What is a Django Model?

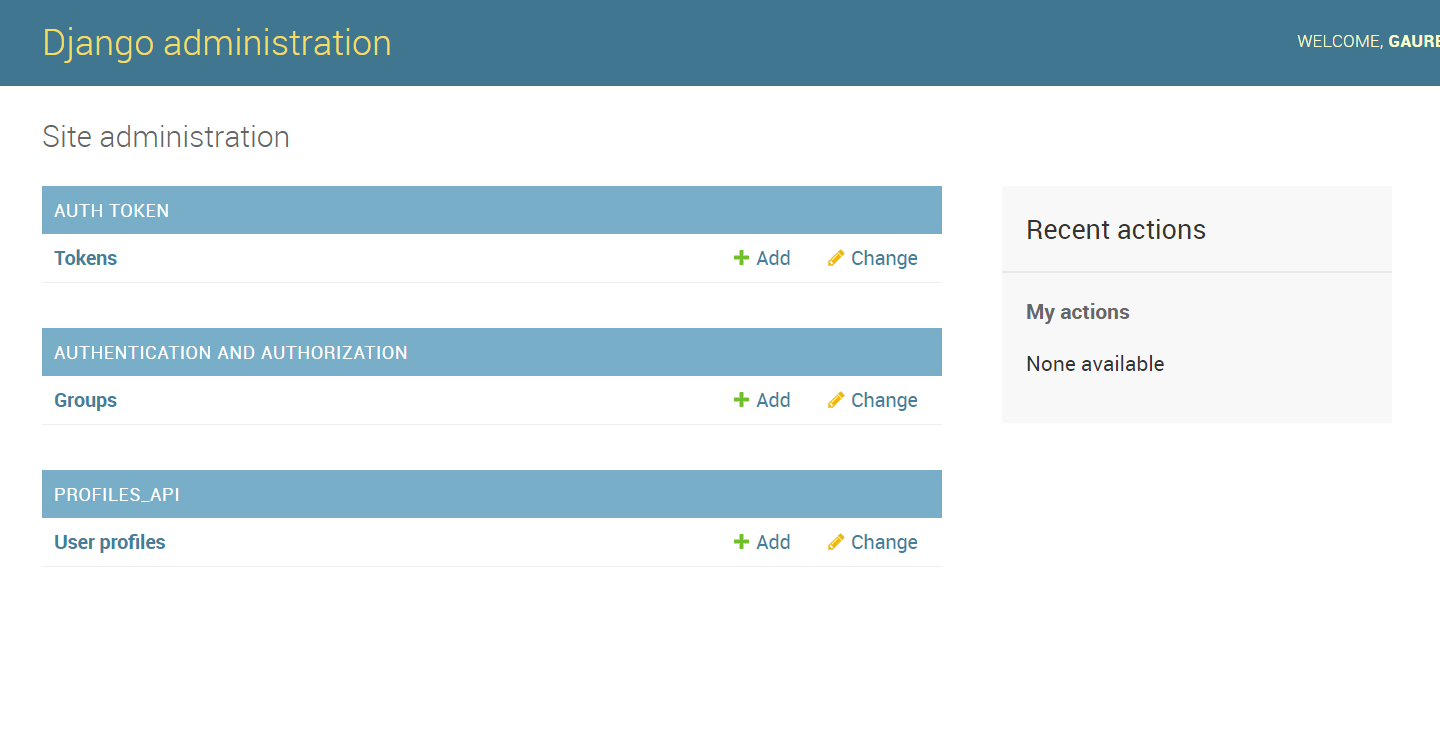
We use models to describe the data we need for our application. Django uses these models to set up a database to store our data effectively

Each model fits in a specific table in the Database.

Django handles the database models for us and we don’t have to write any SQL queries.

What are API views?

How does Django rest work?



Tokens is created by rest framework when we use auth token

Creating User Profile using Django Standard:

from django.contrib.auth.models import AbstractBaseUser #Django default user

from django.contrib.auth.models import PermissionsMixin

from django.contrib.auth.models import BaseUserManager

# Create your models here.

class UserProfileManager(BaseUserManager):

    """ MANAGER FOR USER PROFILE """

    def create\_user(self, email, name, password = None):

        """CREATE A NEW USER PROFILE"""

        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) #encrypts the password

        user.save(using=self.\_db) #standard procedure for saving objects in django

        return user

    def create\_superuser(self, email, name, password):

        """ CREATE AND SAVE NEW SUPERUSER WITH GIVEN DETAILS"""

        user = self.create\_user(email, name, password)

        user.is\_superuser = True #provided by Permissions Mixin

        user.is\_staff = True

        user.save(using=self.\_db)

        return user

AbstractBaseUser, PermissionsMixin, BaseUserManager are standard packages provided by Django to create user models. This is as per Django documentation.

