Intro – Few words for the Technologies

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.

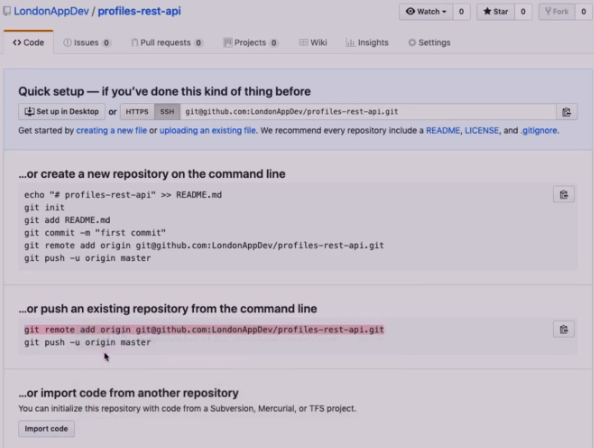
We're going to use Python and a python web framework called Django. On top of Django we're going to use another framework called the Django rest framework. The Django rest framework provides a set of features for making a standard rest api and for building a standard web app.

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

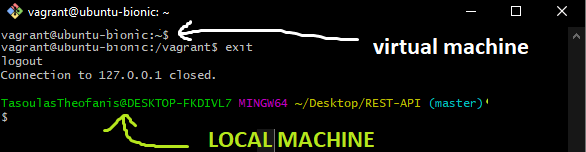
1. Download and Install: Git, Oracle VM Virtual Box, Vagrant, Atom and the Chrome browser extension called mod header.
2. Create a workspace (a folder) and give it a name (e.g. REST-API).
3. Open Atom, then drag and drop the workspace folder on Atom.
4. Open Git Bash, go to the workspace destination with the cd command.
5. In Git Bash type ***git Init*** to initialize a git local repository.
6. Restart Atom and create a “READ\_ME.md” file (optional) to describe your project and a “.gitgnore” 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 ~/.sha*** 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.



1. Go to Git Bash and type those 2 commands (***git remote add origin git@github.com:****username/project-name****.git*** and ***git push -u origin master***).

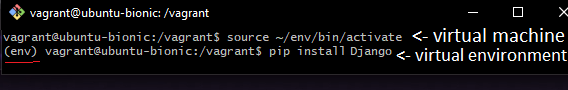
Setting up the Virtual Machine and the Server

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](https://gist.github.com/LondonAppDev/199eef145a21587ea866b69d40d28682).
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.

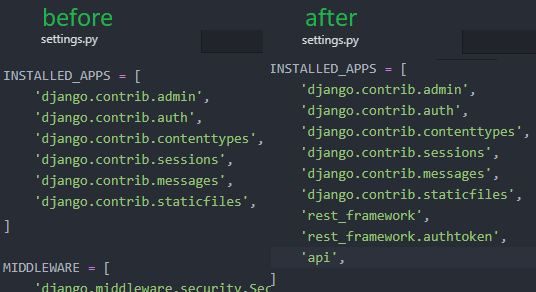


Keep in mind that Vagrant synchronizes files from your virtual to your local machine. Pretty cool huh?

1. 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 out local machine. For instance, if we want 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.
2. 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).



1. Type ***pip install Django*** and then ***pip install Djangorestframework***. Now we downloaded the Django and the djangoframework packages.
2. ***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***.
3. 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”.
4. 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.



1. 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).
2. Open your browser and go to <http://127.0.0.1:8000/>. MAGIC!
3. Don’t forget to ***git add .*** and ***git commit -am "Created django project and app"*** and ***git push origin***.

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):**

**"""Rertieve full name of user"""**

**return self.name**

**def \_\_str\_\_(self):**

**"""Return string representation of the user"""**

**return self.email**

1. 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**

1. 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***. Latsly, connect to the virtual environment by typing ***cd /vagrant*** and ***source ~/env/bin/activate***.
2. Then, type  ***python manage.py makemigrations api*** to create a migration file in our project.
3. Type  ***python manage.py migrate*** to run all the migrations in our projects.
4. 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.
5. 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**

1. 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.
2. 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**

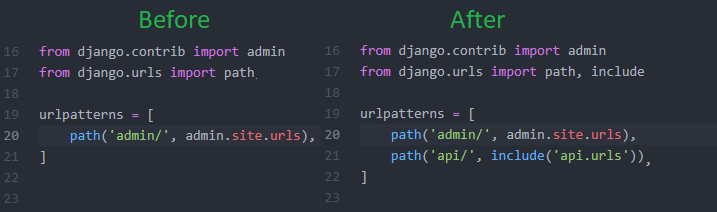
**'Uses HTTP methods as function (get, post, patch, put, delete)',**

**'Is similar to a traditional Django View',**

**]**

**return Response({'message': 'Hello!', 'an\_apiview': an\_apiview})**

1. In api\_project folder ope the “urls.py”. This is the entry point for all urls in our project. Edit it as you see in the picture below:



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/>.

1. Create a “ulrs.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()),**

**]**

1. 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 except 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)**

1. 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 reuest**

**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'})**

1. 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/>](http://127.0.0.1:8000/api/).

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 functionallity 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'})**

1. 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.

1. 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.