

Intro – Few words for the Technologies

This project is based on Udemy - Build a Backend REST API with Python Django – Beginner by [Mark Winterbottom](#). The technologies we're going to use to build our REST API are virtual box, vagrant, Python, Django, the Django rest framework, atom and the MoD had a Chrome extension all of these will work together.

Vagrant is a tool that builds virtual software development environments. It allows us to describe what kind of server we need for our app. We can then save the config as a vagrant file, which allows us to easily reproduce and share the same server with other developers. Our application code and requirements will be installed and running on a virtual server completely isolated from our local machine. This has many benefits such as it makes it easier to share code with others regardless of what operating system we're running our code on. We'll have exactly the same version of all the requirements for our app. We can test our code using exactly the same operating system and requirements that will be used on a real production server. And finally, we can easily create and destroy the server as we need, making it easier to clean up the second set of tools.

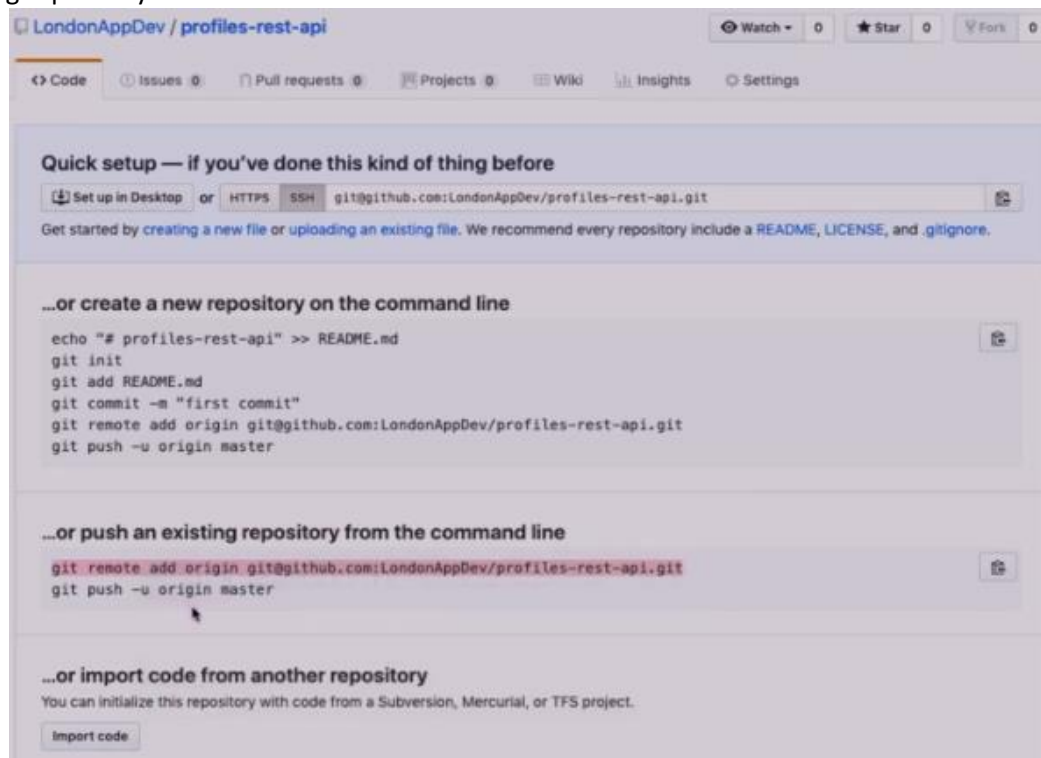
We're going to use Python and a python web framework called Django. On top of Django we're going to use another framework called the Django rest framework. The Django rest framework provides a set of features for making a standard rest api and for building a standard web app.

Also, we'll use the atom editor which is an open source code.

Lastly, we're going to use a Chrome browser extension called mod header which allows us to modify the HTTP headers when we're testing our API.

