

Media type: application/json

Content:

POST

Media type: application/json

Content:

PUT PATCH

delete ✓
get
post
put → *patch*

POST

Media type: application/json

Content: {"name": "Amiya"}

POST

POST Results

Hello Api

Test API View.

```
POST /api/hello-view/
```

```
HTTP 200 OK
```

```
Allow: GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS
```

```
Content-Type: application/json
```

```
Vary: Accept
```

```
{  
  "message": "Hello Amiya"  
}
```

Viewsets

Examples of when you might use a Viewset:

- You need a simple CRUD interface to your database.
- You want a quick and simple API.
- You need little to no customization on the logic.
- You are working with standard data structures.

Uses model operations for functions:

- List, Create, Retrieve, Update, Partial Update, Destroy

Takes care of lot of typical logic for you:

- Perfect for standard database operations
- Fastest way to make a database interface

Examples of when you might use a ViewSet:

- You need a simple CRUD interface to your database.
- You want a quick and simple API.
- You need little to no customization on the logic.
- You are working with standard data structures.

Views:

```
from django.shortcuts import render
from rest_framework.views import APIView
from rest_framework import viewsets
from rest_framework.response import Response

from . import serializers
from rest_framework import status
```

```
class HelloViewSet(viewsets.ViewSet):
    """Test API ViewSet."""

    def list(self, request):
        """Return a hello message."""

        a_viewset = [
            'Uses actions (list, create, retrieve, update, partial_update)',
            'Automatically maps to URLs using Routers',
            'Provides more functionality with less code.'
        ]

        return Response({'message': 'Hello!', 'a_viewset': a_viewset})
```

urls.py

```
1 from django.urls import re_path, path, include
2 from rest_framework.routers import DefaultRouter
3 from . import views
4
5 router = DefaultRouter()
6 router.register('hello-viewset', views.HelloViewSet, base_name="hello-viewset")
7
8 urlpatterns = [
9     path('hello-view/', views.HelloAPIView.as_view()),
10    path('', include(router.urls)),
11]
```


http://localhost:8000/api/

Api Root

Api Root

OPTIONS

GET ▾

The default basic root view for DefaultRouter

GET /api/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "hello-viewset": "http://localhost:8000/api/hello-viewset/"
}
```

http://localhost:8000/api/hello-viewset/

Api Root / Hello List

Hello List

OPTIONS

GET ▾

Test API ViewSet.

GET /api/hello-viewset/

HTTP 200 OK

Allow: GET, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
  "message": "Hello!",
  "a_viewset": [
    "Uses actions (list, create, retrieve, update, partial_update)",
    "Automatically maps to URLs using Routers",
    "Provides more functionality with less code."
  ]
}
```

```
def create(self, request):
    """Create a new hello message."""
    serializer = serializers.HelloSerializer(data=request.data)

    if serializer.is_valid():
        name = serializer.data.get('name')
        message = 'Hello {}'.format(name)
        return Response({'message': message})
    else:
        return Response(
            serializer.errors, status=status.HTTP_400_BAD_REQUEST)

def retrieve(self, request, pk=None):
    """Handles getting any object by its ID."""
    return Response({'http_method': 'GET'})

def update(self, request, pk=None):
    """Handles updating an object."""

    return Response({'http_method': 'PUT'})

def partial_update(self, request, pk=None):
    """Handles updating part of an object. """
    return Response({'http_method': 'PATCH'})

def destroy(self, request, pk=None):
    """Handles removing an object."""
    return Response({'http_method': "DELETE"})
```

http://127.0.0.1:8000/api/hello-viewset/1/

Api Root / Hello List / Hello Instance

Hello Instance

Test API ViewSet.

DELETE /api/hello-viewset/1/

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "http_method": "DELETE"
}
```

Raw data HTML form

Media type: application/json

Content:

PUT PATCH

Plan our profiles-ap

Basic Requirements

- Create new profile
 - Validate profile data
- List existing profiles
 - Search for profiles
- View specific profile
- Update my profile of logged in user
 - Update name/email address
 - Change password
- Delet profile

URLs for our API:

- /api/profile/ -- list all profiles
 - GET (list profiles)
 - POST (create profile)

- /api/profile/<profile_id>/ - manage specific profile
- GET (view specific profile)
- PUT/ PATCH (update profile)
- DELETE (remove profile)

```

from rest_framework import serializers
from . import models

class HelloSerializer(serializers.Serializer):
    """Serializes a name field for testing our APIView."""