class UserProfile(AbstractBaseUser, PermissionsMixin):

    """DB model for users in the 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)

    objects = UserProfileManager()

    USERNAME\_FIELD = 'email' #replace default user name with email

    REQUIRED\_FIELDS = ['name']

    def get\_full\_name(self):

        """RETRIEVE FULL NAME OF USER"""

        return self.name

    def get\_short\_name(self):

        """RETRIEVE SHORT NAME OF USER"""

        return self.name

    def \_\_str\_\_(self):

        """RETURN STRING REPRESENTATION OF OUR USER MODEL"""

        return self.email #recommended for all django models

**API Views: What is an API view?**

1. Basic type of view we can build to enable our API.
2. Allows us to define functions that match standard HTTP methods
3. HTTP get to get one or more items
4. HTTP post to create an item
5. HTTP put to update an item
6. HTTP patch to partially update an item
7. HTTP delete to delete an item
8. Give us the most control over our application logic
9. Perfect for implementing complex logic
10. Calling other APIs
11. Working with local files

**When to use API views?**

1. Need full control over application logic – running complex algorithm, updating multiple data sources in 1 API call.
2. Processing files and rendering a synchronous response
3. Calling other APIs in same requests
4. Access to local files or data

Creating an API View –

class HelloApiView(APIView):

    """TEST API VIEW"""

    def get(self, request, format=None):

        """RETURNS LIST OF API VIEW FEATURES"""

        #it expects a function for different http request and get request to retreive list of objects or

        # 1 object. Whenever http.get requests made to url assigned to this view it will call get function

        # and execute the logic in the get function

        an\_apiview = [

            'Uses HTTP methods as functions - get, post, patch, delete, put',

            'Is similar to traditonal django view but intended for APIs',

            'gives you the most control over app logic',

            'mapped manually to urls'

        ]

        return Response({'message':'Hello','an\_apiview': an\_apiview}) #List or dictionary to be returned

        #it will convert to json - Whenever the url receives a get request, the response will be given

Serializer:

Feature which allows to convert data inputs to python objects and vice versa

If we want to add post or update(patch) to our api, then we need to create serializers

**What is a ViewSet?**

ViewSets allow to write logic for end points

Maps to common API functions – list, create, update, retrieve, partial update, destroy

Take care of lot of common logic for us.

Perfect to make standard database operations

Fastest way to make a database interface

Used for –

CRUD interface for a DB

Quick and simple API

Very basic custom logic

Working with standard database structure

class HelloViewSet(viewsets.ViewSet):

    """TEST API VIEWSETS"""

    serializer\_class = serializers.HelloSerializer

    def list(self,request):

        """return hello message and it is a get request"""

        a\_viewset = [

            'Uses actions - create, retreive,list, update, partial update, delete',

            'Automatically maps to urls using routers',

            'provides more functionality with less code',

        ]

        return Response({'message':'Hello','a\_viewset':a\_viewset})

    def create(self,request):

        """CREATE A NEW HELLO MESSAGE"""

        serializer = self.serializer\_class(data=request.data)

        if serializer.is\_valid():

            name = serializer.validated\_data.get('name')

            message = f'Hello {name}'

            return Response({'message':message})

        else:

            return Response(

                serializer.errors,

                status=status.HTTP\_400\_BAD\_REQUEST

            )

    def retreive(self,request,pk=None):

        """RETREIVE A PARTICULAR OBJECT"""

        return Response({'http\_method':'GET'})

    def update(self,request,pk=None):

        """UPDATING AN OBJECT"""

        return Response({'http\_method':'PUT'})

    def partial\_update(self,request,pk=None):

        """UPDATING AN OBJECT PARTIALLY"""

        return Response({'http\_method':'PATCH'})

    def destroy(self,request,pk=None):

        """REMOVING AN OBJECT"""

        return Response({'http\_method':'DELETE'})

To access the a specific object in API browser we must provide the key after the url such as home-view/1

Creating UserProfileSerializer

Creating UserProfileViewSet

class UserProfileViewSet(viewsets.ModelViewSet):