Setting up the Project to Local Machine and Github

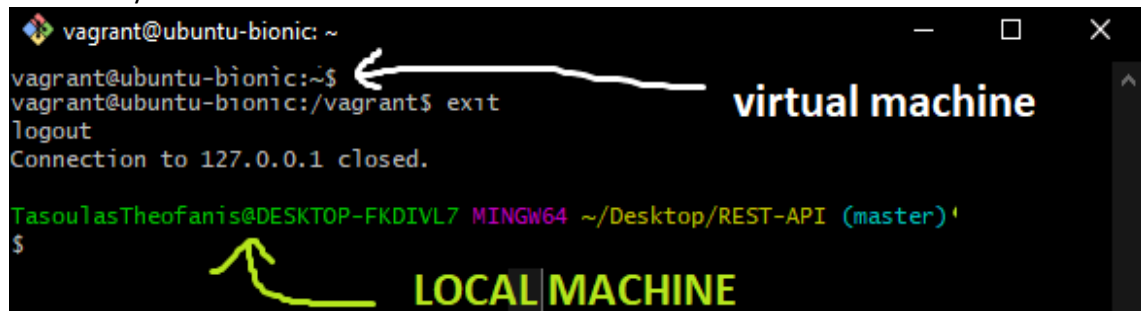
1. Download and Install: Git, Oracle VM Virtual Box, Vagrant, Atom and the Chrome browser extension called mod header.
2. Create a workspace (a folder) and give it a name (e.g. REST-API).
3. Open Atom, then drag and drop the workspace folder on Atom.
4. Open Git Bash, go to the workspace destination with the cd command.
5. In Git Bash type **git Init** to initialize a git local repository.
6. Restart Atom and create a "README.md" file (optional) to describe your project and a ".gitignore" file (optional) to have a license.
7. In Git Bash type **git add .** to add the previously created files to the git project
8. Type **git commit -am "initial commit"** in order to commit the changes to the git project. The **-am** "initial commit" is just a message, describing the changes that we made.
9. Type **ssh-keygen -t rsa -b 4096 -C "your_email.com"** to create a SHA256 key. Then, add an extra passphrase (optional) for further security. Now, if you can type **ls ~/.ssh** to check if your private (id_rsa) and public (id_rsa.pub) key exist.
10. Type **cat ~/.ssh/id_rsa.pub** and then copy your public key.
11. Go to Github.com, go to your profile (top right of the screen), go to your settings, go to SSH and GPG Keys, click add new, paste your public key on the "Key" field and type the name of the machine that you are authenticating this key to the "title" field and hit "add key".
12. On Github.com create a Github repository. After creating it, you will see under the "...or push an existing repository from the command line" 2 commands.



13. Go to Git Bash and type the 1st command **git remote add origin git@github.com:username/project-name.git**. Before typing **git push -u origin master** to push your project to github, make sure you delete any secret keys at your files (e.g. at settings.py in the api/api and project_api folder)!! After committing to github, restore the secret keys, so the project runs without errors.

Setting up the Virtual Machine and the Server

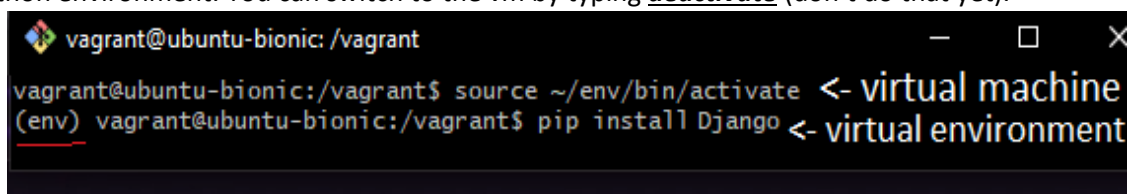
1. Go to Git Bash and type **vagrant init ubuntu/bionic64** to create a new vagrant file.
2. Go to Atom and to the vagrant file and replace its content with the content of this one [here](#).
3. Go to Git Bash and type **vagrant up** to download the base image (or in other words the virtual machine (vm) on our local machine) specified at the vagrant file (in our case an ubuntu bionic64).
4. Type **vagrant ssh** to connect to the virtual machine. The command line tells you whether you work on your local or your vm.



The screenshot shows a terminal window with two distinct environments. The top part, labeled 'virtual machine' with a white arrow, shows the prompt 'vagrant@ubuntu-bionic: ~' and the command 'exit' being entered, followed by 'logout' and 'Connection to 127.0.0.1 closed.' The bottom part, labeled 'LOCAL MACHINE' with a green arrow, shows the prompt 'TasoulasTheofanis@DESKTOP-FKDIVL7 MINGW64 ~/Desktop/REST-API (master)'.

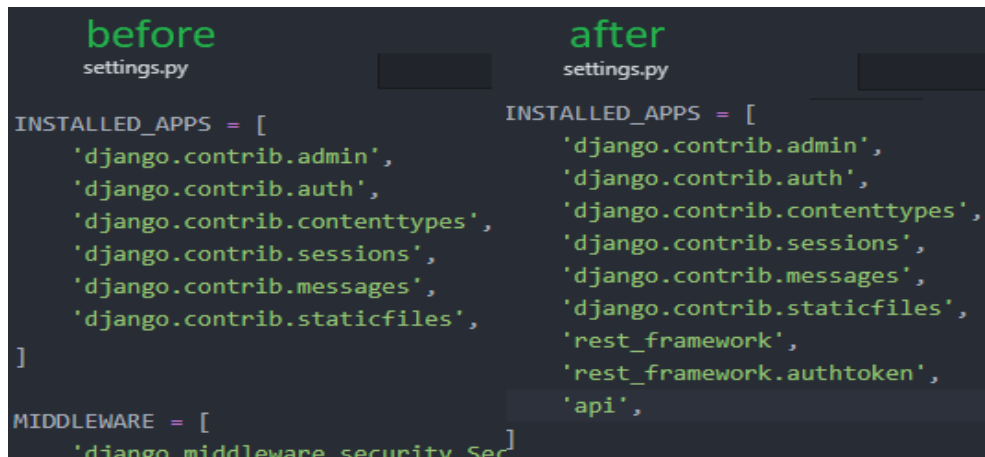
Keep in mind that Vagrant synchronizes files from your virtual to your local machine. Pretty cool huh?