    name = serializers.CharField(max_length=10)

class UserProfileSerializer(serializers.ModelSerializer):
    """ A serializer for our user profiles objects. """
    class Meta:
        model = models.UserProfile
        fields = ('id', 'email', 'name', 'password')
        extra_kwargs = {'password': {'write_only': True}}

    def create(self, validated_data):
        """Create and return a new user."""
        user = models.UserProfile(
            email = validated_data['email'],
            name = validated_data['name']
        )
        user.set_password(validated_data['password'])
        user.save()

        return user

```

Password:
write_only

What creat func
do?

Userprofile viewsets

```

class UserProfileViewSet(viewsets.ModelViewSet):
    """Handles creating, reading and updating profiles."""
    serializer_class = serializers.UserProfileSerializer
    queryset = models.UserProfile.objects.all()

```

It uses viewsets.ModelViewSet, NOT viewsets.ViewSet as earlier

Register profile viewset with the url router:

```
router = DefaultRouter()
router.register('hello-viewset', views.HelloViewSet, base_name="hello-viewset")
router.register('profile', views.UserProfileViewSet)
urlpatterns = [...]
```

No need of base_name in model viewsets, rest framework can automatically figure out from the model.

http://localhost:8000/api/

Look that we have a new entry called "profile"

Api Root

Api Root

The default basic root view for DefaultRouter

GET /api/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "hello-viewset": "http://localhost:8000/api/hello-viewset/",
  "profile": "http://localhost:8000/api/profile/"
}
```

http://localhost:8000/api/profile/
Profile list view

Api Root / User Profile List

User Profile List

Handles creating, reading and updating profiles.

GET /api/profile/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 1,
    "email": "amiyatulu@gmail.com",
    "name": "Amiya Behera"
  }
]
```

Raw data

HTML form

Email

Name

Post a user profile: Email, Name and Password

Raw data HTML form

Email

Name

Password

POST

POST /api/profile/

HTTP 201 Created
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
{
  "id": 2,
  "email": "test@indiandeveloper.com",
  "name": "Testuser"
}
```

See that it doesn't return password, as its write_only

<http://localhost:8000/api/profile/2/>

Use PATCH to update the profile with id 2

Remove id and email field, because you are not changing it.

Raw data HTML form

Media type:

Content:

```
{
  "name": "Testuser2"
}
```

PUT PATCH

Result:

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "id": 2,
  "email": "test@indiandeveloper.com",
  "name": "Testuser2"
}
```

Raw data HTML form

Media type: application/json

Content:

```
{
  "id": 2,
  "email": "test@indiandeveloper.com",
  "name": "Testuser2"
}
```

PUT PATCH

Permission class

```
from rest_framework import permissions

class UpdateOwnProfile(permissions.BasePermission):
    """Allow users to edit their own profile."""

    def has_object_permission(self, request, view, obj):
        """Check user is trying to edit their own profile."""

        if request.method in permissions.SAFE_METHODS:
            return True

        return obj.id == request.user.id
```

has_object_permission does following things:

the request object includes type of request that is made to the api (e.g. get)
If it comes under SAFE_METHODS, i.e. it is going get request, it will return true
next we gonna check if the user is updating his own profile.

```
from rest_framework.authentication import TokenAuthentication
```

```
class UserProfileViewSet(viewsets.ModelViewSet):  
    """Handles creating, reading and updating profiles."""  
    serializer_class = serializers.UserProfileSerializer  
    queryset = models.UserProfile.objects.all()  
  
    authentication_classes = (TokenAuthentication,) ✓  
    permission_classes = (permissions.UpdateOwnProfile,|) ✓
```

Add search profile feature using filters

```
from rest_framework.authentication import TokenAuthentication  
from rest_framework import filters
```

```
class UserProfileViewSet(viewsets.ModelViewSet):  
    """Handles creating, reading and updating profiles."""  
    serializer_class = serializers.UserProfileSerializer  
    queryset = models.UserProfile.objects.all()  
  
    authentication_classes = (TokenAuthentication,)  
    permission_classes = (permissions.UpdateOwnProfile,)  
    filter_backends = (filters.SearchFilter,) ✓  
    search_fields = ('name', 'email',|) ✓
```

The screenshot shows the Django REST framework API browser interface. At the top, it says "Django REST framework" and "Log in". The main heading is "User Profile List" with the description "Handles creating, reading and updating profiles". Below this, the endpoint is listed as "GET /api/profile/". The response is shown as "HTTP 200 OK" with headers: "Allow: GET, POST, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON array of two user profiles:

```
[  
  {  
    "id": 1,  
    "email": "amiyatulu@gmail.com",  
    "name": "Amiya Behera"  
  },  
  {  
    "id": 2,  
    "email": "test@indiandeveloper.com",  
    "name": "Testuser2"  
  }  
]
```

A "Filters" modal window is open over the page, containing a "Search" input field and a "Q Search" button. The background interface also shows a "Filters" button, "OPTIONS" and "GET" buttons, and a "Log in" link in the top right corner.