    """HANDLE CREATING AND UPDATING PROFILES"""

    serializer\_class = serializers.UserProfileSerializer

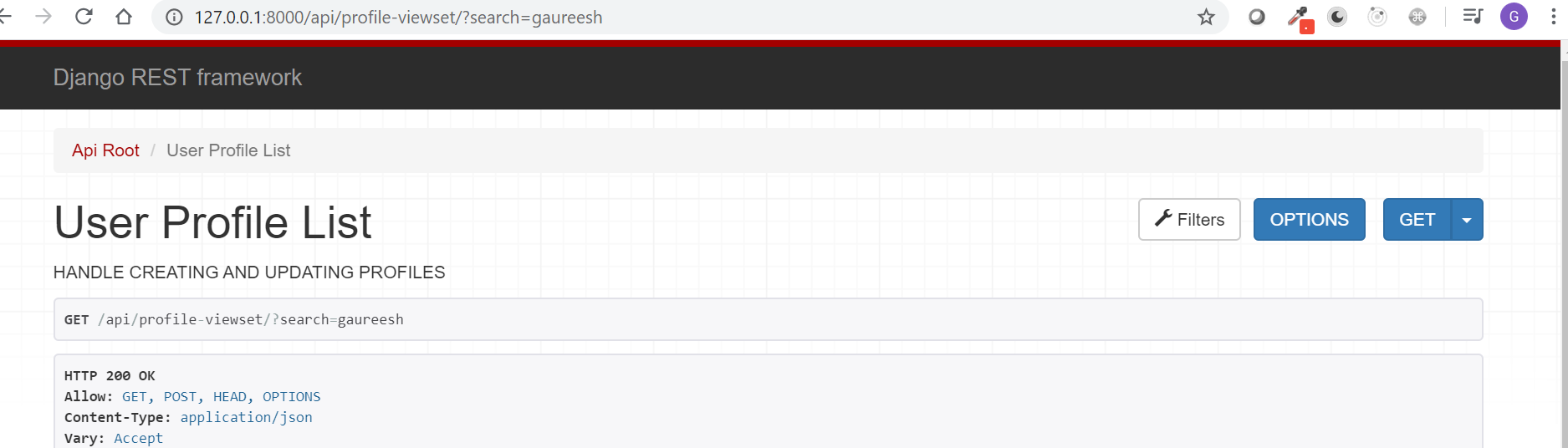
    queryset = models.UserProfile.objects.all() #get all user profile objects in DB

    authentication\_classes = (TokenAuthentication,)#Token authentication

    permission\_classes = (permissions.UpdateOwnProfile,)

    filter\_backends = (filters.SearchFilter,) #Filtering the objects

    search\_fields = ('name','email') #fields for searching



The filter function adds as “search” parameter in the url as a get parameter in the request.

Permission classes are from permissions.py

class UpdateOwnProfile(permissions.BasePermission):

    """ALLOW USERS TO EDIT THEIR OWN PROFILE"""

    def has\_object\_permission(self,request, view, obj):

        """check user trying to edit their own profile and responds a boolean"""

        if request.method in permissions.SAFE\_METHODS:#get is a safe method since it is only reading

            return True

        return obj.id == request.user.id #compare ID of the profile(obj) with user id

Login View:

class UserLoginApiView(ObtainAuthToken):

    """HANDLE CREATING USER AUTHENTICATION TOKEN"""

    renderer\_classes = api\_settings.DEFAULT\_RENDERER\_CLASSES

Imports till now

from django.shortcuts import render

from rest\_framework.views import APIView #import APIView class from rest framework

from rest\_framework.response import Response #used to return response object from the APIView

from rest\_framework import status #HTTP status codes used to return responses from API

from profiles\_api import serializers, models, permissions

from rest\_framework import viewsets

from rest\_framework.authentication import TokenAuthentication #Authenticate user for the API

from rest\_framework import filters

from rest\_framework.authtoken.views import ObtainAuthToken

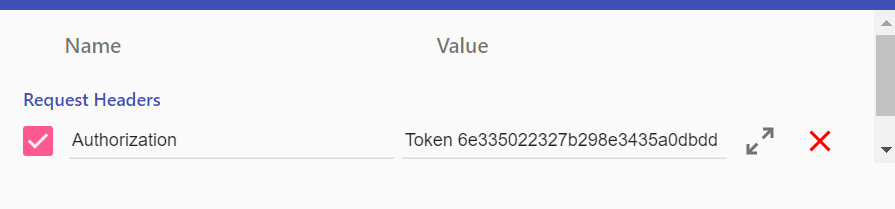
from rest\_framework.settings import api\_settings

# Create your views here.

Here once user logs in user will get a string as an authentication token (API key). This is from the Obtain Auth token class

This key will be used to authenticate the user when he requests.

Enable Modheaders plugin in chrome



In this extension, authorization must be checked and token be entered. This will authenticate the user

That user will be then able to use put or patch requests.

Disable this token when not using Django

In real life, when we use API, token should be passed in any client library – such as Fetch library in JS, requests library in Python or any HTTP client library – those libraries will allow to enter this token to the requests.