5. In the vm, type **cd /vagrant** to switch to the vagrant directory. Type **python -m venv ~/env** or **python3 -m venv ~/env** to create a python virtual environment (install python on vm) on a destination. In this case, we want to create the python environment to the vagrant server, because we don't want to synchronize this python environment to our local machine. For instance, if we want to destroy and recreate the vagrants over from the scratch, you can do that from a fresh python environment. That's why, we created the "env" file, to store the python environment there.
6. Type **source ~/env/bin/activate** to work in the "env" environment. Now we are on the "env" virtual python environment. You can switch to the vm by typing **deactivate** (don't do that yet).



The screenshot shows a terminal window with the prompt 'vagrant@ubuntu-bionic: /vagrant'. The command 'source ~/env/bin/activate' is entered, and the prompt changes to '(env) vagrant@ubuntu-bionic: /vagrant\$'. Then, the command 'pip install Django' is entered. Annotations with arrows point to the prompt change and the command, labeling them as 'virtual machine' and 'virtual environment' respectively.

7. Type **pip install Django** and then **pip install DjangoRESTframework**. Now we downloaded the Django and the djangoframework packages.
8. **django-admin.py startproject api project**, to create a new Django project (a folder with the essential files) called api_project right here in the root location. If you want to add your project to a subfolder just remove the 'dat' and type **django-admin.py startproject api project**.
9. Type **python manage.py startapp api** to create an api app folder with the essential files. In this case, the folder will be named "api".
10. In order to enable an app in a project, open the "api" folder, open the "settings.py" file, which is the configuration file for the Django. Find the line with the "INSTALLED_APPS = []" and add after the already installed apps the "rest_framework", "rest_framework.authtoken", "api". That's how you add apps to your Django project.



```
before
settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
]

ROOT_URLCONF = 'django.urls.conf.urls'

WSGI_APPLICATION = 'django.wsgi.application'

SECRET_KEY = 'django-insecure-...'

DEBUG = True

ALLOWED_HOSTS = []

after
settings.py

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'api',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
]
```

11. Type **python manage.py runserver 0.0.0.0:8000** on the “env” virtual python environment, to start the Django development server in port 8000. You can press ctrl + c to close the server (don’t do that yet).
12. Open your browser and go to <http://127.0.0.1:8000/>. MAGIC!
13. Don’t forget to **git add .** and **git commit -am "Created django project and app"** and **git push origin**. Before typing **git push -u origin master** to push your project to github, make sure you delete any secret keys at your files (e.g. at settings.py in the api/api and project_api folder)!! After committing to github, restore the secret keys, so the project runs without errors.

Create Models

In Django we use models to describe the data we need for our application. Django, then, uses these models to set up and configure our database, in order to store our data effectively. Each model in Django maps to a specific table within our database. Django handles the relationship between our models and the database for us, thus we never need to write any sequel statements or interact with the database directly. So, let's get started and create our first models for our project.

1. In case it doesn't exist, create a "models.py" file in api folder. In this file we include the user and user manager profile model. Fill the file with the following code:

```
from django.db import models
from django.contrib.auth.models import BaseUserManager
from django.contrib.auth.models import AbstractBaseUser
from django.contrib.auth.models import PermissionsMixin

class UserProfileManager(BaseUserManager):
    """Manager for user profiles"""
    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) #lowercase letters before the '@'
        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):
        """Create and save 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

class UserProfile(AbstractBaseUser, PermissionsMixin):
    """Database model for users in the system"""
    email = models.EmailField(max_length=255, unique=True) #email is the username
    name = models.CharField(max_length=255)
    is_active = models.BooleanField (default=True)
    is_staff = models.BooleanField (default=False)

    objects = UserProfileManager()
    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['name']
```

```
def get_full_name(self):
    """Return full name of user"""
    return self.name

def __str__(self):
    """Return string representation of the user"""
    return self.email
```

2. Go to settings.py and find the "STATIC_URL = '/static/'". Below it, write:
AUTH_USER_MODEL = 'api.UserProfile' #go to the 'api' workspace and use the UserProfile model for authentication and registration in our project
3. Django manages the database with a migration file that stores all the models from our project. So, open git bash and connect to vagrant machine (if you are not already connected) , by typing **vagrant up** and **vagrant ssh**. Lastly, connect to the virtual environment by typing **cd /vagrant** and **source ~/env/bin/activate**.
4. Then, type **python manage.py makemigrations api** to create a migration file in our project.
5. Type **python manage.py migrate** to run all the migrations in our projects.
6. Then **python manage.py createsuperuser** to create a super user (like an administrator). Enter your email address and name and password. This superuser has access to all current apps, but when you add new apps, you have to manually give him access.
7. Go to api folder, open or create the admin.py file. The file should include this:

```
from django.contrib import admin
from api import models #from the api folder
```

```
admin.site.register(models.UserProfile) #register the UserProfile model with the admin
```

8. Type **python manage.py runserver 0.0.0.0:8000** on the "env" virtual python environment, to start the Django development server in port 8000.
9. Open your browser and go to <http://127.0.0.1:8000/admin/>. Here, you type the email and password for the superuser that we created before. Now, you can see the 3 apps that we have, the AUTH_TOKEN, the AUTHENTICATION AND AUTHORIZATION and the API.

Create API Views

The API View allows us to define functions that match the standard HTTP methods, like get and post. In addition, it allows us to create more complex stuff, such as access and edit to local files or data, call other APIs, run algorithms, render a synchronous response, etc. Therefore, it gives us control over our application.

1. In case it doesn't exist, create a "views.py" file in the api folder. In this file we include the user and user manager profile model. Fill the file with the following code:

```
from rest_framework.views import APIView
from rest_framework.response import Response

class HelloAPIView(APIView):

    def get(self, request, format=None):
        an_apiview = [ #a list of messages to show up
            'Uses HTTP methods as function (get, post, patch, put, delete)',
            'Is similar to a traditional Django View',
        ]
        return Response({'message': 'Hello!', 'an_apiview': an_apiview})
```

2. In api_project folder open the "urls.py". This is the entry point for all urls in our project. Edit it as you see in the picture below:

| Before | After |
|---|---|
| 16 from django.contrib import admin | 16 from django.contrib import admin |
| 17 from django.urls import path | 17 from django.urls import path, include |
| 18 | 18 |
| 19 urlpatterns = [| 19 urlpatterns = [|
| 20 path('admin/', admin.site.urls), | 20 path('admin/', admin.site.urls), |
| 21] | 21 path('api/', include('api.urls')), |
| 22 | 22] |
| 23 | 23 |

Now, with the **include** we gave the ability to include additional local paths. With the **path('api/', include('api.urls'))**, we gave the ability to read the paths from a file called urls.py (in the api folder) at the <http://127.0.0.1:8000/api/>.

3. Create a "urls.py" file in the api folder. Fill the file with the following code:

```
from django.urls import path
from api import views

urlpatterns = [
    path('hello-view/', views.HelloAPIView.as_view()),
]
```

4. If you type **python manage.py runserver 0.0.0.0:8000** and then open your browser at <http://127.0.0.1:8000/api/hello-view/>, you will view your messages.

Create Serializer

Serializer is a feature from the Django rest framework that allows you to easily convert data inputs into Python objects and the reverse. It's like you define the various fields that you want to expect for the input for your API. As a result, if we're going to add a post or an update functionality to our Hello API view, then we need to create a serializer to receive the content that we post to the API. Point out that serializers also take care of validation rules. Thus, if you want to accept a certain type of values in a certain field (e.g. accept only numbers in a field called "age"), serializer makes sure that the API passes the correct type that you want for that field.

1. Create a serializer.py file into the api folder. This will contain the following code:

```
from rest_framework import serializers

class HelloSerializer(serializers.Serializer):
    """Serializes a name field for testing our APIView"""
    name = serializers.CharField(max_length=10)
```

2. Go to view.py and add those packages:

```
from rest_framework import status
from api import serializers #from the folder api, include the serializers.py file
```

and the following functions in the class HelloAPIView:

```
serializer_class = serializers.HelloSerializer #this configures that we have a serializer

def post(self, request):
    """Create a hello message with our name"""
    serializer = self.serializer_class(data=request.data) #retrieve the serializer class, the data
    are passed as a request
    if serializer.is_valid():
        name = serializer.validated_data.get('name') #check if the data is the correct type (in
    serializers.py we defined name to be Char with max_length=10)
        age = serializer.validated_data.get('age')
        message = f'Hello {name}, you are {age} years old'
        return Response({'message':message}) #return a dictionary
    else:
        return Response(serializer.errors, status = status.HTTP_400_BAD_REQUEST ) #in case
    of a bad request

def put(self, request, pk=None):
    """Fully updating an object (all of its fields)"""
    return Response({'method': 'PUT'})

def patch(self, request, pk=None):
    """partial update of an object (some fields)"""
```



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

```
def delete (self, request, pk=None):
```

```
    """Delete object"""
```

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

3. If you want to test it, type **vagrant up** and **vagrant ssh** to open up the virtual machine and then type **cd /vagrant** and **source ~/env/bin/activate** to enter the virtual environment. After that, type **python manage.py runserver 0.0.0.0:8000/** and try out the new features that we've added at the <http://127.0.0.1:8000/api/hello-view/>.

Create Viewsets

Just like APIViews, Viewset allows us to write the logic for our endpoints. However, APIViews define functions for HTTP methods. Viewsets define functions for the API objects.

1. Go to views.py file and add this package:

```
from rest_framework import viewsets
```

and this new class:

```
class HelloViewSet(viewsets.ViewSet):  
    """Test API ViewSet"""  
  
    serializer_class = serializers.HelloSerializer  
  
    def list(self, request):  
        a_viewset = [  
            'User actions (lists, 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})  
  
    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')  
            age = serializer.validated_data.get('age')  
            message = f'Hello {name}, you are {age} years old!'  
            return Response({'message': message})  
        else:  
            return Response(  
                serializer.errors,  
                status=status.HTTP_400_BAD_REQUEST  
            )  
  
    def retrieve(self, request, pk=None):  
        """Handle getting an object by its Primary Key"""  
        return Response({'http_method': 'GET'})  
  
    def update(self, request, pk=None):  
        """ Update objects """
```

```

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

def partial_update(self, request, pk=None):
    """Partially Update objects"""
    return Response({'http_method': 'PATCH'})

def destroy(self, request, pk=None):
    """Remove object"""
    return Response({'http_method': 'DELETE'})

```

2. In order to make Viewsets accessible to our api, go to api folder and modify the urls.py file. To make it simpler, delete all the code and replace it with this:

```

from django.urls import path, include
from rest_framework.routers import DefaultRouter
from api import views #from api folder import views.py file

router = DefaultRouter()
router.register('hello-viewset', views.HelloViewSet, basename='hello-viewset')

urlpatterns = [
    path('hello-view/', views.HelloAPIView.as_view()),
    path('', include(router.urls))
]

```

What we actually did is that we imported a router package, we created a router that was linked with our viewset and then we linked the router with all the urls.

3. If you want to test it, type **vagrant up** and **vagrant ssh** to open up the virtual machine and then type **cd /vagrant** and **source ~/env/bin/activate** to enter the virtual environment. After that, type **python manage.py runserver 0.0.0.0:8000/** and try out the new features that we've added in <http://127.0.0.1:8000/api/hello-viewset/> and in <http://127.0.0.1:8000/api/hello-viewset/1/> for the GET method and in <http://127.0.0.1:8000/api/hello-viewset/2/> for the PUT method etc.

Create API Profiles

Now we will be creating a profile to handle the registration of new users in the system. This will include validating the profile data, a way to search for users by email or name, a log in feature, an update for the profile of the logged in user users in the and finally we'll provide a way for users to delete their own profile.

1. Go to serializer.py file and add this package:

```
from api import models
```

and this new class, which defines the UserProfile as a model:

```
class UserProfileSerializer(serializers.ModelSerializer):  
    """Serializes a user profile object"""  
  
    class Meta: #configures the serializer to point to a specific model in our project site  
        model = models.UserProfile  
        fields = ('id', 'email', 'name', 'password')  
        extra_kwargs = {  
            'password': {  
                'write_only': True, #we dont want the password to be shown  
                'style': {'input_type': 'password'}  
            }  
        }  
  
    def create(self, validated_data):  
        """Create and return a new user"""  
        user = models.UserProfile.objects.create_user(  
            email = validated_data['email'],  
            name = validated_data['name'],  
            password = validated_data['password']  
        )  
        return user
```

2. We want to restrict users from editing others' data. Therefore, create a permissions.py file in the api folder, with this code:

```
from rest_framework import permissions  
  
class UpdateOwnProfile(permissions.BasePermission):  
    """Allow user to edit their own profile"""  
    def has_object_permission(self, request, view, object):  
        """check if user has the permission to do that"""  
        if request.method in permissions.SAFE_METHODS: #is it in safe methods (e.g. GET)?  
            return True
```

```
else:
    return object.id == request.user.id #check if the data that user tries to access are
his/her own data
```

3. Now we want to create a viewpoint to access the serializer through an endpoint. Therefore, open your views.py file and add these packages:

```
from api import models
from api import permissions
from rest_framework.authentication import TokenAuthentication
from rest_framework import filters
from rest_framework.authtoken.views import ObtainAuthToken
from rest_framework.settings import api_settings
```

and this new class for the profile of the users and their features:

```
class UserProfileViewSet(viewsets.ModelViewSet):
    """Handle creating 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',)
```

and this new class for the authentication of the users:

```
class UserLoginAPIView(ObtainAuthToken):
    """Handle creating user authentication tokens"""
    renderer_classes = api_settings.DEFAULT_RENDERER_CLASSES
```

4. Open urls.py. In order to register our new viewset with a new url, add this router to your code:

```
router.register('profile', views.UserProfileViewSet)
```

and this new path to enable the endpoint for the authentication of the users:

```
path('login/', views.UserLoginAPIView.as_view())
```

5. Now, we can run the api and create some profiles. Go to <http://127.0.0.1:8000/api/profile/> and test for yourself.

Let's create a new user

The screenshot shows the Django REST framework API browser interface. The top section displays a GET request to `/api/profile/` with a successful response (HTTP 200 OK) containing a JSON array with one user profile: `[{"id": 1, "email": "tasoulastheofanis@outlook.com", "name": "Theofanis"}]`. Below this, there is a form for creating a new user profile. The form fields are: Email (`p15taso@ionio.gr`), Name (`Tasoulas`), and Password (masked with dots). A tooltip indicates: "Make a POST request on the User Profile List resource". A "POST" button is visible at the bottom right of the form.

The bottom section shows the same API browser interface after a POST request. The response is "HTTP 201 Created" with a JSON object: `{"id": 2, "email": "p15taso@ionio.gr", "name": "Tasoulas"}`. The "User Profile List" header is visible, along with "OPTIONS" and "GET" buttons.

Hit the GET button, to see all the profiles. Then, hit filters to search someone.

The screenshot shows the Django REST framework API browser interface. The top section displays a GET request to `/api/profile/` with a successful response (HTTP 200 OK) containing a JSON array with two user profiles: `[{"id": 1, "email": "tasoulastheofanis@outlook.com", "name": "Theofanis"}, {"id": 2, "email": "p15taso@ionio.gr", "name": "Tasoulas"}]`. Below this, there is a "Filters" dialog box with a search input field containing "Theofanis" and a "Search" button. The "User Profile List" header is visible, along with "Filters", "OPTIONS", and "GET" buttons.

← → ↻ 127.0.0.1:8000/api/profile/ ASP

Django REST framework

Api Root / User Profile List

User Profile List

Handle creating and updating profiles

OPTIONS GET

POST /api/profile/

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

{
  "id": 2,
  "email": "p15taso@ionio.gr",
  "name": "Tasoulas"
}
```

Now, let's try to log in profile 2 (username is the email)

User Login Api - Django REST fra x +

← → ↻ 127.0.0.1:8000/api/login/ ASP

Django REST framework tasoulastheofanis@outlook.com

Api Root / User Login Api

User Login Api

Handle creating user authentication tokens

OPTIONS

GET /api/login/

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "detail": "Method \"GET\" not allowed."
}
```

Raw data HTML form

Username

Password

POST

User Login Api - Django REST fra x +

← → ↻ 127.0.0.1:8000/api/login/ ASP

Django REST framework tasoulastheofanis@outlook.com

Api Root / User Login Api

User Login Api

Handle creating user authentication tokens

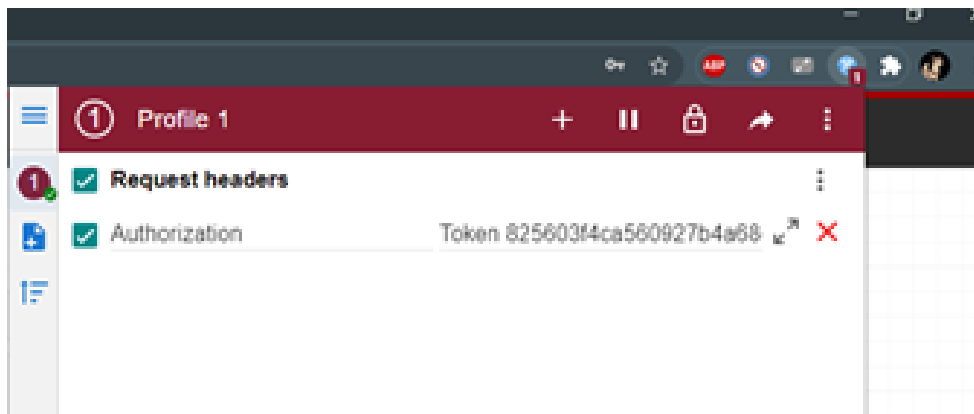
OPTIONS

POST /api/login/

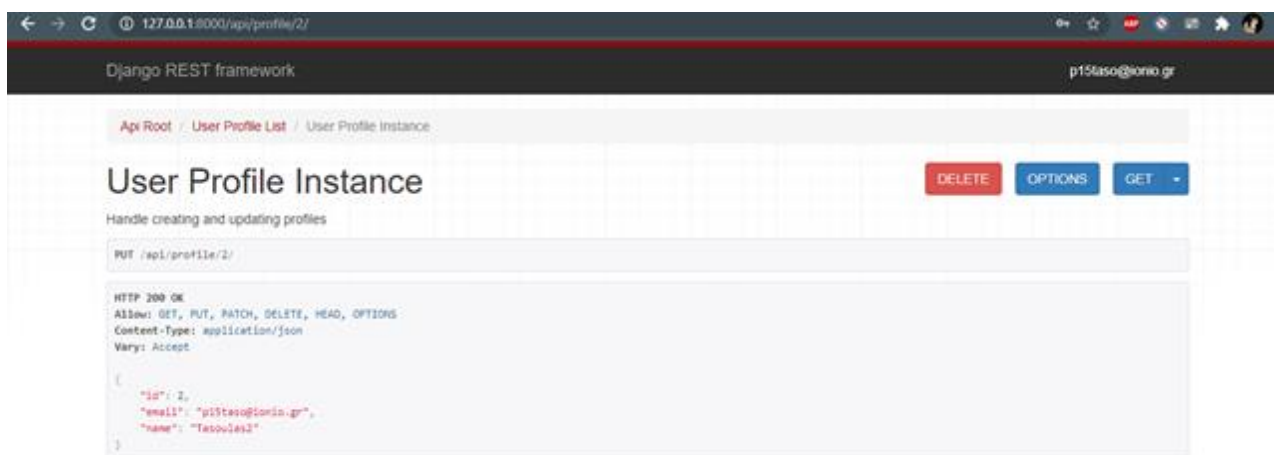
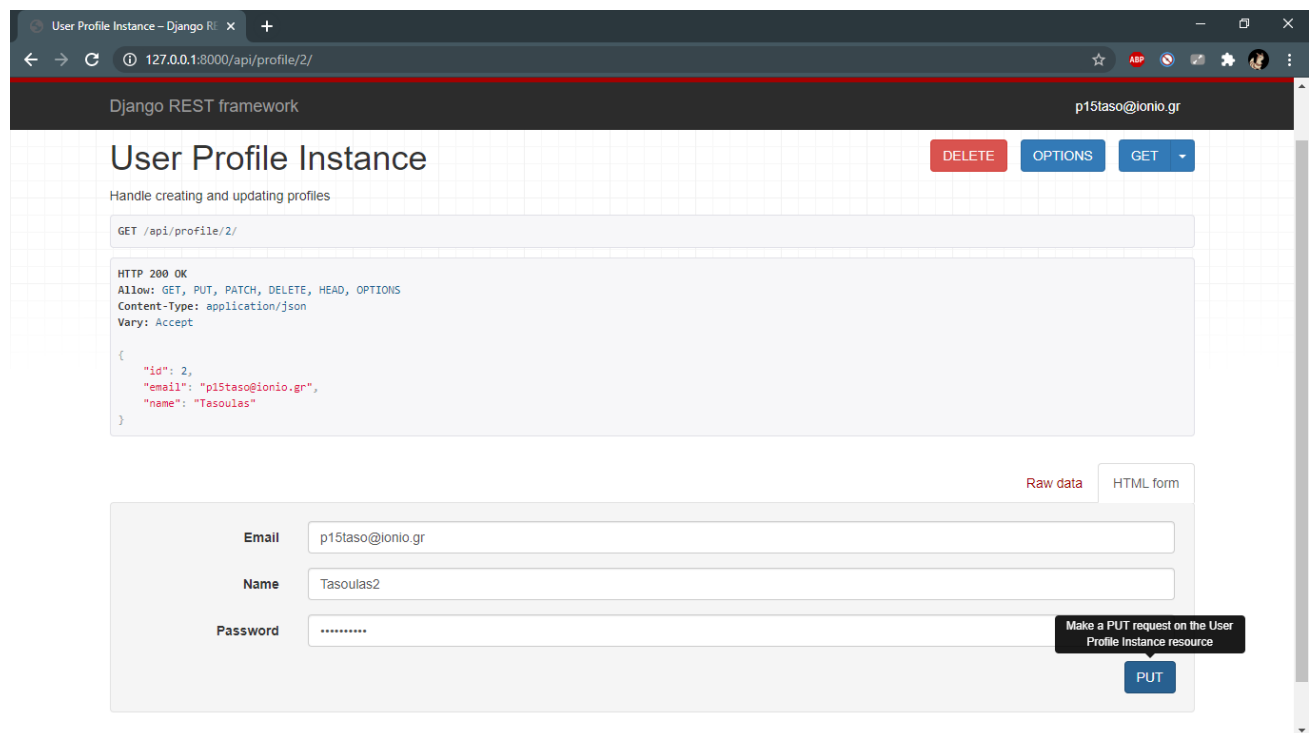
```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "token": "825603f4ca560927b4a68d9642db6add37d0eaa8"
}
```

Ok, cool. Can we change our data? Yes, but we have to authenticate ourselves. We will do that with the mod header app in chrome. Open it and add the token as: **token 8256.....**



Now that we authenticated ourselves as p15taso@ionio.gr, you can see we can change our profile's values.



Can we edit the others' profile data? If we go to <http://127.0.0.1:8000/api/profile/1/> when we are logged in as p15taso@ionio.gr we don't see any PUT button. But still, let's hit the OPTIONS button and try to change them.

The screenshot shows the Django REST framework API browser interface. The browser address bar displays `127.0.0.1:8000/api/profile/1/`. The page title is "User Profile Instance - Django REST framework". The user is logged in as `p15taso@ionio.gr`. The main content area shows the response for an OPTIONS request:

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

{
  "name": "User Profile Instance",
  "description": "Handle creating and updating profiles",
  "renders": [
    "application/json",
    "text/html"
  ],
  "parsers": [
    "application/json",
    "application/x-www-form-urlencoded",
    "multipart/form-data"
  ]
}
```

Below the response, there are tabs for "Raw data" and "HTML form". The "HTML form" tab is active, showing a form with the following fields:

- Email: `tasoulastheofanis@outlook.com`
- Name: `Theofanis22`
- Password: `.....`

A tooltip message says "Make a PUT request on the User Profile Instance resource". A "PUT" button is visible at the bottom right of the form.

The screenshot shows the Django REST framework API browser interface. The browser address bar displays `127.0.0.1:8000/api/profile/1/`. The page title is "User Profile Instance - Django REST framework". The user is logged in as `p15taso@ionio.gr`. The main content area shows the response for an OPTIONS request:

```
HTTP 403 Forbidden
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "detail": "You do not have permission to perform this action."
}
```

Below the response, there are tabs for "Raw data" and "HTML form". The "HTML form" tab is active, showing a form with the following fields:

- Email: `tasoulastheofanis@outlook.com`
- Name: `Theofanis22`
- Password: `.....`

A tooltip message says "Make a PUT request on the User Profile Instance resource". A "PUT" button is visible at the bottom right of the form.

As you can see, we don't have permission. Can we delete this user?

The screenshot shows the Django REST framework API browser interface. The browser address bar displays `127.0.0.1:8000/api/profile/1/`. The page title is "User Profile Instance - Django REST framework". The user is logged in as `p15taso@ionio.gr`. The main content area shows the response for a DELETE request:

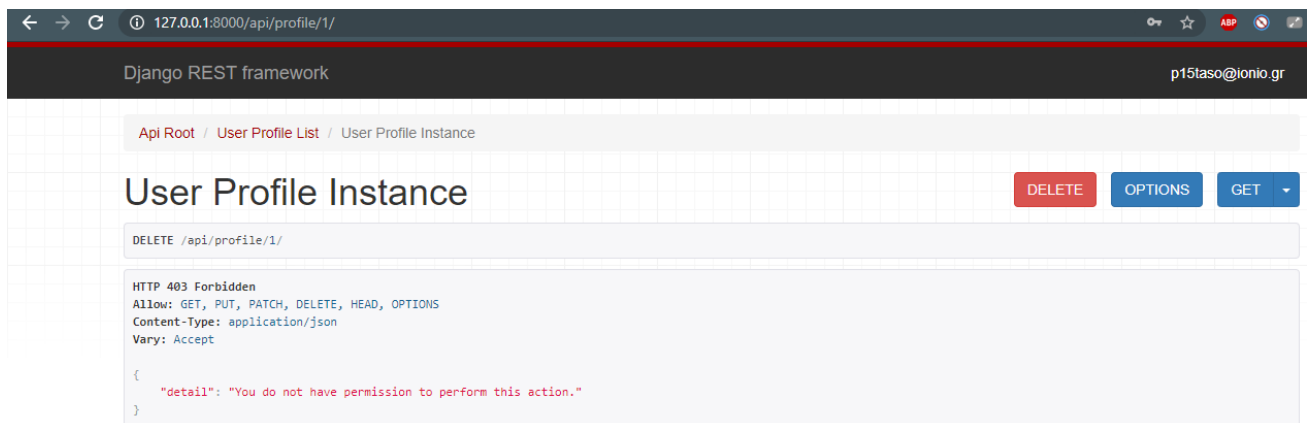
```
HTTP 403 Forbidden
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "detail": "You do not have permission to perform this action."
}
```

Below the response, there are tabs for "Raw data" and "HTML form". The "HTML form" tab is active, showing a form with the following fields:

- Email: `tasoulastheofanis@outlook.com`
- Name: `Theofanis22`
- Password: `.....`

A tooltip message says "Make a PUT request on the User Profile Instance resource". A "PUT" button is visible at the bottom right of the form.



Of course not. But we can delete ourselves.

